ADOBE® DIRECTOR® BASICS

Legal notices

For legal notices, see http://help.adobe.com/en_US/legalnotices/index.html.

Contents

Chapter 7: Unicode support in Director

Chapter 1: Preface

'Adobe Director basics' is an introductory guide for a 3D/game developer to get started with Adobe Director. This document is a self-learning guide, with examples and samples, on how to use Adobe Director to build simple to complex multimedia applications.

Before you begin, install Adobe Director 11.5 and the latest patch (Help menu > Updates). By installing Director, you can view the samples along with the code, and know about various possibilities by trying out different properties as explained in the document. You can also use Shockwave Player to play the samples, without viewing the code.

Note: *Ensure that you install the latest version of Shockwave Player. Else, an error occurs and the samples do not play.*

This is the first version of the document, which explains some of the basic concepts of the following features:

- 3D (Basics, Controlling appearance, and Controlling action)
- Audio mixers and sound objects
- Asynchronous programming
- Unicode support in Director

This document is intended to be used in conjunction with Adobe Director Using Guide and Adobe Director Scripting Dictionary.

Credits

The document is a collaborative effort involving James Newton, Adobe Director pre-release community, and Adobe.

The structure and content of this document were proposed and written by James Newton, a power-user and a renowned Director expert. The Adobe Director pre-release community has lent its support by reviewing the structure of the document and helping us finalize the flow of content.

The Adobe documentation team has done some minor edits to content, wherever necessary, to comply with the Adobe standards.

Feedback

If you have any feedback, complaints, or suggestions about an article, please leave a comment at the end of the article.

Note: *You need to log in with an Adobe ID to provide feedback.*

Chapter 2: 3D basics

Adobe® Director® lets you bring robust, high-performance 3D graphics to the web. With Director, you can develop a wide spectrum of 3D productions, ranging from simple text handling to interactive product demonstrations to complete immersive game environments. Using Shockwave® Player, users can view your work on the web with Microsoft® Internet Explorer®, or other browsers that support web packaging.

Director lets you detect the capabilities of the user's system and adjust playback demands accordingly. A powerful computer with 3D hardware acceleration brings the best results, but users can successfully use Director movies with 3D on most Mac® or Windows® hardware platforms. The faster the computer's graphics processing, the better the results. The ability to adjust for client-side processing power makes Director ideal for web delivery.

What is Shockwave3D?

Shockwave3D allows you to simulate a three-dimensional scene on a two-dimensional screen.

In the real world, there are objects and sources of light. You can move around and see the objects from different angles. Objects that are further away appear smaller than objects that are close to you. The same object looks different when the lighting is different.

In the real world, objects are solid, and they obey the laws of physics. Some objects, like trees and buildings, are static. Other objects, like human beings and chairs can move or be moved. Some objects, like clothes and paper can easily be deformed. Real objects can be broken apart, exploded, burnt, or joined together. Real objects can be made to behave in an infinite number of ways. Often, you cannot predict what will happen in the real world. The real world is very complex.

In a simulated 3D scene, there are just pixels on a flat screen. If you want to give the illusion that these pixels represent solid objects moving in three-dimensional space, you have to cheat. You have to trick your brain to make it imagine real objects and not just pixels.

Shockwave3D does its best to trick your brain. A computer is not as fast as processing information as your brain, so Shockwave3D has to take shortcuts.

Time and motion in the real world is continuous. When you watch a film in movie theatre, you see 24 static images every second. Your brain merges these static images together and imagines that your eyes are seeing continuous motion. A typical computer monitor updates its image 60 times a second. Director calls 1/60th of a second a *tick*.

In a simulated 3D scene, you can divide time into tiny discrete chunks, and your brain will not notice. Let's imagine that the 2D representation of the 3D scene is updated 60 times a second. Let's also imagine that the scene is from a first-person action game. The player's viewpoint changes all the time. There are moving characters and objects. None of this action can be predicted in advance.

Sixty times per second, Shockwave3D has to calculate the new position of the player's viewpoint, and the new position of each object in the scene. It may have to do complicated mathematical calculations to determine how one object falls and whether it has collided with another object. When all the calculations have been done, Shockwave3D is ready to set the color of each of the pixels on the screen.

If the screen is big and there are a lot of moving objects, this process requires a lot of computer power. The 3D section of the Director documentation will help you understand how to make good use of the limited power of a computer to create the best illusion of a 3D scene.

Organization of the 3D documentation

3D Basics

If you have never worked with 3D before, then just keep reading. This first chapter, 3D Basics, explains to you in simple language how Shockwave3D simulates objects in the real world. This chapter also explains where to find all the windows, tools, and objects that you will need to build a virtual 3D world.

Controlling Appearance

You can think of a virtual 3D world in two completely different ways: as an artist or as an engineer. As an artist, you will be concerned with what the virtual 3D world looks like. As an engineer, you will be concerned about how the 3D objects act and interact with each other.

The Controlling Appearance chapter explains how to think like an artist. You will learn:

- How the shape of a 3D object is defined
- How to group objects together so that they move together as one object
- How to add detail and color to the surface of a 3D object
- How to give the illusion of light and shade

By the end of this chapter, you will understand how to create a world of static objects that look almost real.

Controlling Action

The Controlling Action chapter teaches you to think like an engineer. To help you simulate actions and interactions, Shockwave3D provides you with many tools and devices. You can glue virtual objects together, you can join objects together with hinges, you can link objects together with invisible springs.

You will learn:

- How to move objects around
- How to move the user's viewpoint like a movie camera
- How to select an object by clicking on it
- How to make virtual objects appear to be solid
- How to make virtual objects behave as if they had weight
- How to simulate clouds, smoke, flames, fireworks, and other things that are made of many tiny particles
- How to simulate cloth and other soft items

A note about mathematics

Some of this interaction can be achieved without any knowledge of the mathematics used to describe 3D objects and motions. However, if you understand a little 3D mathematics, you can create virtual worlds which are much more convincing. Understanding mathematics is the key to being a good engineer.

3D mathematics is based on the well-known operations of addition, subtraction, multiplication, and division. The mathematics themselves are not very complicated. However, to use 3D mathematics effectively, you need to have a clear image in your head of what operation means in the 3D space. The difficulty is that any notes or drawings that you make will be on 2D paper or a 2D screen. To visualize your calculations clearly, you may want to make simple models out of cardboard, wire and string. To be a good mathematician, you have to be something of an artist.

Introduction to 3D

Several Adobe® Director® features let you create a 3D movie:

A 3D cast member contains a complex internal structure that includes model resources, models, lights, and cameras. Each of these objects has its own array of properties.

Director lets you convert 2D text to 3D and then work with the 3D text as you would with any other 3D cast member. You can apply behaviors to the 3D text, manipulate it with Lingo or JavaScript™ syntax, and view and edit it in the Shockwave® 3D window. You can also add extruded 3D text to a 3D cast member.

Director comes with a library of behaviors that let you build and control a 3D environment without any knowledge of Lingo or JavaScript syntax. Although scripting is still required for complex projects, you can build simple 3D movies with behaviors alone.

The 3D world

The end-user of a Shockwave3D movie sees a 3D sprite on the computer screen. You need to understand what is happening within the computer to create that image in the 3D sprite.

A 3D sprite is a visual representation of a Shockwave3D cast member. Each 3D cast member contains a complete 3D world. It can contain models (the objects that viewers see within the world) that are illuminated by lights and viewed by cameras. A sprite of a 3D cast member represents a specific camera's view into the world. Imagine that the 3D cast member is a room filled with furniture with cameras pointing in from several windows. A given sprite that is using that cast member will display the view from one of those cameras, but the room itself (the 3D cast member) remains the same regardless of which view is used.

Developers will often talk about a "3D world" or a "3D scene". Often this has the same meaning as "Shockwave3D cast member".

The key difference between 3D cast members and other cast members is that the models within the 3D world are not independent entities—they are not sprites. They are integral parts of the 3D cast member.

Your movies can use 2D and 3D cast members simultaneously. For example, a product demonstration movie might consist of a 3D cast member that represents the product and one or more 2D controls that allow users a virtual tryout of the product.

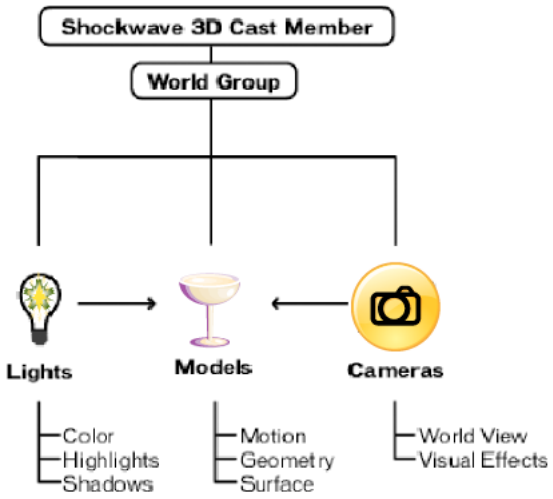
Creating a world

In most cases, a 3D designer will use a third-party 3D modeling application, such as Autodesk® 3ds Max®, to create a W3D file. When you import this 3D file into Director, it becomes a Shockwave3D cast member. To learn more about importing W3D files, see "External 3D Files" on page 64.

You can also create an empty Shockwave3D cast member in Director itself. If you are comfortable with programming, you can then build a virtual world piece by piece, using Lingo or JavaScript syntax. To learn more about creating 3D content with Director, see "Native 3D content" on page 71.

Basic 3D objects

A Shockwave3D cast member contains all the information that defines a virtual world. A Shockwave3D cast member is made up of a hierarchy of objects.



Shockwave3D cast member

In a very simple 3D world, you can find:

- A **camera** object, which represents the user's viewpoint. If there were no camera, there would be no way to see the virtual world.
- A **light** object, which defines a source of light. If there were no light object, there would be now way to create light and shade. Light and shade help the brain to understand the three-dimensional shape of objects.
- A 3D **model** object, which represents a physical object in space. If there were no 3D model, there would be nothing in the virtual world to see.
- A **modelResource** object, which defines the geometrical shape of the 3D model. If there were no modelResource, the model would become an invisible point in space.
- A **texture** object, which shows the color and patterns on the surface of the model. If you think of the model as a sculpture, you can think of a texture as paint on the surface of the sculpture. If there were no texture object, the model would appear in shades of one uniform color.
- A **shader** object, which defines how the surface of the model reacts to light. Where a texture defines color, a shader defines properties such as shininess, transparency, and reflection. You can think of a shader as representing the material that a model is made from. If there were no shader object, all models look as if they were made of the same uninteresting material.
- A **group** object, which defines how the model is related to other models and to the world itself. Every 3D cast member has a group named "World". By default, every model, camera and light is a **child** of the group("World"). You can make one model the child of another model. The two models will then move together as if they were linked together by an invisible bar. A model that is not a child of group("World") will not be visible to a camera that is a child of group("World"). If you want a model to disappear, you can remove it from the World.
- A **motion** is a pre-defined animation sequence that involves the movement of models or model components. Individual motions can be set to play by themselves or with other motions. For example, a running motion can be combined with a jumping motion to simulate a person jumping over a puddle. Motions are controlled by a Bones Player or a Keyframe Player attached to a model. Every 3D cast member has at least one motion named "DefaultMotion". This is a placeholder motion; it has a duration of 0 milliseconds.

Nodes

Only models are visible. The appearance of a model depends on the shaders and textures that are attached to it. You cannot see a camera, a light, or a group.

Models, lights, cameras and groups all have a specific position in space. The term **node** is used to describe any 3D object that has a position in space. Models, lights, cameras and groups are all nodes. You can link nodes together in **parent**-**child** relationships. The two nodes will then move together as if an invisible bar linked them. For example, might have a model called "Head" and a model called "Hat". You can make the hat model the child of a head model. When the head moves, the hat will move with it.

A model that is not a child of group("World") will not be visible to a camera that is a child of group("World"). If you want a model to disappear, you can remove it from the World. A 3D cast member can contain lights, models, groups, and cameras that have no parent. These nodes and their children will not be displayed in the sprite.

Optional 3D objects

A 3D cast member can also contain other optional objects:

- **Overlays** and **backdrops** are two-dimensional areas. An overlay will appear in front of all the 3D models in the virtual world. You can use an overlay to create a frame around a scene, to show a progress bar, to show text, to act as a button, or any number of other things. A backdrop will appear behind all the 3D models. You can use a backdrop to create a 2D background for the scene.
- **Modifiers** provide extra functionality for models. There are modifiers to:
  - Create realistic movements for characters (Bones Player, Keyframe Player, Mesh Deform)
  - Change the appearance of a model (Inker, Toon, Level of Detail, Subdivision Surfaces)
  - Create the illusion of solidity (Collision)
- **Physics objects** provide ways to control the interactions between models. The list of physics objects includes rigid bodies, terrains, springs, joints, and constraints.

Sound

A 3D world is silent. You need to create your 3D soundscape independently of the visual 3D world. You can add sound to a Director movie in many ways. Director 11.5 and later supports 5.1 sound. If your end-users have a 5.1 sound system connected to their computer, you can simulate sounds in three dimensions. If your end-users have headphones or stereo speakers, you can use stereo as an alternative.

See Sound, and in particular "Audio mixers and sound objects" on page 400 for more information on how Director can place sounds in 3D space.

3D Sprites

When you place a Shockwave3D cast member on the Stage, a 3D sprite is created. Every 3D sprite has at least one camera. The settings for the camera determine the view that appears in the 3D sprite.

Rect

By default, a 3D sprite is 320 x 240 pixels. Like other sprites, you can change the rect of the sprite by dragging the corners and edges. The bigger you make the sprite, the more computer processing power it will need to update the image. Director needs to calculate the color of every pixel in the sprite image every time the image is refreshed. More pixels mean more calculations. If you are designing a project for relatively slow computers, test whether the size of your 3D sprite is appropriate for the target computers.

DirectToStage

By default, the cast member of a 3D sprite is drawn "Direct To Stage" (DTS). This means that Director sends the image for the 3D sprite directly to the screen display driver controlled by your computer's operating system. This is very much faster than sending the image to Director's display system, where each sprite needs to be layered on top of other sprites. The disadvantage is that a 3D sprite will appear on top of all other sprites. Sprite inks will not work on a 3D sprite set to display direct to stage.

Suppose you want to show sprites on top of a 3D sprite. Suppose you want your 3D sprite to have a transparent background, so that you can see the sprites that are behind it. You can switch off the DTS display. The disadvantage is that your movie will run more slowly, since Director now has to do more work with inks and layers.

To learn more about setting the Direct To Stage property of a Shockwave3D cast member at "Using the Property Inspector for 3D" on page 14. For more information on this property, see `member3D.directToStage`.

Effect of DirectToStage on frame tempo

The effect of switching off Direct To Stage display depends on the operating system. In Windows, the movie may run at around 85 % of its maximum possible DTS speed.

In Mac OS 10.6 and later, Director 11.5 movies may be capped at 60 frames per second. Changes made in Mac OS 10.6 and Director 11.5 now limit the number of screen update requests. The screen will no longer be updated faster than the monitor can be refreshed.

When a Shockwave3D cast member is set to display Direct To Stage, and where there is no other animation, Director can ignore the monitor refresh rate. The movie can report a much faster frame tempo. Nonetheless, the end user will only see screen updates at the rate at which the monitor is refreshed.

If you set the movie's frame tempo to 60 frames per second or less, you can avoid unexpected playback results on all supported platforms.

Frames and Backgrounds

A DTS sprite will be rectangular and opaque. You will not be able to see any sprites behind it.

You may want to give the impression that a DTS 3D sprite is an irregular shape (not a rectangle). You can use overlays to create a border that continues the design for the background for the rest of your movie.

You may want to give the impression that the 3D models are floating above the background on the Stage. You can use a backdrop which fits seamlessly into the Stage background.

See "Overlays and backdrops" on page 47 for more details of both these features.

Controlling the 3D world

Elements of a 3D scene can be modified and manipulated with 3D behaviors. The Library Palette contains many behaviors that can be used to perform simple actions. You can find more information about these at "3D behaviors" on page 75.

In general, the behaviors in the Library Palette are a good way to control a single aspect of a 3D world. Behaviors are a good way to:

- Display a rotating logo
- Show a panoramic view of a scene
- Control the rotation of an object, so that the viewer can see it from all sides
- Play a pre-animated 3D scene
- Drag a single model with the mouse and other simple interactions.

More complex control of 3D world requires an understanding of scripting. Chapters "3D: Controlling appearance" on page 88 and "3D: Controlling action" on page 202 explain scripting for 3D worlds in more detail.

To create an interactive 3D game, for instance, you will need to write your own custom behaviors. You will probably find it useful to understand the principles of Object-Oriented Programming (OOP). In Director, you can use Parent Scripts and ancestors to create separate modules of code. Each module or Parent Script can deal with one specific feature of your interactive 3D world.

You can find an excellent introduction to Object-Oriented Programming in Lingo by Irv Kalb here. Irv Kalb's eBook does not specifically deal with controlling 3D worlds, but all the principles in the book are applicable.

2D and 3D workflows

Perhaps you are used to working with Director's 2D Stage, Score, Sprite, and Cast Member workflow. If so, and if you are new to 3D, then this section is for you. If you are new to Director, but have some knowledge of 3D design, then you will also find this section interesting. Also see Chapters Workspace and Score, Stage and Cast, to discover how Director's 2D workflow functions.

The 2D workflow

Director works best at assembling media that have been created in specialized applications. For creating images, you will probably use Photoshop. To create sounds, you can use Adobe Audition or Adobe Soundbooth.

For creating a complex 3D world, you will need a third-party 3D design application like 3ds Max. You can export the 3D world as a W3D file, and import it as a Shockwave3D cast member. You can then place the Shockwave 3D cast member on the Stage as a sprite.

To summarize:

- The Stage is the authoring area where the Director movie is assembled.
- The Score displays the arrangement of channels that organize, display, and controls the movie over time.
- The Cast window is where all cast members, including the 3D cast members, are stored. Cast members are the media in your movies, such as sounds, text, graphics, and 3D scenes.
- Sprites are instances of cast members that appear on the Stage with individual properties and attributes.

- A sprite of a 3D cast member displays a particular camera's view into the 3D world. The 3D cast member contains models, which are individual objects inside the 3D cast member.
- The Library Palette lets you select the behaviors you want to use.
- The Behavior inspector lets you create and modify behaviors.

You will use the Stage, Score, Sprite and Cast Member workflow to place a Shockwave3D cast member on the Stage. You can use the Library Palette and - Behavior Inspector to add and edit behaviors on the 3D sprite. For an introduction to 3D behaviors and the Behavior inspector, see "Learning more about 3D behaviors" on page 20. For a full discussion on the built-in 3D behaviors, see "3D behaviors" on page 75.

The 3D workflow

To control what happens inside the 3D sprite, you will need to use a different workflow. To control a 3D scene on a 2D screen you need to use scripts much more than for a 2D scene.

Director provides you with two windows where you can manipulate the view in the 3D sprite.

- "Using the Shockwave 3D window" on page 12
- "Using the Property Inspector for 3D" on page 14

These windows allow you to change settings that affect the 3D world as a whole. They also allow you to manipulate the position and rotation of the main camera. However, they do not allow you to adjust the position or properties of any models or any of the other nodes.

Using Lingo and JavaScript syntax

You can perform many basic 3D operations by using the built-in 3D behaviors in Director (see "3D behaviors" on page 75).

For all other manipulations of the 3D world, you will need to use your own custom scripts and commands. You will need to use either Lingo or JavaScript™ syntax, the built-in scripting languages of Director.

You can use the Message Window to set the properties of objects within the Shockwave3D cast member and to send commands to the 3D sprite. You can use the Object Inspector to display the values of any 3D property in real time.

The 3D documentation assumes that you understand either Lingo or JavaScript syntax. If you have not yet learned Lingo or JavaScript syntax, see the Scripting Reference topics in the Director Help Panel, which list all of the Lingo and JavaScript syntax methods and properties that are available in Director. The Scripting Reference topics in the Director Help Panel describe each expression, illustrate its syntax, and provide examples.

Lingo and JavaScript syntax are the Director scripting languages. They can be used to create movies that are more complex and interactive. For detailed Lingo and JavaScript syntax information, see "3D: Controlling appearance" on page 88 and "3D: Controlling action" on page 202.

In the Message and Script Editor windows, you can use the Alphabetical 3D Lingo and Categorized 3D Lingo contextual menus to find appropriate scripting terms.

Getting started with 3D Text

Director provides easy but powerful 3D text handling. If you are new to 3D or to writing scripts, 3D text is a good place to start learning. You will be able to create a simple 3D movie with no need to write any code of your own. For more information, see "3D text" on page 58.

Adjusting appearance through scripting

There are two main ingredients in a Shockwave3D world: appearance and action. A 3D graphic designer can define the appearance of a 3D world using third-party 3D design software. In many cases, you will not need to use scripting to set the appearance of the 3D world at all.

If you plan your project right, you can limit scripting to only those things that need to change in real time. This is the main topic of "3D: Controlling appearance" on page 88.

You will need to use scripting to:

- Change how models are grouped together ("Nodes" on page 88)
- Change the view shown by the camera ("Cameras" on page 101)
- Create static or dynamic overlays and backdrops ("Overlays and backdrops" on page 112)
- Change the lighting of a scene ("Directional lights" on page 124)
- Change the appearance of the surface of models "Shaders and appearance modifiers" on page 130, "Textures" on page 141, "Rendering" on page 154)
- Change the shape of a model ("Geometry" on page 159)
- Control the display of a particle system ("Particles" on page 197)

Programming movement and interactions

Without action, a 3D world becomes a static 2D image. Most of the code that you write controls the movement of objects and the interactions between objects. This is the main topic of "3D: Controlling action" on page 202.

Actions

You need to use scripting to control:

- How objects move relative to each other and relative to the world ("Motion" on page 260).
- The movement of the camera "Moving the camera" on page 216).
- How the user interacts with the 3D world using the keyboard and the mouse ("User interaction" on page 240).
- The illusion that the models in the 3D world are solid, and the collisions between them ("Collisions" on page 279)
- The illusion that the models are bound by the laws of physics ("Physics" on page 293).
- The events that occur when models interact with each other ("Events" on page 355).

Mathematics

Shockwave3D is designed to take care of as much of the low-level mathematics as possible, so that you do not need to worry about mathematics. However, to get the most out Shockwave3D, you need to understand:

- The 3D coordinate system
- How to define movements, positions, rotations and orientations
- How 3D space appears differently to different 3D objects

These topics are dealt with in "3D mathematics" on page 361.

Performance

Simulating a 3D world requires an enormous number of calculations, many times a second. You will want to get the most out of computers with low specifications, and to create the most spectacular 3D experience on high-end computers.

The less you ask the computer to do, the faster it can do it. In "Performance" on page 387 you will learn how to minimize your use of certain techniques in order to maximize the end-user's overall experience.

The 3D Xtra extension

The 3D capabilities of Shockwave3D are brought to Director by the 3D Xtra extension. The 3D Xtra extension depends on a number of other xtra extensions in order provide all the 3D features. For Shockwave3D to work correctly, the following xtra extensions must be included in a projector:

Macintosh:

- Shockwave 3D Asset Xtra.xtra
- SWA Decompression PPC Xtra.xtra
- InetUrl PPC Xtra.xtra
- NetFile PPC Xtra.xtra

Windows

- Shockwave 3D Asset.x32
- SWADCmpr.x32
- INetURL.x32
- NetFile.x32

If your movie uses simulated physics, then you will also need to include:

Macintosh:

- Dynamiks.xtra

Windows

- Dynamiks.x32

You will find all these xtra extensions in the Xtras folder, which is inside the Configuration folder placed alongside the Director application.

The 3D Xtra extension lets you include 3D models in a Director movie. You can import 3D models or worlds created with a 3D modeling program, and use Director to deliver them on the web. You can also combine the abilities of Director and your 3D modeling software by building a 3D world in your modeling program and adding to it or modifying it in Director.

To use 3D images and text created in a third-party rendering software, you must convert the file to W3D (Web 3D), DAE (Collada), and SKP (Sketchup), which Director supports. Typically, each rendering application requires its own specific file converter to create W3D files. For more information about creating W3D files, see the documentation for your 3D modeling software.

Panels for managing 3D content

When adding 3D features to a Director movie, you are likely to use the following panels:

- Stage
- Cast
- Score
- Script editor
- Object Inspector
- "Using the Shockwave 3D window" on page 12
- "Using the Property Inspector for 3D" on page 14
- "3D Behaviors in the Library Palette" on page 17
- "Learning more about 3D behaviors" on page 20

The Stage, Cast, Score, Script editor and Object Inspector panels function in exactly the same way for 3D content as for any other content. This section provides information on the panels that have 3D-specific features.

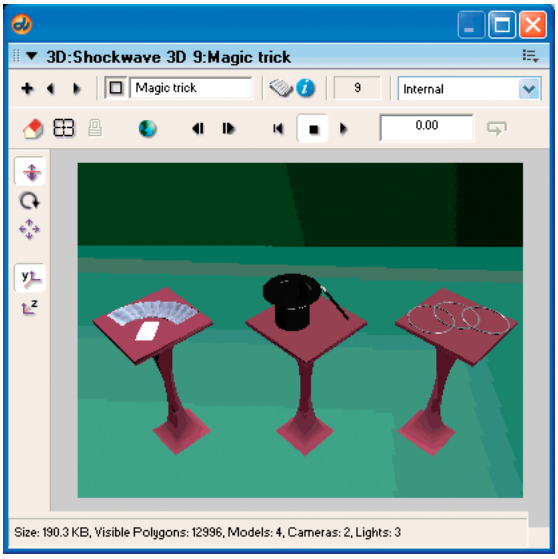
Using the Shockwave 3D window

The Shockwave 3D window provides an easy way for you to inspect a 3D cast member. Some properties of 3D cast members can also be edited in this window. However, you cannot manipulate any of the models directly.

1 Select a 3D cast member in the Cast window, in the Score, or on the Stage.

2 Click the Shockwave 3D Window button on the Director toolbar, double-click on the 3D cast member, or press the Return or Enter key.

The Shockwave 3D window appears, displaying the 3D cast member currently selected in the cast.



Shockwave 3D window

3 Use the following controls:

- The camera buttons along the side (Dolly, Rotate, and Pan) let you change your viewing angle by zooming in and out, moving around the world origin, and moving in a straight line horizontally or vertically, respectively. Hold the Shift key while using these tools to make the camera move faster.

Camera buttons

A. Dolly Camera B. Rotate Camera C. Pan Camera

- The two buttons below the camera buttons let you control whether the *y*-axis or the *z*-axis is the up axis when using the Camera Rotate tool.

Camera buttons

A. Camera Y Up B. Camera Z Up

- The playback buttons let you either play the cast member's animation at normal speed or step through the animation, forward or backward, by using mouse clicks to control the movement.
- The Loop button lets you play animations within the 3D cast member repeatedly.
- The Set Camera Transform and Reset Camera Transform buttons let you set and undo the changes you make to camera angles for the member's default camera. Reset Camera Transform restores the camera to the previously remembered position. Set Camera Transform remembers the current camera position. When you save your movie, the newly set camera position and orientation will be saved.

Camera buttons

A. Reset Camera Transform B. Set Camera Transform

- The Root Lock button fixes an animation in place, so that it doesn't change its position on the Stage while playing.
- The field at the top of the Shockwave 3D window shows the name of the cast member on display. The square button to the left of the text box lets you drag that cast member to the Stage or the Score.
- The New Cast Member, Previous Cast Member, and Next Cast Member buttons at the upper left of the Shockwave 3D window let you add or display 3D cast members.
- The Reset World button restores the 3D scene to its original state, with all models, cameras, and so on assuming their original positions. If you have used Set Camera Transform, the camera will revert to the last position that you set. If you need to reset your 3D world completely, use File > Revert. However this option deletes all the changes since you last saved.

  Alternatively, you can re-import the W3D file.

Using the Property Inspector for 3D

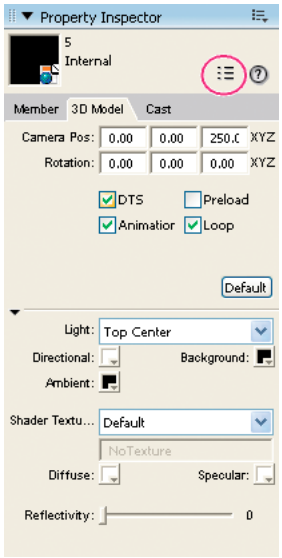
With the Property inspector, you can modify the 3D cast member without using scripts. The 3D Model tab of the Property Inspector offers a simple way to view and control numerous aspects of the 3D world.

1 Select a 3D cast member in the Cast window, on the Stage, or in the Score.

2 Click Property Inspector in the toolbar.

3 Click the 3D Model tab in the Property Inspector.

The Property Inspector appears in Graphical view. If the Property Inspector is in List view, click the List View Mode icon to toggle the view to Graphical.



Property Inspector at the 3D Model tab with List View Mode icon circled in red

The Property Inspector's 3D Model tab provides the following options:

- The text boxes at the top of the tab show the initial position and orientation of the default camera. The default (0, 0, 0) represents a vantage point looking up the *z*-axis through the middle of the screen. The values you enter in these text boxes replace the displayed values and move the camera.

  See "Cameras" on page 101 for more information on the camera view. See "Vectors" on page 361 and "Transforms" on page 370 for more information on how the position and orientation of the camera is defined.

- The Direct To Stage (DTS) option controls whether rendering occurs directly on the Stage (the default) or in the Director offscreen buffer. The offscreen graphics buffer is where Director calculates which sprites are partly hidden behind other sprites. When Direct To Stage is on, Director bypasses its offscreen buffer and saves time, increasing playback speed. However, when Direct To Stage is on, you cannot place other sprites on top of the 3D sprite.
- The Preload option controls how media that is being downloaded to the user's computer is displayed. The media can be held back from display until it has been completely streamed into memory, or it can be displayed progressively on the Stage as data becomes available.
- The Play Animation option controls whether any existing animation, either bones or keyframe, is played or ignored.
- The Loop option controls whether the animation loops continuously or plays once and stops.

  For more information on animations, see "Pre-defined animations" on page 270.

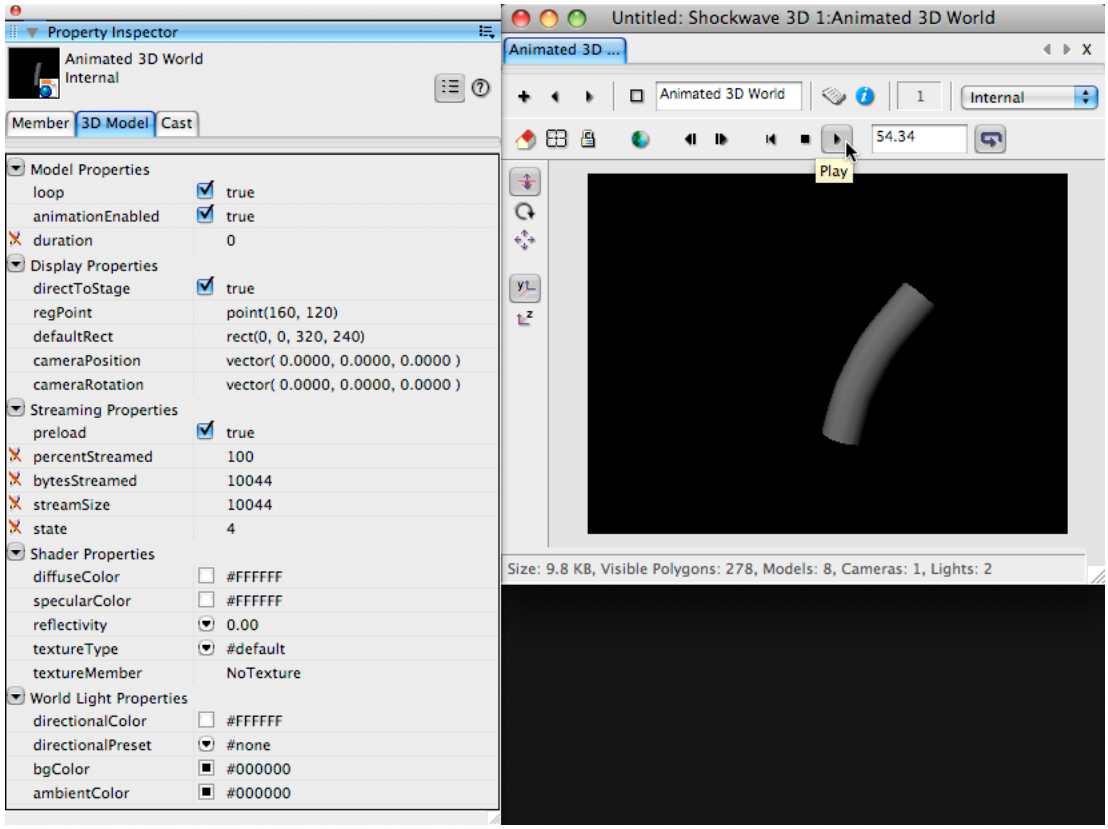
- The Director Light area lets you select one of ten lighting positions to apply to a single directional light. You can also adjust the color for the ambient light. (Directional light comes from a particular, recognizable direction; ambient light is diffuse light that illuminates the entire scene). Finally, you can adjust the background color of the scene.

  For more information see "Pre-defined animations" on page 270

- The Shader Texture area lets you work with shaders and textures. A *shader* determines the method used to render the surface of a *node*. A *texture* is an image that is applied to the shader and drawn on the surface of the model. All new models use the default shader until you apply a different shader. Using the Property inspector, you can assign a texture to the default shader. You can also control the default shader's specular (highlight) color, its diffuse (overall) color, and its reflectivity. For more information, see "Shaders and appearance modifiers" on page 130 and "Textures" on page 141.

Using the Property inspector for 3D: List View

When the Property Inspector is in List View, the 3D Model tab gives you access to the same information in a different format, along with some new items:



Property Inspector in List View showing the 3D Model tab

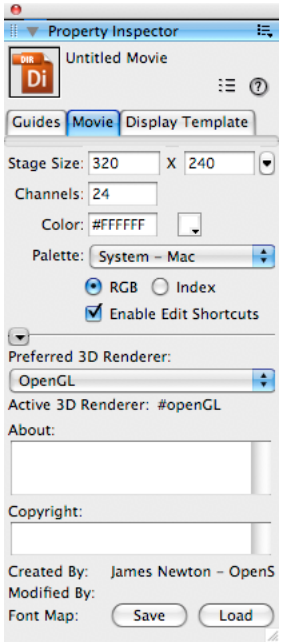
In List View, the Property inspector's 3D Model tab provides the following additional information:

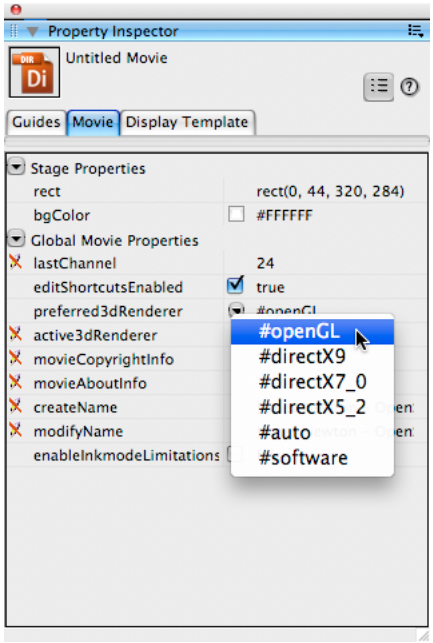
- Duration of the movie's animations (read-only).
- RegPoint of the 3D cast member. By default this is `point(160, 120)`, but you can set it to any point value.

- DefaultRect of the 3D cast member. By default, this is 320 x 240 pixels. This determines the initial size of the 3D sprite when you drag the cast member onto the Stage, or into the Score. You can change the rect of the sprite on the Stage without affecting the member's default rect.
- PercentStreamed, bytesStreamed, streamSize and state properties (all read-only). A 3D member that has been imported into the cast will show maximum values for all these properties. A linked external W3D file that is streaming from a remote server can show a series of different values as the W3D file is downloaded.

Using the Property inspector for 3D: Renderer

Information on the 3D renderer that a Director movie uses is displayed on the Movie tab of the Property Inspector.



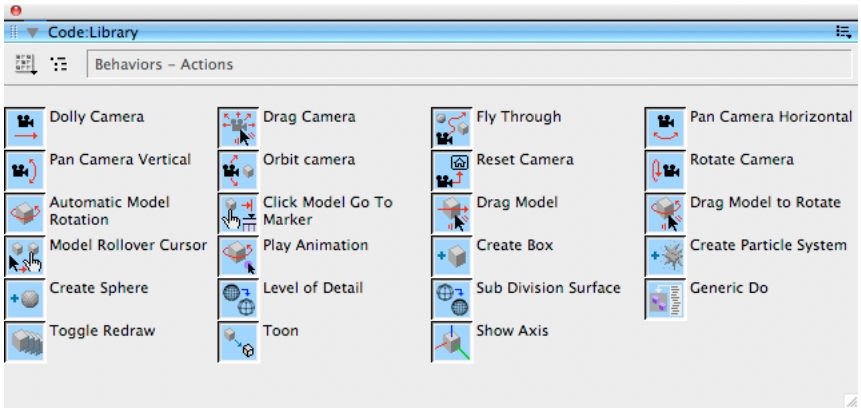


Two views of the 3D Renderer information in the Property Inspector

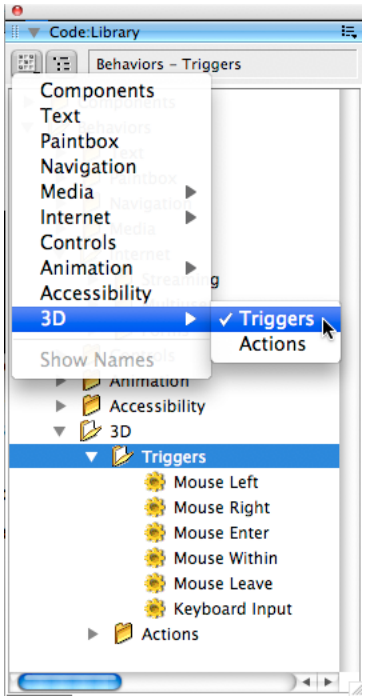
For more information on the different 3D renderers, see "Rendering" on page 154.

3D Behaviors in the Library Palette

The Director Library Palette includes 3D-specific behaviors. To open the Library Palette, choose the menu Window > Library Palette. You can find the 3D behaviors under 3D > Actions and 3D >Triggers.



The Library Palette in Thumbnail view showing 3D Action behaviors



The Library Palette in Tree view, showing how to access the 3D Trigger behaviors

3D behaviors are divided into four types: local, public, trigger, and independent.

Local Behaviors

For the local behaviors listed below, the user must click and drag the 3D sprite itself, or press a key while the 3D sprite has keyboard focus. One or more Trigger behaviors need to be added to the same 3D sprite in order to control the action.

- Fly Through
- Drag Camera
- Drag Model
- Drag Model to Rotate
- Click Model Go To Marker
- Play Animation
- Create Box
- Create Sphere
- Create Particle System

Public Behaviors

For the public behaviors listed below, you can place a Trigger behavior on any sprite. For example, for the Dolly Camera behavior, you can create two separate buttons on the Stage. To one of these buttons you can add a Trigger behavior to make the camera dolly in (move forward). To the other button, you can add a Trigger behavior to make the camera dolly out (move backwards). The user can now click these buttons in order to control the forward and backward movement of the camera.

- Dolly Camera
- Pan Camera Horizontal
- Pan Camera Vertical
- Rotate Camera
- Reset Camera
- Toggle Redraw
- Generic Do

Trigger Behaviors

Local and public behaviors must be paired with Trigger behaviors. When the user performs the appropriate mouse or keyboard action, the associated Action behavior on the 3D sprite will be triggered. For example, attaching the Create Box action and Mouse Left trigger behaviors to a 3D sprite causes a box to be created in the 3D world each time the sprite is clicked with the left mouse button.

- Mouse Left
- Mouse Right
- Mouse Enter
- Mouse Within
- Mouse Leave
- Keyboard

Independent Behaviors

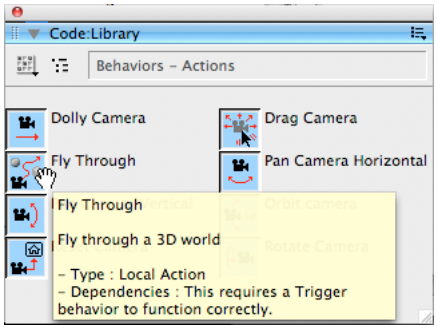
The behaviors listed below do not need a trigger. Their functionality is applied automatically. The Toon behavior, for example, changes a model's rendering style to the toon style.

- Automatic Model Rotation
- Orbit Camera
- Model Rollover Cursor
- Sub Division Surface
- Level of Detail
- Show Axis
- Toon

Learning more about 3D behaviors

Viewing tool tips for 3D behaviors

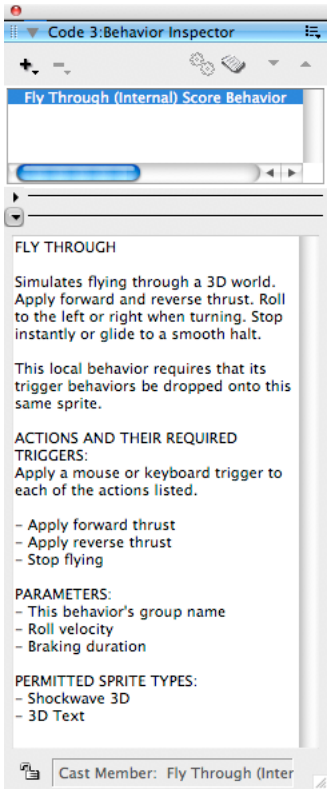
In the Library Palette, hold your mouse over the icon for any behavior. A tool tip appears to give a brief description of what the behavior does, and how to use it.



Hover the mouse over a behavior in the Library Palette to see a tool tip

Using the Behavior Inspector with 3D behaviors

You can drag a behavior from the Library Palette onto a sprite on the Stage or in the Score, or you can drag a behavior into the Cast window. You can now use the Behavior Inspector to see a more complete description of the behavior.



The Behavior Inspector showing the description of the Fly Through behavior

Select a behavior in the Cast Window, or select a sprite to which the behavior is attached and make sure the Score or the Stage is the front window. If you now open the Behavior Inspector you can read a full description of the behavior.

To open the Behavior Inspector:

- Double-click on any behavior (except if you changed the preferences for the default editor for behaviors, as described at Launching external editors).
- Right-click (Windows) or Control-click (Mac) on a sprite, and choose Behaviors from the contextual menu.
- Choose the menu item Window > Behavior Inspector.
- Use the associated keyboard shortcut: Ctrl-Alt-; (Windows), Command-Option-; (Mac).

The elements of a 3D world

In "The 3D world" on page 4, you can find a summary of the different elements that can be found inside a 3D cast member. This section explains the relationships between all these elements. You will learn how all the objects in a 3D cast member work together to create the scene in the 3D sprite. Each article in this section describes one type of 3D object in detail, and shows how it interacts with the other 3D objects.

Everything that you can see in a 3D sprite is a *model*. Every model has a position and an orientation in *3D* space. A model *resource* defines the shape of a model. A *shader* defines the appearance of the surface of a model. A *texture* is an image that appears in a layer of a shader; a texture appears like paint on the surface of the model. A *transform* defines the position and orientation of a model in 3D space. A *light* illuminates the 3D world. Some lights create an effect of light and shade on models. A *camera* shows the models in the 3D world from one particular viewpoint. A *physics object* gives a model the appearance of weight, solidity and flexibility.

This section shows you how these concepts work together. This section also includes many demonstrations of what Shockwave 3D can do.

- 3D space
- Model resources
- Models
- Groups
- Frames of reference
- Transforms
- Shaders
- Textures
- Lights
- Cameras
- Physics

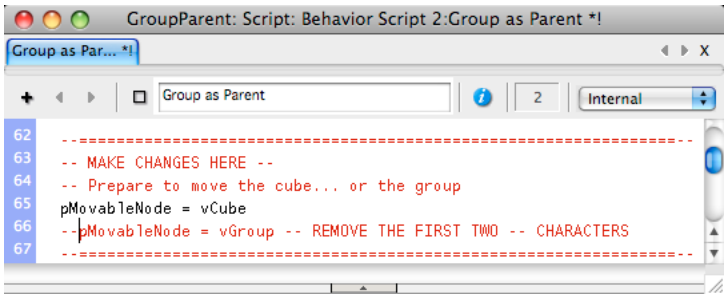
Concepts and Code

This section is designed to help you understand the concepts involved in creating an interactive 3D movie. The section also includes a number of movies that are designed to illustrate these concepts.

This section is not designed to teach you to write your own 3D scripts. You will learn to write 3D scripts in "3D: Controlling appearance" on page 88 and "3D: Controlling action" on page 202.

Do not worry if you do not understand all the code in all these example movies. A simple illustration sometimes requires complex code. First focus on the concepts. You can come back to these movies later to study the code in more detail.

Some of the movies contain code that you will need to change in order to understand a particular concept. Here's an example:



If you need to edit any code, the changes will be clearly labeled

For simplicity, all the code in this section will be in Lingo only. You will find links to articles in other sections where can find information on writing code in JavaScript syntax.

3D space

On Director 2D Stage, the position of every sprite is defined by its distance in *pixels* from the left side and from the top of the Stage. A sprite at the point (102, 34) is positioned 102 pixels from the left and 23 pixels from the top of the Stage.

In a 3D world, positions are measured in *world units* from a position at the center of the world. It is up to you to decide whether these world units represent microns, millimeters or miles. You need to measure in three different directions. These directions are named x, y and z.

- **x** - to the right
- **y** - upwards
- **z** - forwards

These directions are orthogonal, which means that they are all at right-angles to each other. Each of these directions is called an axis. The plural of axis is axes (pronounced ax-eez).

In Director, these axes are colored:

- **x-axis** - red
- **y-axis** - green
- **z-axis** - blue

To visualize these axes, download the movie 3dSystem.dir and launch it.

Open the Shockwave 3D window. To do this, double-click on the 3D sprite in the Score or on the Stage, or on the "3D World" member in the Cast window. You can now view the scene from different angles, and check that the three axes are indeed always at right-angles to each other.

Select the Rotate Camera tool, then drag the view in the preview window.

As you rotate the camera in the Shockwave 3D window, you can see two other sets of axes. The behavior on the 3D sprite executes a statement similar to the following:

```
member("3D World").debugFlags = 2
```

This statement shows the position and orientation of all the invisible objects (cameras, lights and groups) in the scene by showing the axes of each object. What you see in the initial view is the position and orientation of the group("World").

As you rotate the camera in the Shockwave 3D window, you can also see the axes for a directional light and for the camera that is used to create the view in the 3D sprite. (You can see the axes for the camera at the top right in the image above). An ambient light is also present. However, its axes coincide exactly with those of group ("World"), so you will not see a separate set of axes for the ambient light.

A handy mnemonic device

You can reproduce the directions of these three axes with your right hand.

Hold your right hand up in front of you at the same level as your eyes, with the palm towards you. Point your thumb out to the right. Point your index finger upwards. Point your middle finger towards your eyes.

Note: *Use your right hand even if you are left-handed.*

The three fingers indicate the directions of the 3D world's three axes:

x-axis - thumb to the right

y-axis - index finger upwards

z-axis - middle finger towards you

The right-hand rule

The origin of the world

Every 3D cast member contains a group called "World". This group cannot be deleted. By definition, this group is placed at the center of the 3D space. Its position is defined as `vector(0, 0, 0)`.

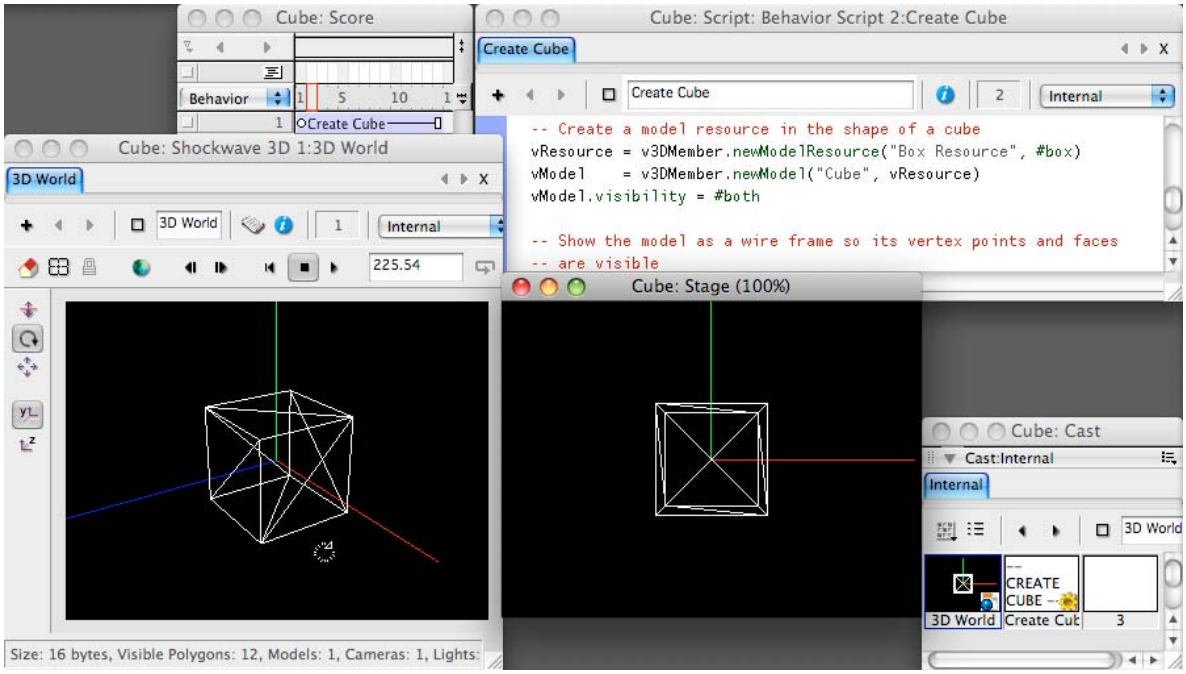
By default, all the models, lights, and objects in a 3D cast member are *children* of group("World"). (You will discover an interesting technique with nodes that are not children of the World in "Sky box" on page 108). The position of all these child nodes can be measured from the origin point at `vector(0, 0, 0)`.

Defining a shape in 3D space

A *model resource* defines a 3D shape in space. In the real world, a three-dimensional object is solid. On a computer, a simulated 3D object is made of *vertex points* that are joined together by lines to create triangular *faces*. These triangular faces are often called *polygons*. (Some 3D design applications allow you to create faces with more than 3 sides, but Director only allows you to use triangles.)

It is not possible to see a model resource. As far as your computer is concerned, a model resource is just a set of numbers. To see what those numbers represent, you need to create a model from the model resource.

To see an example, download the movie Cube.dir and launch it.



The movie Cube.dir creates a cube model and places it at the center of the 3D world

Open the Shockwave 3D window by double-clicking on the "3D World" member in the Cast window, and use the Rotate Camera tool to view the cube from all sides.

The code in the "Create Cube" behavior does 4 key things:

1 Creates a model resource using a *box primitive* (see "Regular primitives" on page 71).

2 Creates a model using the new model resource.

3 Makes both the inside and the outside faces of the model visible.

4 Changes the shader used by the model so that the vertex points and faces are visible.

Do not worry about how the code works yet. What you need to study is how the box shape is defined. When you finish reading this section, you will be able to understand the code.

To create a realistic box, you only need 8 vertex points and 12 polygons.

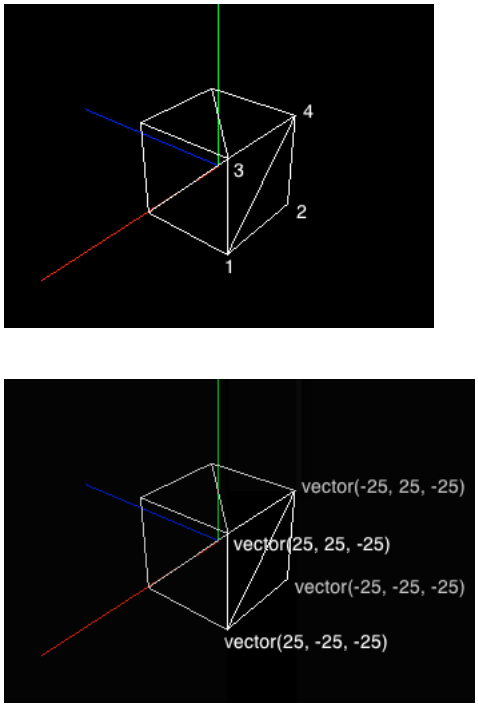
Note: *To create a realistic human figure, you need several hundred vertex points and faces. The more polygons a model resource has, the more calculations Director needs to make in order to display it. If your 3D world needs to react quickly to user input, it is a good idea to use as few polygons as possible.*

Vertex points

The model resource in the Cube.dir movie has 8 vertex points. The first one is at the position `vector(25, -25, -25)`. To reach that point, you:

- Start at the center of the world at `vector(0, 0, 0)`
- Move 25 world units in the same direction as the red x-axis
- Move 25 units downward (in the opposite direction of the green y-axis)
- Move 25 units backwards (in the opposite direction of the blue z-axis)

The pictures below show a simplified rendering of the cube shown from back. The first vertex is shown at the bottom of each picture.



The index numbers and the world positions of the first 4 vertex points in the Box Resource

Here's a complete list of vertex points for the cube:

```
[vector(25, -25, -25), vector(-25, -25, -25), vector(25, 25, -25), vector(-25, 25, -25),
vector( -25.0000, -25.0000, 25.0000 ), vector( 25.0000, -25.0000, 25.0000 ), vector( -25.0000,
25.0000, 25.0000 ), vector( 25.0000, 25.0000, 25.0000 )]
```

You'll notice that all vertex points are 25 world units away from the center of the world, in each direction. This means that the center of this cube is at the center of the world.

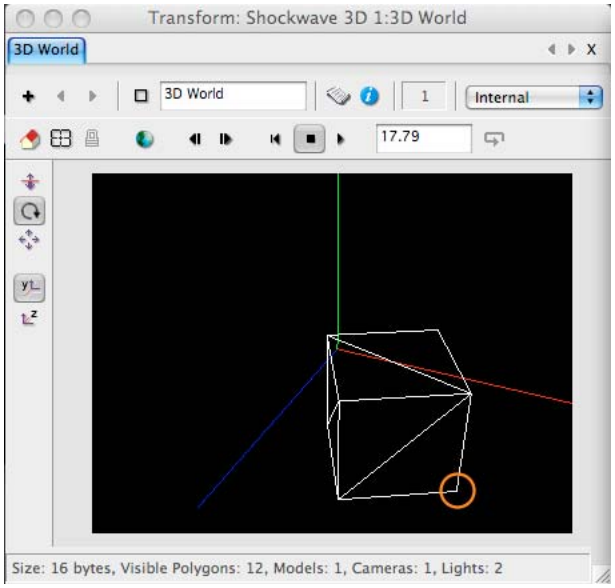
Note: *The 8 vertex points are not enough on their own to define a cube shape. The same 8 points could be linked together in different ways to create many different shapes. You will find more information about how the faces of a model resource are defined at "Creating a mesh resource" on page 170.*

World space and model space

In the example in "Defining a shape in 3D space" on page 23, the axes of the model resource and the axes of the world are aligned. The rear bottom right corner of the Box Resource was at the position `vector(25, -25, -25)` in the 3D world, and it was also at `vector(25, -25, -25)` in its own internal space. What happens if you move the model or rotate it?

Download the movie Transform.dir and launch it. This movie shows the same cube as in the movie Cube.dir with two differences:

- The inside of the box is not visible. (This is a cosmetic difference to simplify the view).
- The cube has moved and rotated.



The position of vertex 1 after the cube has been moved and rotated

As before, use the Rotate Camera tool to look at the cube from all sides.

In this movie, the cube can be imagined with respect to two different *frames of reference*: model space and world space. The first vertex in the model resource's vertexList is still at `vector(25, -25, -25)`, when it is seen from the model's point of view. Because of the altered position of the model in the 3D world, the same point is now at a different point when it is seen in world space. (See the orange circle). The output in the Message window indicates the position of the vertex in world space: `vector(59.1506, -50.0000, -9.1506).`

Using the debug property of a model

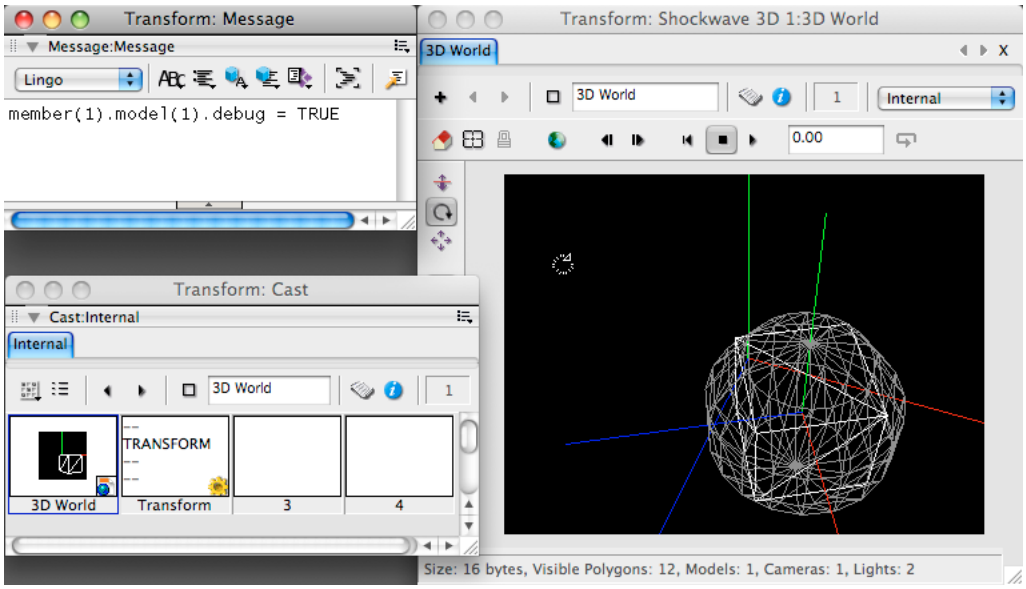
Type the statement below into the Message window.

```
member("3D World").model("Cube").debug = TRUE
```

When you do so, Director displays:

- The x-, y- and z-axes for the model
- The model's *boundingSphere*. This is the smallest sphere that can contain all the vertex points in the model.

The image below shows the result. Note that the statement shown in the Message window has exactly the same effect as the statement quoted above.



Using the debug property of the Cube model to visualize its origin, axes and boundingSphere

You can now see two sets of red, green, and blue axes: one for the world itself, and one for the model. This illustrates the different frames of reference for the world and for the model.

Frames of reference

In the real world, we are able to handle multiple frames of reference without being aware of them. When you reach for your mouse to click on a button on the screen, you make a complex movement in 3D space with your arm that results in the mouse moving in horizontal 2D space on your desk. This in turn causes the mouse pointer to move in an almost vertical 2D space on your monitor. You can move the mouse pointer without being conscious of all the muscles that you use.

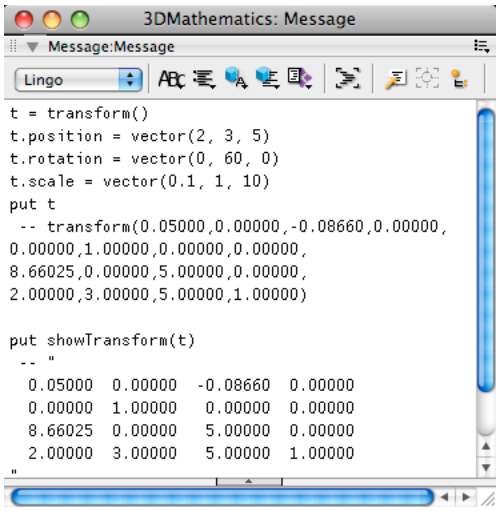
When simulating movement in a virtual 3D world, you need to pay close attention to the frame of reference of each object. Simulating the movement of a virtual hand means controlling the movement of a virtual upper arm, a virtual forearm and a virtual hand. Each of these objects moves in a different frame of reference, controlled by a shoulder, an elbow, and a wrist.

You can learn more about this in "Transforms, translation, rotation, and scale" on page 27 and "Using a parent to change the frame of reference of a 3D object" on page 30.

Transforms, translation, rotation, and scale

How does Shockwave 3D control movements in three dimensions? This article treats the ideas illustrated in "World space and model space" on page 25 in more detail.

Download the movie Transform.dir and launch it. This movie creates a 3D cube at the center of the world, and then moves it to a new position. It prints out information about the movement in the Message window.



The Message window displays the transform of the cube and the position of vertex 1, both before and after the movement

Transforms

The position of the Cube model is described by its *transform*. You can see the numbers that make up the model's transform in the picture above. In the Message window, there are two sets of numbers. The first set of numbers (#before transform) describes the cube at the center of the world. The second set of numbers (#after transform) describes the cube's new position and rotation.

Director lets you manipulate transforms in many ways without using any complicated mathematics. Much of the time, you will not even realize how much number-crunching Director is doing for you.

The position and axes of a transform

Here is the transform of the cube model after it is moved, arranged in a table with explanatory headers:

	world x-axis	**world y-axis**	**world z-axis**	**direction (0) or position (1)?**
transform x-axis	`0.50000`	`0.00000`	`0.86603`	`0`
transform y-axis	`0.00000`	`1.00000`	`0.00000`	`0`
transform z-axis	`-0.86603`	`0.00000`	`0.50000`	`0`
position	`25.00000`	`-25.00000`	`0.00000`	`1`

The first three rows in the transform indicate the direction of each axis of the model in world space. The first line indicates that the x-axis of the model points a little to the right, neither up nor down, and quite a bit forward. The second line shows that the y-axis of the model points in the same direction as the y-axis of the world. The third line shows the direction of the z-axis of the model in world co-ordinates. You can compare these figures with the directions of the colored axes that you see when you set the `debug` property of the Cube model to `TRUE`.

The last four numbers indicate the position of the model. The model has been moved 25 units along the world's x-axis and 25 units downwards, in the opposite direction to the world's y-axis. The final 1 in the transform indicates that this information defines a position in space.

As an exercise, try to create the same table for the #before transform. Do you see how each axis of the model corresponds exactly to the same axis in the world?

Applying a transform to a position

A transform is a frame of reference in mathematical terms. When a transform is applied to a vector position, it moves that vector position to a new point in space. The lines highlighted in yellow in the Script window and the Message window show what happens when a neutral transform is applied to the position of vertex 1: it stays the same. The lines highlighted in orange in the Script window and the Message window show what happens when the modified transform is applied to the position of vertex 1. In this case, the vertex moves to the new position that is shown by the orange circle.

For more mathematical information concerning transforms, see "Transforms" on page 370.

Translation

To move a model from one position to another, you can use the node.translate() command. In the simplest terms, the `translate()` command tells the model how far to move to the right, how far to move up, and how far to move forward.

After a `translate()` command has been executed, the model will still be facing in the same direction. When you think about a `translate()` command, the order of the three separate movements is not important. Imagine that you move a model 3 units to the left, 4 units down and 7 forward. The model would arrive in exactly the same spot as if you started from the original position and moved it 7 units forward, 3 units left, and then 4 units down.

Rotation

To rotate a model, you can use the rotate() command.

In a 3D world, you can rotate around three different axes: x, y and z. The `rotate()` command expects a value in degrees for each axis. For example, this statement will rotate the Jug model 45°around the x-axis, -90° around the y-axis, and 0° around the z-axis:

```
member("Dinner Party").model("Jug").rotate(45, -90, 0)
```

The order in which the rotations are performed is important.

Imagine that you are sitting at a table. You have a cup in front of you, and you are holding a jug of water to the right of the glass. Imagine that the handle of the jug is to your right. You take hold of the jug and tilt it forward. The water will pour out of its spout into the cup. Imagine that you keep the jug tilted rotate it around its vertical axis. The water will continue to pour into the cup, but from the side of the jug. All is well (if a little unconventional).

Now imagine that you performed those rotations in a different order. Imagine that you start by rotating the jug around its vertical axis. The spout is now pointing away from you, and not at the cup. If you now tilt the jug, the water will spill on the table. This is probably not good.

Direction of rotation

In the Cube example above, the cube model was rotated through -60° around the y-axis. How do you know if that is a clockwise or a counter-clockwise rotation?

The second "right-hand rule" helps here. Stick out the thumb of your right hand and curl the fingers. Imagine that your thumb is pointing in the direction of the axis of rotation. Now turn your hand in the direction that your fingers are pointing. That is the direction of a positive rotation.

The right-hand rule for rotation

Scaling a model

When you create the models for a project, it makes sense to create them all to the same scale. Sometimes, however, you may want to use models that were originally created for different projects at different scales. Alternatively, you may want to make an object shrink or grow. One example is a 3D progress bar that you want to stretch along one dimension but not along the others.

You can use the scale() command to change the scale of a model. The movie Transform.dir does not illustrate a change of scale. You can find an example setting the scale of a transform in "Using a parent to change the frame of reference of a 3D object" on page 30.

Order of execution

You can perform the following actions on a model (and by definition, on its transform):

- `translate()`
- `rotate()`
- `scale()`

The order in which you perform the operations is important. If you walk forward three paces and then turn left, you will be in a different place than if you turn left and then walk forward three paces. If you scale up to 10 times your original size before you start take three paces, you will go much further than if you take three normal paces, and then scale up to 10 times your original size.

Frame of reference for a movement

When considering a movement, you must also consider the frame of reference.

Imagine a model named "Car" in a 3D cast member named "Street Scene". Imagine that the Car model has been rotated through 90° with respect to the world, so that the Car's z-axis is parallel with the world's x-axis. Imagine that the z-axis of the car model points from the rear of the car towards its front. This statement will make the car move 30 units in the direction of its own z-axis.

```
member("Street Scene").model("Car").translate(0, 0, 30, #self)
```

This might make the car seem to drive along the street. The next statement, however, will move the car in the direction of the world's z-axis:

```
member("Street Scene").model("Car").translate(0, 0, 30, #world)
```

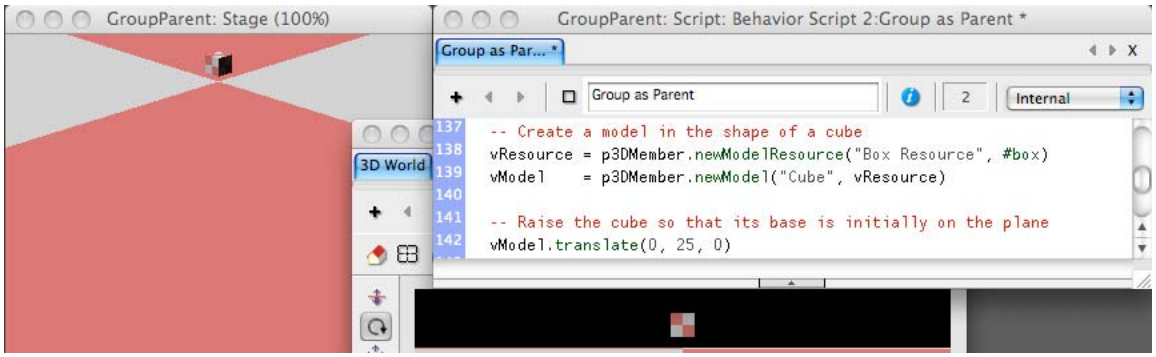
This is perpendicular to the direction in which the car is pointing. The car will move sideways. It is a very useful (but unrealistic) way to park the car in very tight parking spot.

Using a parent to change the frame of reference of a 3D object

Every model has an origin point. This corresponds to the point `vector(0, 0, 0)` in the model's resource space. When you translate, rotate or scale a model, the change is made with respect to the origin point.

Sometimes you may want to rotate or scale a model around a different point. To do this, you can make the model the child of a different node, and then apply your changes to the parent node.

To see an example of this, download the movie GroupParent.dir and launch it.



The GroupParent.dir movie creates a box model, raises it by 25 units, then scales the model around its origin point

The code in the GroupParent.dir movie is quite complex. You do not need to understand all the code at this time. In simple terms, the following is what the movie does when it starts:

- Create a horizontal plane at the height where y = 0
- Create a cubic box model whose sides are 50 units long
- Raise the Cube model by 25 units, so that its base is on the horizontal plane where y = 0
- Change the scale of the Cube

Note: *This movie uses the default shader with the default red and white texture for both the plane and the Cube model. The previous movies in this section used an altered version of the default shader to create a luminous wire frame. You can learn more about shaders in* *"Using a shader to change the appearance of a model" on page 34.*

Scaling

When the Cube is first created, the value of the `scale` property of its transform is `vector(1.0, 1.0, 1.0)`. This means that 1 unit in the model resource space is equivalent to 1 unit in world space, along all axes.

The scale of the cube is then reduced to `vector(0.3, 0.3, 0.3)`. This means that the sides of the model are now only 15 units long. The center of the model is 25 units above the plane, so there will be a gap between the bottom of the cube and the plane.

Note: *You can use the three values of the scale vector to change the scale in different proportions along the different axes. Imagine a box model resource that is 50 units wide, 50 units high, and 50 units long. Imagine a model created from this resource, and that the model's transform has a scale value of* `vector(1.0, 2.0, 0.5)`*. This means that the model will be 50 units wide, 100 units high, and 25 units long.*

Moving the Cube

If you click on the plane in the GroupParent.dir movie, the origin of the Cube model will move to the point where you clicked. The cube model is now cut in half by the plane model. Its position relative to the plane changed.

Imagine that you want the Cube to sit on the surface of the plane at all times. In other words, you want it to move and scale around a point in the center of its base. You can do this by creating a group object and setting it as the parent of the Cube.

Setting the parent of a model to a custom group

In the `beginSprite()` handler, after the cube is created and placed at the world position `vector(0, 25, 0)`, the following two lines of code are executed:

```
-- Create a group to act as the parent of the cube
vGroup = p3DMember.newGroup("Cube Base")
vGroup.addChild(vCube, #preserveWorld)
```

By default, the new group will be created at the center of the world, at `vector(0, 0, 0)`. This vector now corresponds to the center of the base of the Cube. Instead of moving and scaling the Cube, you can now move and scale the group("Cube Base"). This requires a small change in the code of the Group as Parent behavior.

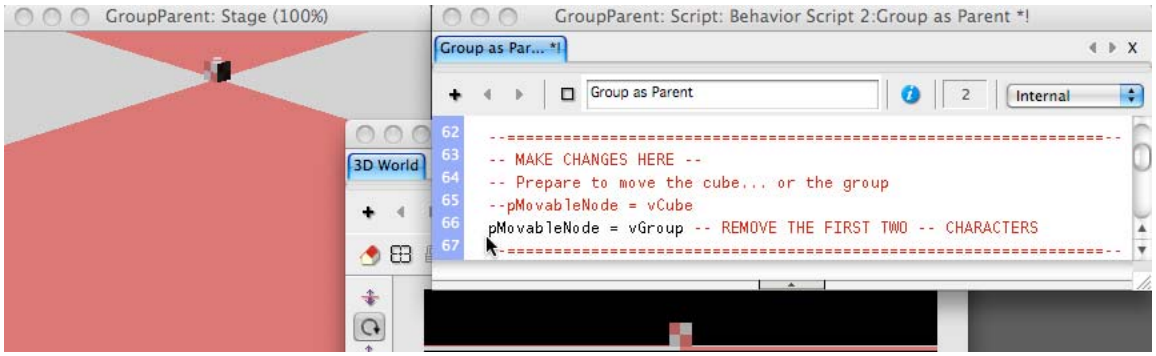
Find these lines:

```
--------------------------------------------------------------
-- MAKE CHANGES HERE --
-- Prepare to move the cube... or the group
pMovableNode = vCube
--pMovableNode = vGroup -- REMOVE THE FIRST TWO -- CHARACTERS
--------------------------------------------------------------
```

Change these lines to:

```
--------------------------------------------------------------
-- MAKE CHANGES HERE --
-- Prepare to move the cube... or the group
--pMovableNode = vCube
pMovableNode = vGroup -- REMOVE THE FIRST TWO -- CHARACTERS
--------------------------------------------------------------
```

Now restart the movie and watch the cube as its scale changes. Click on the plane model, and drag the mouse pointer around.



If you move and scale the Cube Base group, the Cube will remain sitting on the plane

This technique consists of three steps:

1. Creating a group.
2. Placing the group at the position that you want to use as the new origin point for the model.
3. Making the model a child of the group.

In this movie, step 2 was achieved by moving the model rather than by moving the group. See addChild() for more information on the options for step 3.

Note: *For more information on detecting the model under the mouse pointer, see "User interaction" on page 240.*

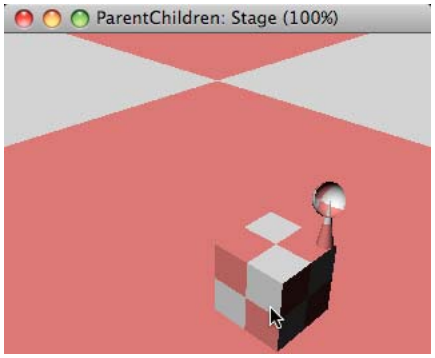
Using a parent to group several objects together

You can use any node as the parent of any other node. Cameras, lights, models and groups can all have child objects.

A child node can not have more than one parent. A parent node can have any number of child nodes. Any change made to the transform of the parent node will affect all its children.

A parent node can have children that have children of their own.

To see an example of this, download the movie ParentChildren.dir and launch it.



The ParentChildren.dir movie creates a contraption out of three models that move around as a group

The Multiple Children behavior creates the following models:

- A horizontal plane
- A cube
- A cone
- A hemisphere
- A group

The node hierarchy is set up as follows:

```
group("World")          - camera("DefaultView")
                        - light("UIAmbient")
                        - light("UIDirectional")
                        - group("Cube Base")      - model("Cube")
                                                  - model("Cone")        - model("Hemisphere")
```

Drag the mouse pointer around on the surface of the plane. You can see all the children of the group(“Cube Base”) follow the pointer around. The Hemisphere rotates around its parent Cone while the Cone follows its parent group(“Cube Base”).

Hierarchy

Each 3D cast member contains a group object called *world*, which may contain a tree-like parent-child hierarchy of nodes, such as models, groups, lights, and cameras. Each node may have one parent and any number of children. Nodes that have world as an ancestor are rendered. A cast member may also contain nodes that do not have world as an ancestor, such as nodes with a parent property set to VOID. Nodes such as these are not rendered.

The primary benefit of these parent-child relationships is that they make it easier to move complex models around in the 3D world and to have the component parts of those models move together in the proper way. In the example of a car, if the wheels of the car are defined as children of the car model, then moving the car will cause the wheels to be moved with the car in the expected manner. If no parent-child relationship is defined between the car and the wheels, moving only the car will cause the wheels to be left behind in their original position in the world.

Using a shader to change the appearance of a model

In order to be visible, a 3D model needs to have a shader. A shader provides instructions to the Shockwave 3D rendering engine on how represent the surface of the model.

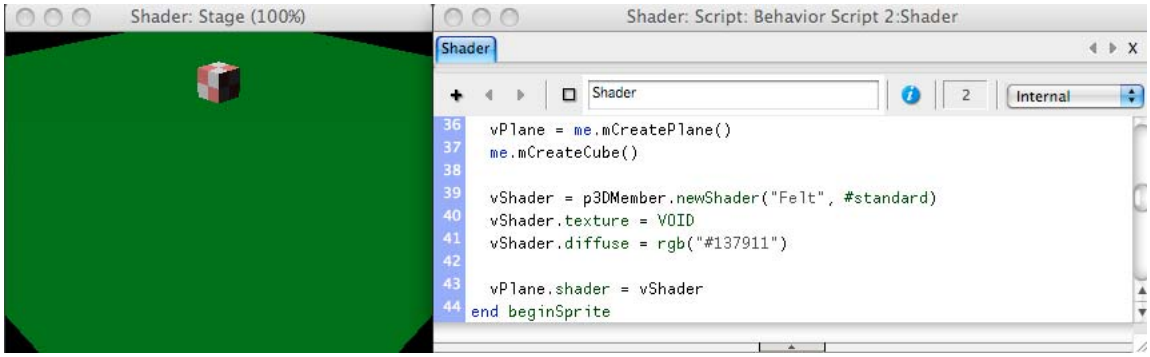
A shader is not part of a model. It is a separate object. Many models can use the same shader. Some models can use more than one shader.

When a 3D designer creates a virtual world in a third-party 3D design application, the designer is likely to apply attractive shaders and textures to the models before exporting the world as a W3D file. If you are happy with the appearance of the models in the imported file, you may not need to change the shaders and their textures at all.

All models created in Shockwave 3D by the `newModel()` function start with the built-in shader: `shader("DefaultShader")`. This has a characteristic red-and-white checker pattern.

If you change a shader, the appearance of all the models that use that shader will change. To change the shader for one model only, you need to create a new shader and apply it to the model.

To see this in action, download the movie Shader.dir and launch it.



The Shader.dir movie creates a new shader, the color of green felt, and applies it to the horizontal plane

The following Lingo code creates a new shader in the cast member named “3D World”. It removes the default checkered texture, sets the diffuse color of the shader, and applies the shader to model named “Plane”:

```
vMember = member("3D World")
vShader = vMember.newShader("Felt", #standard)
vShader.texture = VOID -- removes the checkered default texture
vShader.diffuse = rgb("#147911")
vMember.model("Plane").shader = vShader
```

Shader types

When you create a new shader, you must indicate both a unique name for the shader and a type. The following are the different types of shaders:

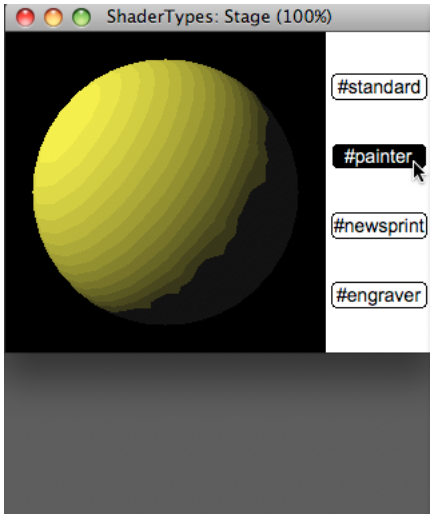
- #standard
- #painter
- #newsprint
- #engraver

Standard shader

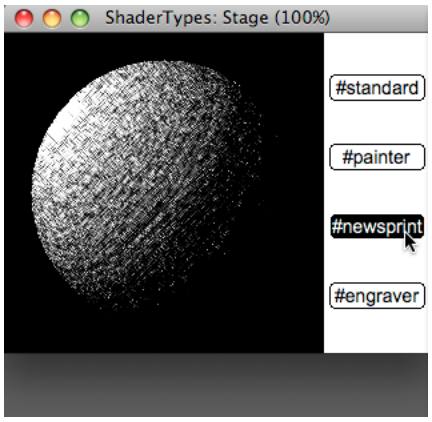
The `#standard` shader is the most versatile and the most realistic. Only the `#standard` shader supports the use of texture images. The next 3 articles will help you understand the possibilities of the `#standard` shader.

Painter, newsprint and engraver shaders

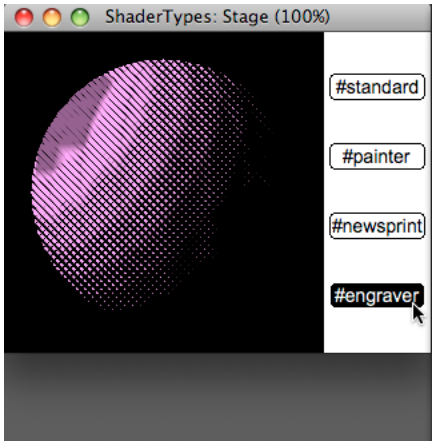
The other three shader types create a variety of artificial effects, based on light, shade and diffuse color. You can explore these using the movie ShaderTypes.dir.



The `#painter` shader allows you to create scenes that resemble cartoon strips.



The `#newsprint` shader allows you to create monochrome images similar to photos in a newspaper.



The `#engraver` shader gives a similar monochrome image with old-fashioned cross-hatchings.

Interaction with lights

Shaders of the type `#standard` react to all the lights in the 3D world in a complex way. This is discussed in "Lights" on page 40.

The other shader types react to lights in a limited way. They will react to the first non-ambient light according to this order of priority:

1. The `#directional` light that was added to the world the **longest time ago**, regardless of its position, or its intensity.
2. The `#point` or `#spot` light that shines the **brightest** on any face of the model, if there are no `#directional` lights in the scene. If the brightest light is a `#spot` light, its `#spotAngle` and `#rotation` properties of the `#spot` light are ignored.
3. An `#ambient` light, if that is the only type available.

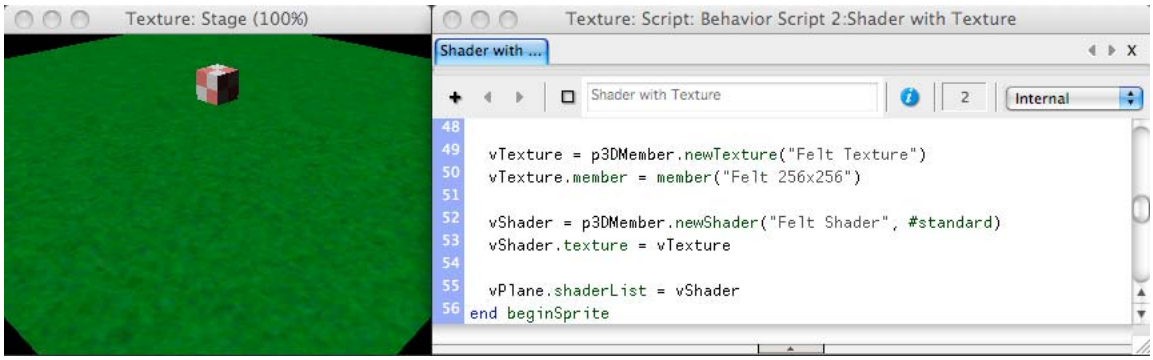
Shaders of type `#painter` also react to the color of ambient lights.

Shaders of type `#newsprint` and `#engraver` do not react to ambient lights at all, unless there is no other type of light available. They react to the **direction** but not to the color of the light with the highest priority, regardless of its type. You can change the color of the highlight area on a `#newsprint` or `#engraver` shader by setting its `#diffuse` color property (see the image of the `#engraver` shader above.)

Using a texture to place an image on the surface of a model

Each shader can have textures applied to it. Textures are 2D images drawn on the surface of a model. The appearance of the model's surface is the combined effect of the shader and textures applied to it. If you do not specify a texture, a default red-and-white bitmap is used.

To see how to create a new texture object and apply it to a shader, download the movie Texture.dir and launch it.



You can create a texture from an image object or a member with a .image property

This example creates a new texture in the cast member "3D World" to display the image of the bitmap member "Felt 256x256":

```
v3Dworld = member("3D World")
vName = "Felt Texture"
vBitmap = member("Felt 256x256")
vTexture = v3Dworld.newTexture(vName, #fromCastMember, vBitmap)
```

Alternatively, you can create a texture without associating an image to it. You can set the image or member of a texture at any time:

```
vTexture = v3Dworld.newTexture(vName)
vTexture.member = vBitmap
```

Displaying a texture

Textures can be displayed in three different ways:

- As a 2D image projected by the camera, as an overlay or a backdrop
- One each of the particles emitted by a particle system
- As part of a shader object that is attached to a model

For more information on overlays and backdrops, see "Overlays and backdrops" on page 47.

For more information on particle systems, see "Particle emitters" on page 74.

A shader contains 8 layers; a texture can be applied to each of the layers. You can adjust the settings of the different layers to create different effects, such as surface texture, reaction to diffuse light, reflections that appear to come from the environment, variations in the shininess of the surface and highlights. For details on how textures interact with the various shader layers, see "Standard shaders" on page 131.

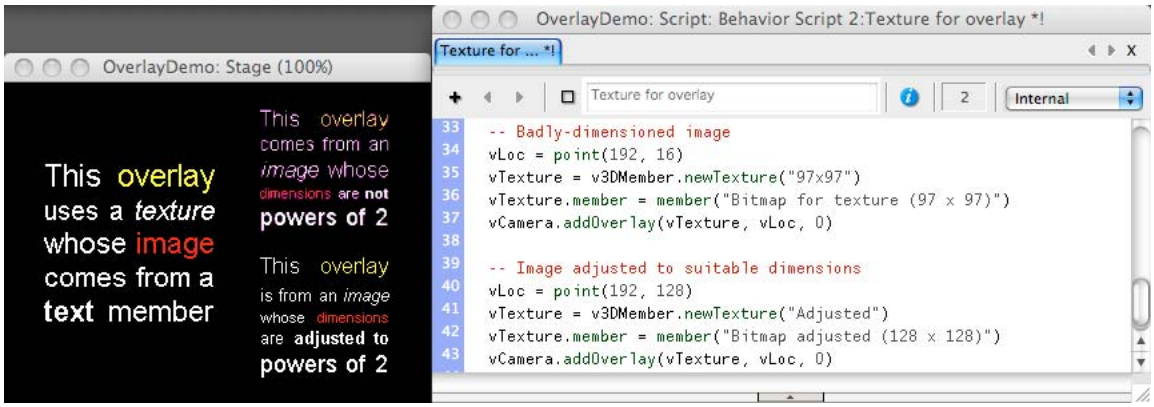
The following example sets the texture used by the first layer of the shader “Felt Shader” of the cast member “3D World”:

```
v3DMember = member("3D World")
vShader = v3DMember.shader("Felt Shader")
vTexture = v3DMember.texture("Felt Texture")
vShader.texture = vTexture
```

Dimensions of texture images

The pixel height and width of the 2D images that you use as textures must be powers of 2 (that is, 2, 4, 8, 16, 32, and so on). This is because most video cards scale images to powers of 2. If the image used does not have pixel dimensions that are a power of 2 (values including 2, 4, 8, 16, and so forth), both rendering performance and visual quality will decrease.

To see a demonstration of this, download the movie OverlayDemo.dir and launch it.



An image whose dimensions are not powers of 2 results in a poor quality texture

For a badly-sized image, most video cards will automatically redimension the texture so that its height and width are set to powers of 2. This may reduce the size of the texture (resulting in lower resolution), or it may increase the size of the texture (resulting in stretching and wasted RAM usage).

Computer memory requirements

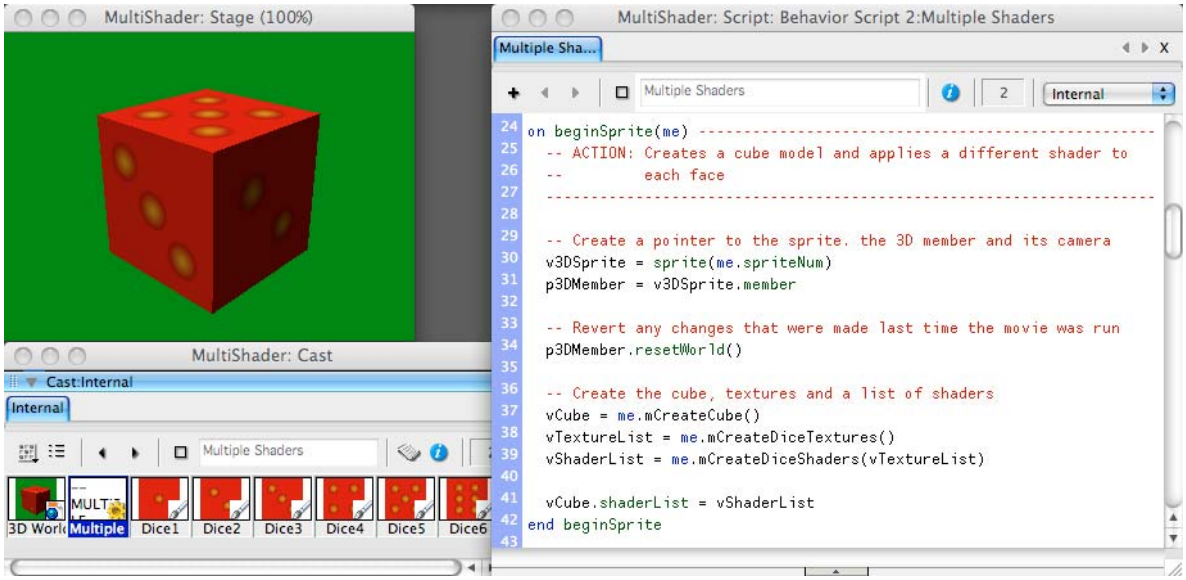
All the textures used in a 3D scene must be able to fit in the computer’s video RAM at the same time. If not, Director switches to software rendering, which slows performance.

Be aware of the limitations of your video RAM and that of your intended audience. Some video cards have as little as 4 megabytes of video RAM. Carefully budget your total texture size when designing your 3D world.

Resources, meshes, and shaders

A model resource is made of one or more meshes. A *mesh* is one section of a model resource. A model created from a model resource possesses as many shaders as there are meshes in the resource. These shaders are stored in the model's *shaderList*.

To see an example, download the movie MultiShader.dir and launch it.



A Box primitive has 6 different meshes. You can apply a different shader to each face of a Box model.

The MultiShader.dir movie contains 6 bitmap cast members named “Dice1” to “Dice6”. Each of these bitmaps has a height and width of 256 pixels (a power of 2). The movie creates six textures, one for each bitmap. It then creates six shaders, one for each texture.

You can change all the shaders for a model with one statement. In the MultiShader.dir movie, the shaders on all 6 faces of the Cube model are replaced by the following line:

```
vCube.shaderList = vShaderList
```

Note: *In the MultiShader.dir movie, the number of pips on a given face of the cube indicates the index number of that mesh. This allows you to work out the order in which the meshes of a* `#box` *primitive are created: back, right, front, left, top then bottom. This is not the order in which the pips appear on a real dice. Download and launch the movie Physics.dir to see a realistic simulation of a dice.*

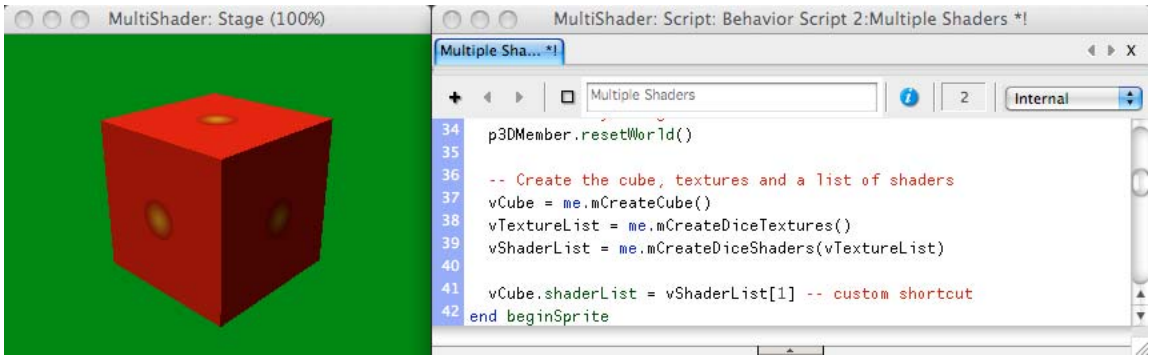
Custom handling of the shaderList

In Physics.dir, the resource for the Cube model has six meshes, so the model has a shaderList with six entries. If try to set the model's `shaderList` to a list containing fewer than six shaders, the built-in `shader("DefaultShader")` is used to fill in the gaps. If you provide a list that contains more than six shaders, the extra shaders are ignored.

You can give all the meshes in a model the same shader by using a custom shortcut. Instead of providing a list for a model's `shaderList`, you can provide a single shader. You can try this in the MultiShader.dir movie. Try changing the second last line of the `beginSprite()` handler to this:

```
vCube.shaderList = vShaderList[1]
```

Now, run the movie again. You see that all the faces show a single pip. The shader(“Dice1”) is used for all faces.



Set a model's shaderList property to a single shader to copy that shader into each slot in the shaderList

Advantage of multiple shaders

Imagine that you want a model of a computer monitor. You can create this model with two separate meshes: one for the plastic body of the monitor, and one for the screen. This approach allows you to change the shader on the screen mesh without changing the shader for the body. You can refresh the screen of your monitor efficiently.

When designing a 3D model for a human character, you can use a separate mesh for the skin, the hair, and for each of the characters clothes. This makes it easy to use the character as a customizable avatar.

Model resources and texture coordinates

An associated technique allows you to map specific points inside a texture to specific vertices of model resource. Your third-party 3D design software will normally do this for you before you export your 3D world as a W3D file.

For more details on how to do this from within Director, see "Mapping a texture to a mesh resource" on page 150.

Lights

The appearance of a model depends on both the shaders that are attached to it and the lights that are shining on it. The *shader* defines what the surface of the model looks like. The *lights* in a 3D world create an effect of light and shade on the model. In a static scene on a 2D screen, the variations in light and shade on a model give your brain clues about the 3D shape of the model.

In the real world, photons of light travel from a light source, and collide with real world objects. Depending on the wavelength of the photon and the material it strikes, the photon may bounce off and change color. If you can read this, there are countless photons around you, all moving at the same time.

When a team is making a high-budget computer-animated film, they can afford to simulate the movement of a large number of photons. Each image of the film may be show for less than 42 milliseconds, but the company can afford to have multiple state-of-the-art micro-processors working for several minutes to create each image.

In a movie created for Shockwave 3D, you need to rely on the micro-processor in your end-user's computer doing all it can in less than 42 milliseconds. Shockwave 3D has to take shortcuts. The more realistic you want your scene to look, the less efficient the shortcuts become.

Virtual light and virtual light sources

Shockwave 3D simulates light in three different ways: ambient light, diffuse light, and specular light. Shockwave 3D provides you with four different types of light source: ambient lights, directional lights, point lights, and spot lights. It is important to distinguish between *light* (the concept) and *lights* (the sources of light).

Simulated light

Shockwave3D uses a combination of three different techniques for simulating light in a virtual 3D world: ambient light, diffuse light, and specular light. All three are clever fictions.

Ambient light

It is night and it is dark. You switch on the ceiling light in your room. The room seems to fill with light. Photons bounce back towards your eyes from every surface in the room. The brighter your ceiling light, the brighter the room appears.

In Shockwave 3D, ambient light is an extreme simplification of the effect of all these photons bouncing around your room. The concept of ambient virtual light is that every surface in your room reflects the same amount of light, regardless of the material it's made of or it orientation to the light source. This is blatantly false, but it takes very little time for a computer to calculate.

Diffuse and specular light

Imagine a glass mirror with a heavy curtain in front of it. Light that falls on the mirror is reflected. Photons bounce off the mirror like billiard balls bouncing off the edge of a billiard table.

In a virtual world, light that is reflected at a precise angle in this way is called *specular* light. The more specular light a model reflects the shinier it will appear.

Light that falls on the curtain is dissipated in all directions. Photons that arrived at the same angle move off in all directions, like matches that scatter when you drop a box of matches.

In a virtual world, light that is scattered in this way is called diffuse light.

Types of light and computing time

The table below shows what calculations the video card needs to make for every surface for each type of light.

	light color	surface color	angle of incidence	angle of reflection
Ambient light	Yes	Yes		
Diffuse light	Yes	Yes	Yes	
Specular light	Yes	Yes	Yes	Yes

Because it requires the fewest calculations, ambient light is the most efficient. However, if your scene only has ambient light, everything will look flat.

Diffuse light from directional sources creates highlights and shades. It gives your brain clues about the shape of three-dimensional objects.

Specular light is the most expensive, in terms of processing time. It makes objects look more real, but it does not make them look more three-dimensional. If your scene has no specular light, then all objects will look matte. A rough wooden table looks the same shape as a polished one. Specular light is a luxury. In "Specular Light" on page 130, you learn how to switch it off.

Light sources

Shockwave 3D provides four types of light source. A light object has a type property which can take one of the four following values:

- #ambient
- #directional
- #point
- #spot

Ambient Lights

By default, every 3D cast member has one ambient light object: `light("UIAmbient")`. You generally do not require more than one ambient light. The ambient light determines how dark the shaded areas will appear. By default, the UIAmbient light is black. This means that shaded areas will be 100% black. If you increase its brightness, shaded areas will appear gray. The position and orientation of the ambient light has absolutely no importance.

For more details on Ambient lights, see "Ambient light" on page 123.

Directional Lights

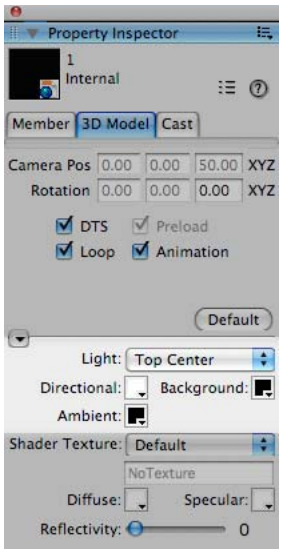
By default, every new 3D cast member has one directional light object: `light("UIDirectional")`, which is, by default, white. This gives the maximum contrast between lit areas and shaded areas. The position of a directional light has no importance. It acts as if its rays travel in parallel from infinity in the direction of the lights negative z-axis (from positive to negative). Changing the rotation of a directional light will change the way it illuminates each triangular face of the models in the 3D world.

For the best effects, the ambient light and main directional light need to be of the same color.

For more details on Directional lights, see "Directional lights" on page 124.

Using the Property Inspector to set the default lighting

Select a 3D member or sprite, open the Property Inspector at the 3D Model tab, and make basic adjustments to light("UIAmbient") and light("UIDirectional").



Lighting controls for 3D cast members in the Property Inspector

Point lights and spot lights

Point lights and spot lights are more complex than directional lights. A point light occupies a particular point in space, and sends out light in all directions. You can set a point light's `attenuation` property so that its brightness decreases with distance.

A spot light has the same characteristics plus two additional properties: `spotAngle` and `spotDecay`. The `spotAngle` property sets the angle of the cone of light. The spotDecay property gives you a simple way to make the spot light's brightness decrease with distance.

Moving a point or a spot light around will change both the angle and the distance from which the light falls on the nearby faces.

For more details on Point and Spot lights, see "Point lights" on page 125 and "Spot lights" on page 126.

Types of light sources and computing time

The following table shows what calculations the video card needs to make for every surface for each type of light source:

	color/ intensity	angle of incidence	specular light	light position	distance	cone
Ambient light	Yes					
Directional light	Yes	Yes	Optional			
Point light	Yes	Yes	Optional	Yes	Optional	
Spot light	Yes	Yes	Optional	Yes	Optional	Yes

The more lights you have in a scene, and the more complex those lights are, the slower the movie will run. Adding an ambient light to a scene has virtually no effect on computation time. Adding a single spot light to a scene can slow down the playback of the movie noticeably.

To get acceptable results on slow computers, you can use a very limited number of directional lights with their `specular` option switched off. See "Specular Light" on page 130 for more details.

The shortcomings of lighting in Shockwave 3D

Shockwave 3D takes a number of shortcuts to create a realistic-looking scene with the least computer processing time. One critical shortcut is that light passes straight through all models in its path. This has two unrealistic effects:

- Models do not cast shadows on the models around them
- Objects cannot be reflected in mirrors.

Shadows

Your brain relies on information from shadows to determine how close objects are to each other. For situations where this information is important, Shockwave 3D developers have found workaround solutions. You can find a demonstration of real-time shadows here.

You can find a tutorial for different technique for creating shadows here.

Mirrors

You can simulate mirrors and other flat shiny surfaces by using strategically placed mirror image models. Here is an example.

Cameras

Cameras act as windows into a 3D world. Each camera that exists in a 3D cast member offers a different view into it, and each sprite that uses a 3D cast member uses one of these cameras. You can use the Director 3D behaviors or script to manipulate camera positions

All 3D cast members have a camera called `camera("DefaultView")`. This camera cannot be deleted. Each 3D sprite has one camera whose view fills the entire sprite. The viewpoint from the DefaultView camera is adopted by default when a sprite is placed on the Stage. This example shows how to access the main camera used by the sprite named "3D":

```
put sprite("3D").camera
-- camera("DefaultView")
vCamera = sprite("3D").member.newCamera("Custom View")
sprite("3D").camera = vCamera
put sprite("3D").camera
-- camera("Custom View")
```

Interface control

You can use the Shockwave 3D window and the Property Inspector to manipulate the DefaultView camera. These windows do not give you any control over other cameras, even if you change the camera of the sprite. To control other cameras, you must use scripting.

3D and 2D views

A 3D camera displays all the models visible in its field of view. It can also display 2D *overlays* and *backdrops*. Overlays appear in front of the visible models, backdrops appear behind all the models. For more details see "Overlays and backdrops" on page 112.

Refreshing the camera view

Every camera has a `colorBuffer` property. This determines what color to show in areas that are not filled with models, overlays or backdrops. You can set this background color for all cameras in the 3D cast member through the Background color chip in the Property Inspector at the 3D Model tab.

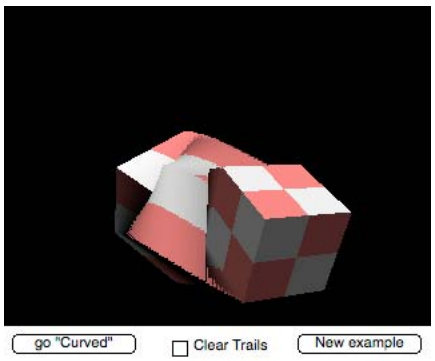
When a camera refreshes its view of the world, the render process follows this order:

- Fill view with background color
- Show backdrops
- Render visible models
- Show overlays

To clear or not to clear at render

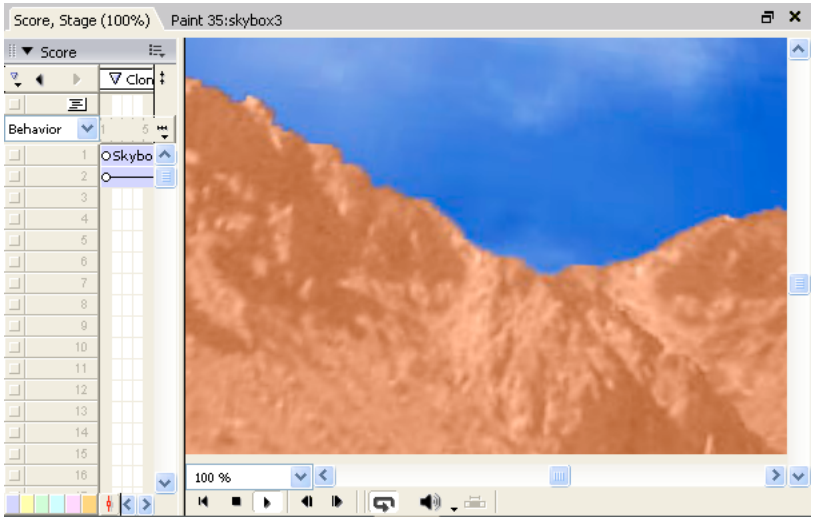
You can tell a camera not to clear the previous view with the background color. When the 3D sprite next refreshes its view, it will render the models in their new position on top of the existing view. This can be exploited in different ways: to create trails (if you use one camera), or to create a skybox effect (if you use multiple cameras).

To see an example of trails, download the movie Interpol.dir and launch it.



Setting a camera's colorBuffer.clearAtRender property to FALSE can create trails.

To see an example of a skybox, download the movie Skybox.dir, , and launch it. You can use the mouse and arrow keys, and keys I, O, W and S to move the camera.



One camera renders the skybox, then camera("DefaultView") adds its view of the ground as a separate layer.

For details on how to create a skybox, see "Sky box" on page 108.

Multiple cameras

As the skybox example demonstrates, you can show the output of more than one camera at a time in the 3D sprite. You can also create on inset view from one camera on top of the image from another camera.

Fog

The pastel quality of the main garden scene above is created with *fog*. There is no fog in the Inset camera, so the colors are crisper.

In the real world, the air is not perfectly clear. There is dust in the air. Light that reaches you from distant objects is dimmed by the dust. Objects that are close to you have vibrant colors, but distant mountains seem pale.

For more information on the use of fog with a camera object, see "Fog" on page 111.

Field of view

Our eyes see the world in perspective. Objects in the real world seem smaller the further away they are.

An engineer or an architect needs to create 3D models that indicate precise dimensions. A component at the far end of a piece of apparatus needs to have the same dimensions as a similar component close to the viewer.

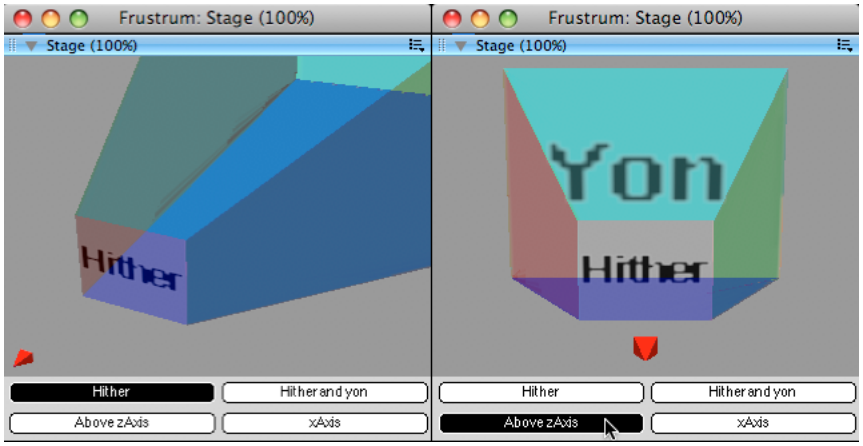
Shockwave 3D provides two different ways of projecting the view from a camera onto the sprite:

- #perspective
- #orthographic

Frustrum

A camera has a blind spot just in front of it. It cannot see anything that is closer than 1 world unit. When a camera is showing a perspective view, the visible volume is in the shape of a *frustrum*, or truncated pyramid.

To see the shape of a camera frustrum for a perspective view, download the movie Frustrum.dir and launch it.



The red shape represents the camera. The colored form represents the volume in which models are visible.

For an orthographic view, the frustrum is box shaped.

Hither and Yon

The *hither* plane of a camera is like the frame you make with the thumb and forefingers of both hands when imagining a camera view.

The hither plane of a camera

The camera will ignore everything that is closer to it than the hither plane. The view in the sprite is the view projected onto that plane. By default, the distance to the hither plane is 1 world unit.

The *yon* plane of the camera is further away than the hither plane. The camera will ignore everything beyond the yon plane. By default the distance to the yon plane is over 3.4 x 1038 world units away. If you consider world units to be a micron, the default yon plane is many galaxies away.

You can set both the `hither` and the `yon` distance.

Overlays and backdrops

Backdrops are textures displayed behind of all models appearing in a given camera's view. Backdrops are drawn before any of the models are rendered. They are shown in the order that they appear in the camera's backdrop list. The first item in the list appears behind all other backdrops and the last item in the list in front of all other backdrops.

Overlays are textures displayed in front of all models appearing in a given camera's view. Overlays are drawn after all models have been rendered. As with backdrops, the order in which they are drawn depends on the order in which they are created.

Unlike models, the position of overlays and backdrops does not change as the camera moves through 3D space. They appear in fixed positions, as if glued to the screen.

Each camera has its own list of overlays and backdrops to display.

You can use overlays and backdrops to:

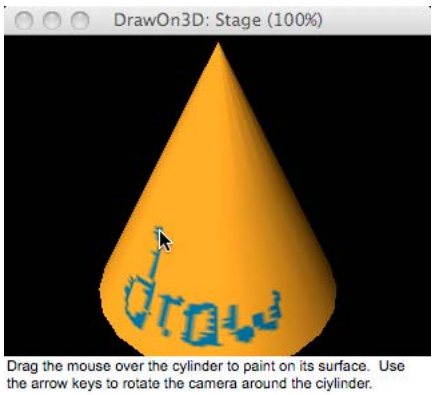
- Create a background
- Create a border
- Show static and dynamic text
- Display an interactive button
- Show toolbars
- Display tool tips
- Display callouts
- Create a progress bar
- Design a head-up display

For details on using overlays and backdrops with cameras, see "Overlays and backdrops" on page 112.

Interactions

Like text and field sprites, 3D sprites can receive keyboard focus. Like all graphic sprites, 3D sprites can receive mouse events. You can interact with a 3D sprite using both the keyboard and the mouse.

To see a demonstration of mouse and keyboard interactions, download the movie DrawOn3D.dir, and launch it.



When you click on a 3D model, Director can tell with precision exactly where you clicked.

Mouse interactions

When you click on a 2D sprite, Director knows exactly which pixel you clicked. When you click on a view into a 3D world, there are an infinite number of points under the mouse.

Imagine that you point into a crowded scene with a thin laser beam that can pass through anything in its path. The laser beam can touch many objects along its path.

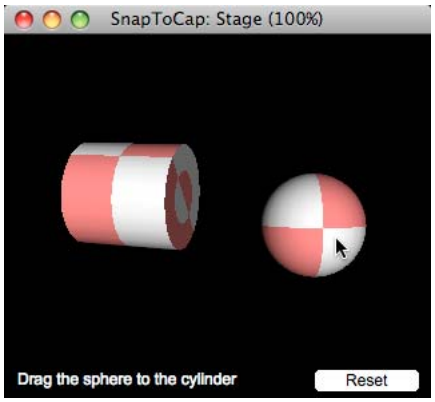
You can send such a ray into the 3D world, and ask the camera to tell you which objects are in the path of the ray. You can find more information about this at modelUnderLoc() and modelsUnderLoc().

If there are no models in the path of the ray, the camera gives you the 3D co-ordinates of one of the points in space that the ray passes through. See spriteSpaceToWorldSpace(). Conversely, you can use worldSpaceToSpriteSpace() to find where a particular point in space appears on the surface of the 3D sprite.

For more information on detecting which models lie under a particular location in the 3D sprite, see "Picking" on page 242 and "Pick Action behavior" on page 244.

Dragging

With the help of the functions mentioned above, and some 3D mathematics, you can drag a model around inside the 3D world. To see this in practice, download the movie SnapToCap.dir and launch it.



In SnapToCap.dir, you can drag the sphere around. When it gets close to the cylinder, the sphere will lock in place.

For more information on how to drag models around inside a 3D sprite, see "Dragging" on page 250.

Steering

Another common use of the mouse is to steer a first-person avatar through a scene. You can achieve this in different ways, including:

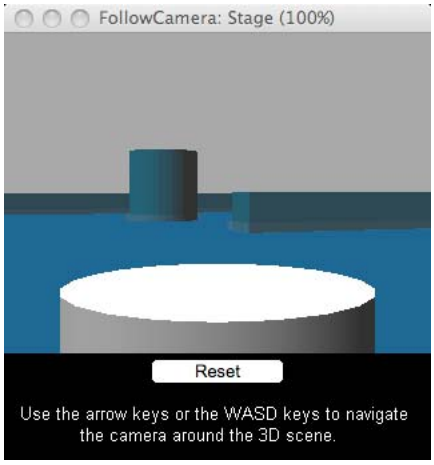
- Click on a destination point.
- Move the mouse left or right to turn in that direction, and up or down to move forwards or backwards.

For more information on using the mouse to move around inside a 3D world, see "Steering with the mouse" on page 221.

Steering with the keyboard

Another standard steering technique is to use the keyboard. Right-handed players with an English-language keyboard like to keep their right hand on the mouse and use the W, S, A, and D keys to move forwards and backwards and to turn left and right. Left-handed players may prefer to place their fingers on the arrow keys for the same controls.

To test one approach to steering with the keyboard, download the movie FollowCamera.dir, and launch it.



In FollowCamera.dir, use WASD or the arrow keys to steer the red cube around the landscape

For more information on how using the keyboard to interact with a 3D world, see "Steering with the keyboard" on page 224.

Modifiers

Shockwave 3D provides 8 modifiers that you can attach to specific models in a 3D cast member. Modifiers let you control many aspects of how models are rendered and how they behave. When you attach a modifier to a model, you can then set the properties for that modifier with script. Depending on the type of modifier you use, setting its properties can give you fine control over the model's appearance and behavior.

Appearance modifiers

1. Level of Detail (LOD)
2. Subdivision Surfaces (SDS)
3. Toon
4. Inker

Behavior modifiers

- Collision

Animation Modifiers

1. Bones Player
2. Keyframe Player
3. Mesh Deform

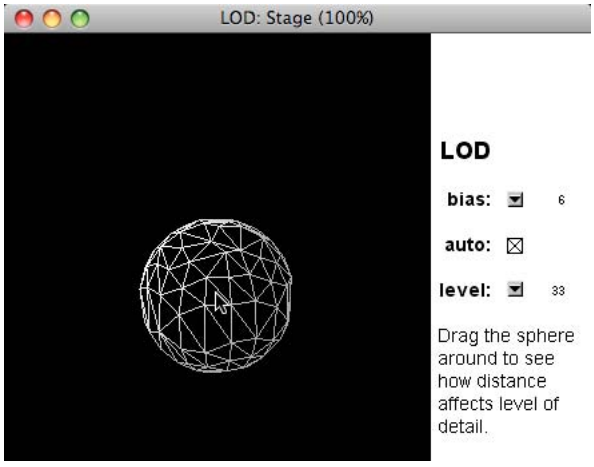
Level of Detail (LOD)

In the real world, as objects get further away, the less detail is visible. In a virtual 3D world, faces that are far from the camera appear tiny. Nevertheless, the computer processor must spend as much time calculating the color of the pixels for a distant face as for a face close to the camera.

The solution is to simplify the geometry of distant models, so that they have fewer faces. All model resources created in third-party 3D design software contain data that allows the Shockwave 3D engine to reduce the level of detail on models far from the camera. You can set the `lod` properties of the imported model resource directly.

Even when you change the settings at the resource level, the LOD feature acts on the model, not on the resource. Two models that share the same resource can display different numbers of faces, depending on how close they are to the camera.

The `#lod` modifier lets you give individual models their own `lod` settings, independently of the `lod` settings for the shared model resource. To see the `#lod` modifier in action, download movie LOD.dir and launch it.



The #lod modifier lets you set how aggressively to simplify model geometry as distance from the camera increases

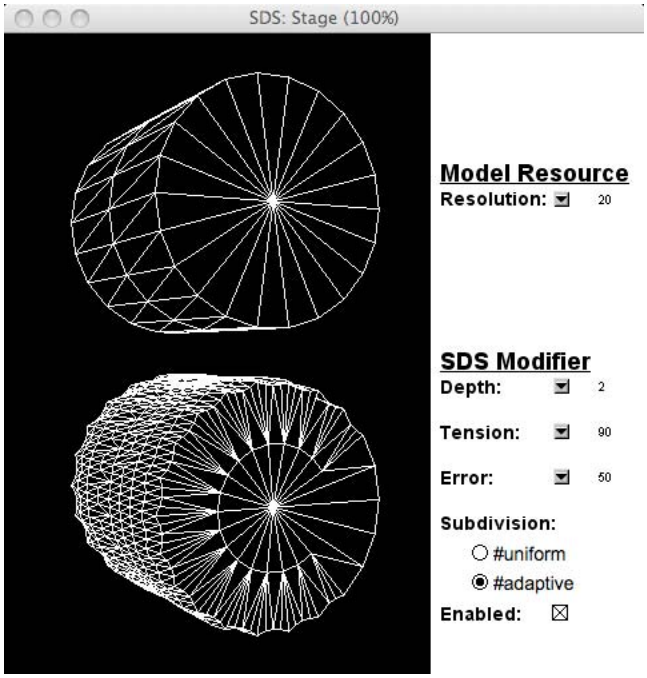
The LOD.dir movie uses a wire frame shader to make the effect obvious. Select a low value for `bias`, and ensure `auto` is checked. Now drag the sphere around. You will see the value of `level` change as sphere moves closer to away from the camera.

Note: *Models created in Director from 3D primitives do not contain LOD data. There is no advantage to be gained from adding the LOD modifier to such models.*

Subdivision Surfaces (SDS)

While the `#lod` modifier reduces detail on distant models, the `#sds` modifier adds geometric detail to models and synthesizes additional details to smooth out curves as the model moves closer to the camera.

To see the effect of the `#sds` modifier, download the movie SDS.dir and launch it.



The #sds modifier multiplies the number of faces displayed by a model. This can deform the shape of the model.

The `#sds` modifier and the `#lod` modifier behave in contrasting ways. It is not advisable to use both modifiers on the same model. The results may be unpredictable.

The `#sds` modifier is best used on models that were created in a third-party 3D design application. If used with models created from Shockwave 3D primitives, the modifier affects the model resource, and all the models that share that resource.

Note: *If you are using spot lights with a narrow beam, then smaller faces will give a better result. A face will only be lit correctly if all three vertex points are within the light beam. If the faces are large, compared to the light beam, the beam may "hit" the center of a face but not its vertices. In this case, the face will not light up at all. See "How faces are lit" on page 127 for more details.*

Toon and Inker

The `#toon` and `#inker` modifiers interact with the model's shader. The `#inker` modifier adds silhouettes, creases, and boundary edges to an existing model; the `#inker` properties allow you to control the definition and emphasis of these properties.

The `#toon` modifier draws a model using only a handful of colors, resulting in a cartoon style of rendering of the model's surface. When the `#toon` modifier is applied to a model, the `diffuse` color of the model's shaders will be respected but many of the other properties of the shaders are ignored. The `#toon` modifier and the `#painter` shader act in similar ways. If you use them both together, the properties of the `#painter` shader will be overridden. Experiment to see which of these gives you the most pleasing effect.

Note: *The `#inker`, `#toon`, and `#sds` modifiers cannot work together on the same model. If you add one of these modifiers to a model, any attempt to add one of the others will fail silently.*

Collision

The `#collision` modifier manages the detection and resolution of collisions. It is easy to detect collisions between models with simple geometry, such as spheres and cubes. Calculating collisions between models with more complex geometry (bananas or teapots, for instance) requires much more processing time.

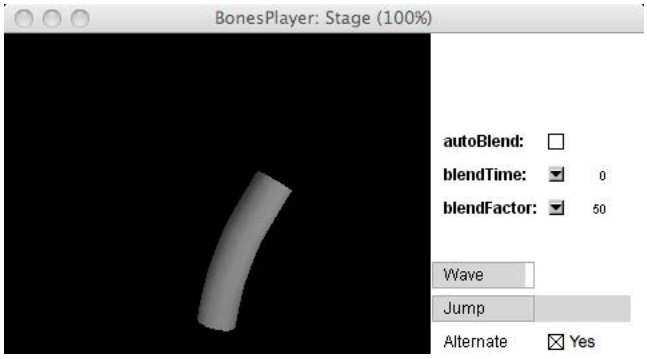
The `#collision` modifier was introduced in Director 8.5. It can handle simple collisions. However Director 11.5 features a much more powerful collision detection system, provided by the Dynamiks xtra extension. See "Physics" on page 293 for more details.

Bones Player

The `#bonesPlayer` modifier manages the use of motions by models. The motions managed by the `#bonesPlayer` modifier animate segments, called bones, of the model. Bones animations modify the model's geometry over time.

Motions and the models that use them must be created in a 3D modeling program, exported as W3D files, and then imported into a movie. Motions cannot be applied to model primitives created within Director. Creating bones animation in a 3D modeling application can be complex, but it results in more natural-looking movements.

The `#bonesPlayer` modifier lets you launch one motion at a time on any given model, but you can blend the end of one motion into the beginning of the next motion in a couple of different ways. To explore this blending feature, download the movie BonesPlayer.dir, and launch it.



In BonesPlayer.dir, you can experiment with the autoBlend, blendTime and blendFactor settings.

You can queue, play, and pause bone animations, and you can vary the rate at which they are played. You can use motion mapping to create new motions out of a combination of existing motions, and play these new motions back using the `#bonesPlayer` modifier.

Keyframe Player

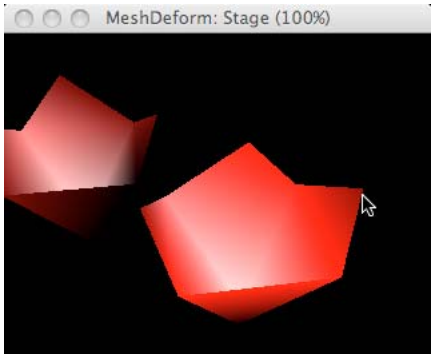
Keyframe animations modify a model's `transform` properties over time. Like all the other modifiers, the `#keyframePlayer` modifier can only be attached to a model. Perhaps you want to apply a keyframe motion to a camera, a light or a group. To do this, you can use a placeholder model with its `resource` property set to `VOID`, and add the camera, light or group is a `child` of the placeholder model.

You can combine a keyframe animation with a bones animation on the same model. For example, for an animated character, you can combine a "run in place" bones animation with a "move around the room" keyframe animation.

Mesh Deform

If you attach the `#meshDeform` modifier to a model, you gain access to the vertex and face data of the model's resource. You can use this information to move individual vertex points around. Any changes that you make to a model's resource via the `#meshDeform` modifier are visible on all models created from that resource.

To see this in action, download and launch the movie SimpleMeshDeform.dir.



Click on a vertex point and drag it around

The SimpleMeshDeform.dir movie contains two models created from the same model resource. When you move individual faces on one model, you see the effect duplicated on the other.

Notice that you can only drag the gray outside faces of the octahedron. The red faces (lit by a point light at the center of the shape) face "backwards". Backward-facing polygons are not detected by the `modelsUnderRay()` function.

Although the red light is inside the octahedron, the internal faces continue to glow red, even when they are turned to face outwards. This is because lighting is applied to the faces in their original orientation, even after the model's mesh has been deformed. See "Manipulating a mesh resource" on page 182 for details on how to adjust the lighting effects.

Motions

A *motion* represents a predefined animation sequence that involves the movement of a model or a model component. Original motions must be authored in your 3D-modelling application. With Director, you can combine existing motions to create new motions, but you cannot create a motion from scratch.

Individual motions can be set to play by themselves or with other motions. For example, a running motion can be combined with a jumping motion to simulate a person jumping over a puddle.

You can create a reference to a motion by using the `motion` property of the 3D cast member object. You can identify an individual motion by its name or by its index position in the list of motions.

With the `#keyframePlayer` modifier and the `#bonesPlayer` modifier, you can use a set of motions authored in your 3D-modeling application. For bones animation, each motion contains a list of tracks, and each track contains the keyframes for a particular bone. A bone is a segment of the skeleton of a model. For example, track 14 of the motion named Run can be named `RtKneeTrack` and move a bone named `RtKnee`. These names are defined in the 3D modeling application.

- **Play list**: The Keyframe and Bones players manage a queue of motions. The first motion in the play list is the motion that is currently playing or paused. When that motion finishes playing, it's removed from the play list and the next motion begins.

  Motions can be added with *`bonesPlayerOrKeyframePlayer`*`.play("motionName")`, which adds the motion to the top of the list, or *`bonesPlayerOrKeyframePlayer`*`.queue("motionName")`, which adds it to the end of the list. Using the play method starts the motion immediately. The motion previously at the beginning of the play list is halted unless `autoBlend` is turned on. When you use `queue()`, the motion is added to the end of the play list. Motions are removed from the play list automatically when they are complete. You can remove them explicitly by using `bonesPlayer.playNext()`.

- **Motion blending**: If `autoblend` is `TRUE`, a motion that is coming to an end blends smoothly into the next motion. You use the *`bonesPlayerOrKeyframePlayer`*`.blendTime` property to determine how long the blend should be. If you set *`bonesPlayerOrKeyframePlayer`*`.autoBlend` to `FALSE`, you can then use *`bonesPlayerOrKeyframePlayer`*`.blendFactor` to control the blend frame by frame.
- **Motion mapping**: You can create new motions by combining existing motions. For example, a walking motion could be combined with a shooting motion to produce a walk-and-shoot motion. This is available only with Bones player animations.

You can add the Keyframe player modifier at runtime to a model created in Director, but you cannot add the Bones player modifier at runtime. The Bones player modifier is automatically attached to models with bones animation exported from a 3D-modeling application. Since the required bones information cannot be assigned in Director, it has to exist before the model is imported into Director.

Physics

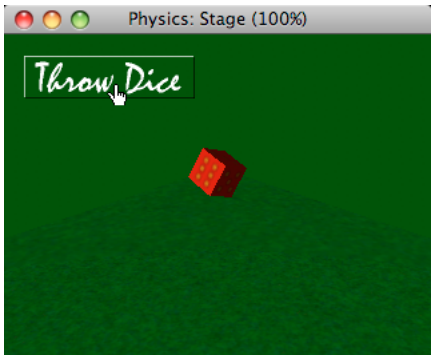
The Physics Xtra is a high-performance tool that helps developers create 3D worlds in which objects interact. The Xtra performs calculations to determine the results of collisions, factoring in object properties such as mass, velocity, and rotation. Forces can be applied, and objects can be connected to each other with constraints. The constraints available are 6 degree of freedom joints, linear joints, angular joints, and spring joints.

Additionally, terrains and raycasting are supported. A terrain is similar to a bumpy plane that is finite in two dimensions and defines an elevation along the third. Raycasting is the mechanism of collision detection with rays. Raycasting can be done against all types of rigid bodies and terrains.

With this Xtra, developers can focus on game play and user interaction, and not worry about creating a real-time physics engine with Lingo scripts.

The Physics (dynamiks) Xtra is a fully integrated rigid body physics simulation engine for Adobe® Director®. The dynamiks Xtra is supported on Windows and MAC platforms.

For a simple demonstration of some of the potential of the Physics Xtra, download the movie Physics.dir and launch it.



The Physics.dir movie simulates rolling a gaming dice

The Physics.dir movie demonstrates many of the features discussed earlier in this section, including:

- Defining a 3D shape
- Placing nodes in virtual space
- Using a shader to change the appearance of a model
- Using a texture to place an image on the surface of a model
- Lights
- Camera
- Overlays and backdrops
- Using raycasting to detect which model is at a particular location
- Using a modifier (MeshDeform)

If you can understand the principles that are used in this movie, you are ready to start exploring 3D scripting. You are ready to create interactive virtual worlds of your own.

Review

Inside a 3D sprite, you see colored pixels. The color of these pixels is calculated to give you the illusion that a 3D world of solid objects exists. The solid objects, or *models*, are defined as a collection of triangular *faces*. The geometry for these faces is stored in a *model resource*. Multiple models can be created from one model resource.

Models can be positioned, rotated and scaled in 3D space. A set of complex lighting algorithms determines the color and shading for each individual face of each model. These algorithms take into account:

- The geometry of the model resource
- The position and orientation of the model in space
- The shaders, textures and appearance modifiers that have been applied to the model
- The orientation, position and distance of each light source
- The placement of the camera

Models can be grouped together in parent-child relations. They can be animated by both keyframe and bones motions. To make models appear to be solid objects in a world that obeys the laws of physics, you can apply a variety of physical characteristics to the models.

You can interact with the 3D world through the keyboard and the mouse.

A virtual 3D world is far less complex than the real world, but you get to make all the rules. The limits of what you can create in a virtual world depend only on your imagination.

3D output

Director can output 3D content in three different ways:

- Render a 3D world to the computer screen. See "3D Renderers" on page 57. On some computers, you can benefit from hardware antialiasing. See "3D Anti-aliasing" on page 57.
- Save changes that you make to a 3D world inside Director. See "Saving the 3D world" on page 58.
- Export the current state of a 3D world to an external W3D file. See "Saving the 3D world" on page 58.

3D Renderers

Shockwave 3D relies on the 3D rendering technologies present on your video card. If no hard-ware accelerated 3D rendering is available, you can use the built-in `#software` renderer. This is slower and not as sophisticated as hard-ware based technologies.

You can use the getRendererServices().rendererDeviceList to determine what renderers are available on the client machine.

This returns a list of symbols. The contents of this list determine the range of values that can be specified for the renderer and preferred3DRenderer properties. This property can be tested, but not set.

This property is a list that can contain the following possible values:

`#openGL` specifies the openGL drivers for a hardware acceleration, which work with both Mac and Windows platforms.

`#directX9` specifies the DirectX 9 drivers for hardware acceleration that work only with Windows platforms.#auto sets the renderer to DirectX 9. In Mac-Intel, only #OpenGLrenderer is available.

`#directX7_0` specifies the DirectX 7 drivers for hardware acceleration, which work with Windows platforms only.

`#directX5_2` specifies the DirectX 5.2 drivers for hardware acceleration, which work with Windows platforms only.

`#software` specifies the Director built-in software renderer, which works with both Mac and Windows platforms.

3D Anti-aliasing

In Adobe Director 11.5.8, hardware anti-aliasing is supported in the Shockwave 3D asset. The `antialiasingMode` property can be set for anti-aliasing the 3D Sprites.

The `antialiasingMode` property supports the following options:

- `#default` (no anti-aliasing)
- `#multisample2x`
- `#multisample4x`

- `#multisample8x`
- `#multisample16x`

For more information, see "Antialiasing" on page 390

Saving the 3D world

Changes made to the 3D world can be saved in authoring and projectors. The saved world can be reused as a new cast member. You must save the Director movie for the changes to the 3D world to get effected. Saving

To save changes to an internal 3D cast member while authoring use the member3D.saveWorld() method. This does not save the changes immediately: you then need to save the movie. The `saveWorld()` method simply marks the member as edited. Its modified state will be written to the movie file when the Director movie is saved.

To create an external W3D file while authoring or from a projector, use the member3D.savew3d() method.

Note: *Overlays and backdrops will not be saved with the 3D world. These are runtime properties of a particular camera. The code that creates overlays and backdrops must be executed each time you want to display them in your 3D sprite.*

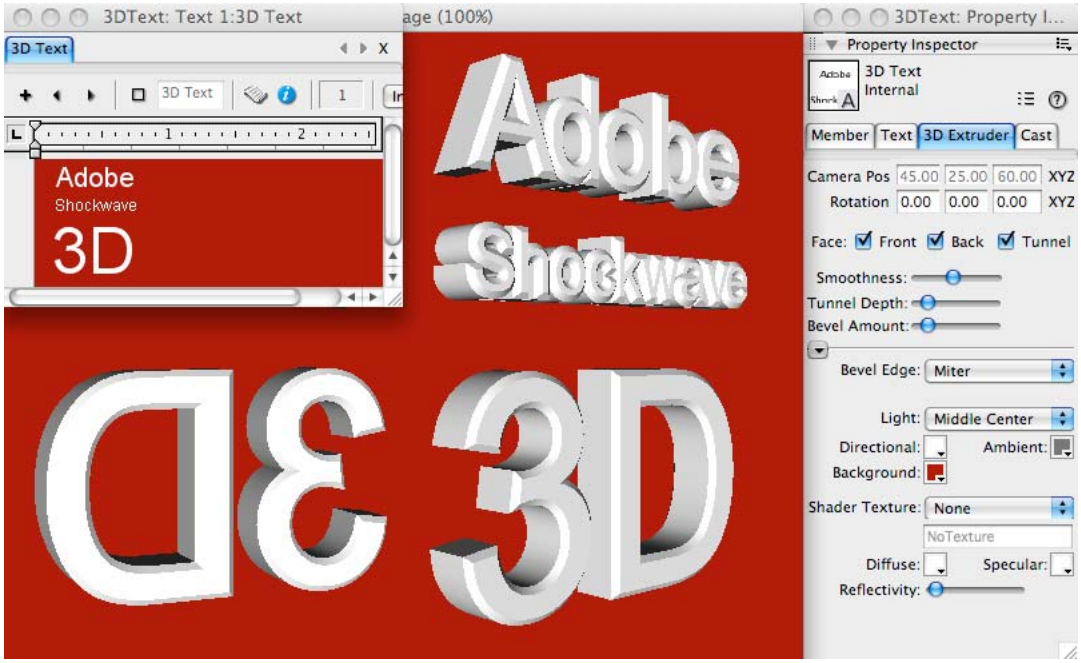
3D text

You can easily create 3D text in Director.

1. Create a normal (2D) text cast member.
2. Convert the text to 3D by selecting 3D Mode from the Display menu on the Text tab of the Property Inspector.
3. Set properties of the 3D text by using the 3D Text tab to manipulate the specific properties of the 3D text.

You can also manipulate the text cast member with script or a behavior. For more information, see "Creating 3D text" on page 59.

To see an example, download the movie 3dText.dir and launch it.



You can create a rotating logo in Shockwave 3D with no knowledge of scripting

extrude3d()

An alternative solution is to use the textMember.extruder3d() function to create a model resource inside a 3D cast member.

Creating 3D text

To create 3D text, first create 2D text, and then convert the text to 3D.

1. Select Window > Text to open the text editor.
2. Select the desired font, size, style, and alignment. Most standard fonts work well with 3D text. Experiment to get specific results.
3. Enter the text. (You can edit the text after you have entered it).
4. Drag the text cast member onto the Stage. Either drag the cast member from the Cast window, or drag the Drag Cast Member button next to the Name text box in the Text window.
5. Click the Property Inspector button in the Director toolbar.
6. Click the Text tab in the Property Inspector.
7. Select #mode3d Mode from the Display menu.

The text on the screen changes to 3D. See "Modifying 3D text" on page 60 for next steps.

Modifying 3D text

After your 2D text has been changed to 3D, you can modify it.

1 Click the Property inspector 3D Extruder tab.

2 Set the camera position and rotation. As with the standard 3D Property inspector tab, you control camera position and rotation with the values that you enter in the text boxes at the top of the pane. The default camera position represents a vantage point looking up through the middle of the scene.

Note: *You can also define these settings by using the Shockwave 3D window. Use Window > Shockwave 3D, then use the Next Cast Member and Previous Cast Member buttons to locate the 3D Text member that you want to work with.*

3 Select from among the Front Face, Back Face, and Tunnel check boxes. These options control which sides of the text appear.

4 Set the smoothness. Smoothness determines the number of polygons that are used to construct the text. The more polygons that are used, the smoother the text appears.

5 Set the tunnel depth. Tunnel depth is the length of the tunnel from the front face to the back face.

6 Select a beveled edge type. Beveling makes the edges of the 3D letters appear rounded or angled. Select Round for rounded edges or Miter for angled edges.

7 Select a bevel amount. Bevel amount determines the size of the bevel.

8 Set up the lighting. Select a color and position for the text's default directional light. A directional light is a point source of light and comes from a specific, recognizable direction. You can also select a color for the ambient and background lights in the 3D world that the text occupies. Ambient light is diffuse light that illuminates the entire world. Background light appears to come from behind the camera.

9 Apply a shader and a texture. Shaders and shader properties determine the appearance of the surface of the 3D text model. Textures are 2D images drawn on the surface of the text. Use the Property inspector to assign a texture to the text's shader. You can also control the color of the shader's specular highlights and its diffuse or overall color and reflectivity.

As with any model, you can apply a texture that uses a bitmap cast member. You can import a bitmap cast member or create a new one in the Paint window. Be sure to give your bitmap cast member a name if it does not already have one. To assign this bitmap as the texture, specify it in the Property inspector. Select Member from the Shader Texture menu, and then enter the name of the member you want to use in the text box to the right of the menu.

Script and 3D text

Director contains methods and properties in Lingo and JavaScript syntax for working with 3D text. Most 3D methods and properties work with 3D text exactly as with any other object. Properties are also described in the Scripting Reference topics in the Director Help Panel.

However, there are some standard text properties that cannot be applied to 3D text. You can find a list of these at "Exceptions" on page 61.

On the other hand, model resources created from 3D text have additional properties that are not available to ordinary 3D model resources. You can find these listed at "Lingo and JavaScript syntax script for 3D text" on page 62.

Exceptions

The following methods and properties work differently when used with 3D text.

Type of Script	Element
Member property	antiAlias
Member property	antiAliasThreshold
Member property	antiAliasType
Member property	picture
Member property	preRender
Member property	scrollTop
Member property	useHypertextStyles
Member property	autoTab
Member property	boxType #scroll
Member command	scrollByPage
Member command	scrollByLine
Member function	charPosToLoc
Member function	linePosToLocV
Member function	locToCharPos
Member function	locVToLinePos
Sprite property	editable
Sprite function	pointInHyperLink
Sprite function	pointToChar
Sprite function	pointToItem
Sprite function	pointToLine
Sprite function	pointToParagraph
Sprite function	pointToWord
Hypertext Lingo	hyperlinkClicked
Hypertext Lingo	hyperlink
Hypertext Lingo	hyperlinks
Hypertext Lingo	hyperlinkRange
Hypertext Lingo	hyperlinkState

Lingo and JavaScript syntax script for 3D text

In addition to working with most existing methods and properties, 3D text also adds some properties of its own. These properties give you a more precise way to define the characteristics of the text than is possible using the Property Inspector.

These properties can be set while the text is in 2D mode. They have no visible effect until the text appears in 3D mode.

When you access the properties listed in the following table for an extruded 3D text model that you created by using the `extrude3D()` method, you must refer to the model resource of the text. The Lingo syntax for this is shown in the following example:

```
member(whichMember).model[modelIndex].resource.3DTextProperty
```

For example, to set the `bevelDepth` property of the first model in cast member 1 to a value of 25, use the following syntax:

```
member(1).model[1].resource.bevelDepth = 25
```

Properties of extruder resources for text

Property	Access	Description	Range or Default
`bevelDepth`	Get and set	Degree of beveling on front or back faces.	Floating-point value from `1.0` to `100.0` Default is `1.0`

`bevelType`	Get and set	Type of bevel.	`#none`
			`#miter`
			`#round`
			Default is `#miter` for 3D text in a Text member and `#none` for extruder resources in a 3D cast member.

`displayFace`	Get and set	Faces of shape to display.	`#front`
			`#tunnel`
			`#back`
			Default is to show all three faces

`smoothness`	Get and set	Number of subdivisions for curved outlines.	Integer from `1` to `100` Default is `5`
`tunnelDepth`	Get and set	Extrusion depth.	Floating-point value from `1.0` to `100.0`

Note: *The properties in the table above can be applied either to the extruder resource in a Text member in 3D mode, or to an extrude resource created in a 3D member by the* `textMember.extrude3d(member3D)` *function.*

Property of Text members

Property	Access	Description	Range or Default

`displayMode`	Get and set	Specifies how the text appears.	`#modeNormal`
			`#Mode3D`
			Default is `#modeNormal`, which is 2D text.

3D Method for Text members

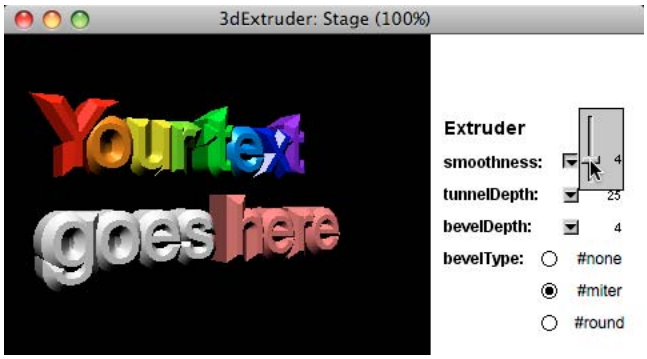
`member("text member").extrude3d(member("3D member"))`	Creates a new model resource in member "3D world" by extruding the text in member "text member".

Adding a text model to a 3D cast member

To add 3D text to a 3D cast member, you can use the textMember.extrude3D() method. This creates an extruder model resource inside the 3D cast member of your choice. Each character in the text member is converted to a separate mesh inside the extrude resource. This means that you can apply a different shader to each letter.

Note: *You have to apply different shaders manually.*

To see an example of this, download the movie 3dExtruder.dir, , and launch it.



The extruder3d() command creates an extruded text model inside a 3D world

The extruder model resource possesses the same properties as the text in a 3D text member. See "Lingo and JavaScript syntax script for 3D text" on page 62for the full list.

Text in overlays and backdrops

If you want to display 2D text in a 3D sprite, you can create a texture from a bitmap or a text member, and display that as an overlay or backdrop of a camera. The process for doing this is quite different from the process for creating models in the shape of 3D text.

See "Text in overlays and backdrops" on page 116 for more information.

Sources of 3D content

Primitives

Models are the objects you see in the 3D world. You can create models within Director. Spheres, boxes, planes, cylinders, and particle systems can be created either with Lingo or JavaScript syntax or with Director behaviors. These simple shapes are called *primitives*. They are the basic shapes from which more complicated models are built.

Particle systems are different from the other primitives. Instead of being shapes, they create cascades of moving particles. You can also create mesh primitives, which allow you to define any kind of complex shape you wish. You can create extruder resources from text members, to add 3D text to the scene in your 3D cast member.

3D modeling applications

For the most part, you should create complex models outside of Director, use a 3D modeling application, and then import them into Director in the W3D file format.

Model resources defined by a W3D file imported into Director or loaded through a script have a type value of `#fromfile`. File-defined resources are automatically assigned level of detail (LOD) modifier settings that allow models using those geometries to adjust their level of detail as needed, depending on the model's distance from the camera. For more information, see "Level of detail (LOD) modifier" on page 193.

Combining sources

You can build a world in a single 3D cast member from multiple sources. For example, imagine a 3D cast member called "World" containing all the models for a particular scene, and other 3D cast members, each containing one animated character. You can clone the models and animations from the character cast members into the "World" cast member. You can create a custom sign from basic primitive shapes and extruded text that you generate from the end-user's own text input. You can add particle models to create flames and smoke and fireworks. The final scene can be much more than the sum of the parts.

External 3D Files

If you have access to third-party 3D design software, or a 3D designer, then the easiest way to create animated 3D content for your movies is as a W3D file. At the time of writing, the following software packages export to the W3D format:

- Autodesk's 3ds Max (32-bit install) is commonly used to model and export to W3D format.

  For high-quality character animation, one workflow is to start in Autodesk's Motion Builder, then to export the output in the FBX file format to 3ds Max (32-bit). Finally, you can export the animated character as a W3D file.

- NewTek's LightWave 3D

Converting to W3D

If your 3D design software package does not export to the W3D format, you can export to OBJ format, and then use the OBJ Converter (Windows only), to convert to W3D.

Other alternatives include:

- Right Hemisphere's Deep Exploration (no support for animations)
- Okino's Shockwave-3D Exporter (Windows only)
- Okino's Poly-trans Xtra (Windows only), which allows you to control the 3D Exporter from within Director.
- awaW3DTrans Xtra (Mac Intel and Windows)
- Sw3dC (Mac OS 10.6 and later, Windows XP and later)

Linked or imported W3D files

You can either import your W3D files as internal 3D cast members, or link to the external file. Linking a cast member to an external file makes it easy to update the file.

During development, your 3D design team can continue to work on the scene and the animations, and you can simply replace the external W3D file with the latest version.

Changes you make to the defaultRect and regPoint properties of the 3D cast member will be maintained when you replace the linked external file. Other settings, such as default camera position, background color, and lighting are read from the external file.

If you choose to import the W3D file, and the 3D designer makes changes to it, then you have to re-import the new version of the file. In such a case, all changes made to the 3D cast member will be lost.

Loading from external files

In many cases, you have to store models for your 3D world in several separate files. For example, you can save the geometry for the 3D scene in one W3D file and the animated characters in separate 3D files.

The loadFile() command allows you to copy the entire contents of an external W3D file into a 3D cast member. The following examples copies all the models, resources, groups, textures, shaders, and motions from the "WhiteKnight.w3D" file into the 3D cast member "Dark Castle", without replacing the existing objects:

```
vFile = "USBKey:3D:WhiteKnight.w3D"
vReplace = FALSE
member("Dark Castle").loadFile(vFile, vReplace)
```

SketchUp

Using Google SketchUp, you can create 3D models from scratch or customize models created by others. The Google 3D Warehouse is one such online repository from where you can download and use SketchUp models.

Note: *In 3D worlds, created in SketchUp, the world axes are rotated differently from a 3D world created in Director. In native Director 3D cast members, the y-axis points upwards. In a 3D world created in SketchUp, the z-axis points upwards. When you use the Shockwave 3D window to view a SketchUp world, remember to click on the Camera Z Up button before rotating the world.*

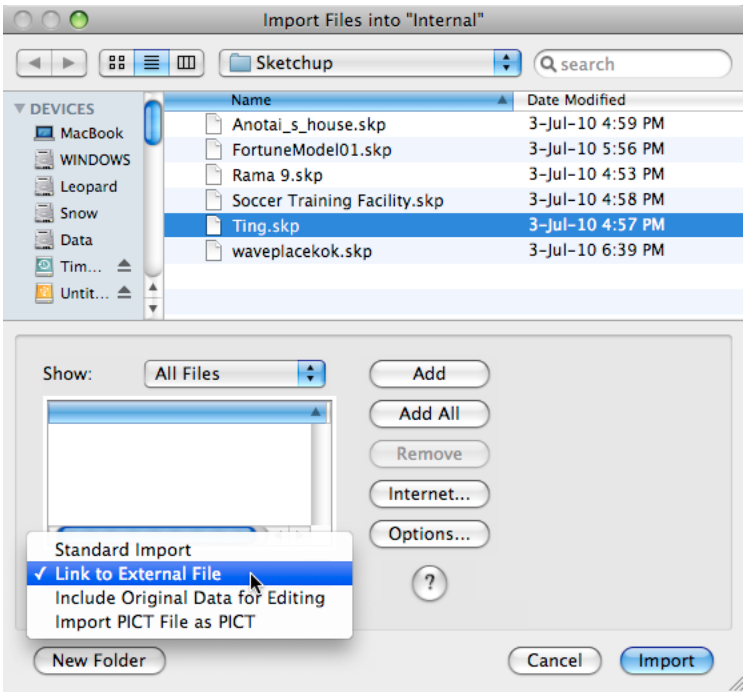
After a SketchUp model is imported into Director, it can be used like any other Shockwave 3D (W3D) file. You can apply all available 3D and Physics properties in Director to the imported model.

Import a Google SketchUp model

1. Click File > Import.
2. Select the SketchUp model in the Import Files dialog box, and click Import. Google SketchUp model files have a .skp extension.

   By default, the Google SketchUp model is imported and stored as a cast member inside the Director movie file.

   Alternatively, choose Link To External File from the Media pop-up menu to import the SketchUp model as an external W3D file. An updated copy of this external W3D file is imported to the cast every time you run the movie.



If you choose Link to External File, you will be prompted later to provide a path for the converted W3D file

3 In the Import Options dialog box, specify whether you want to import the hidden groups and layers in the model. The hidden groups and layers in the Google SketchUp model remain hidden even after they are imported into Director.

Note: *To unhide a model, set the model's visibility to* `front` *using* `model(hiddenModel.visibility=#front)`*. To unhide invisible faces in a model, use* `model(modelRef).shader(shaderRef).transparent=0` *to set the* `transparent` *property of the shader corresponding to the faces to 0.*

4 Specify if the model has two-sided faces. Director imports the two-sided faces in the model as two front-facing meshes.

Note: *If you do not select the Faces Are Two-Sided option, Director imports only the front faces of the SketchUp model. Information about the back faces is ignored. This may lead to some objects being invisible from certain angles.*

5 Specify how Director should import the hierarchy of groups in the Google SketchUp file. The SketchUp file can have multiple models arranged as nested groups.

Original Preserves the original hierarchy. The groups and models in the Google SketchUp file are replicated in the W3D file.

Collapsed Models that belong to the same group in the SketchUp file are collapsed into a single model in the W3D file. Any groups in the hierarchy are ignored during import.

Flattened Creates a one-to-one mapping of models between the SketchUp and W3D files. Any groups in the hierarchy are ignored during import.

Single Model Imports all groups and models in the SketchUp file as a single model in the Director W3D file.

1 Select the textures in the Google SketchUp model that you want to import.

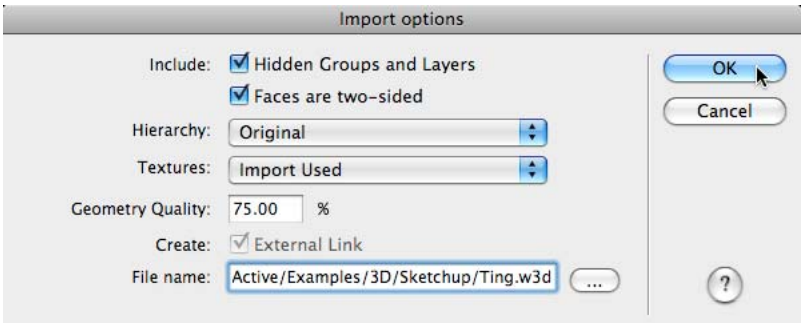
Import Used And Unused Imports all textures in the SketchUp file.

Import Used Imports only the textures used by the SketchUp model.

None Textures in the SketchUp file are not imported.

2 Enter the desired compression level for the imported geometries in the Geometry Quality box. By default, Director compresses the imported geometries to 75% of their original quality.

3 If you chose to import the model by reference as an external W3D file, enter a name for the file in the Create External Link box. This external file is imported afresh to the cast every time you run the movie.



The Create External Link check box will be disabled. Its state will depend on the choice you made at step 2

4 Click OK.

Import models through scripting

While authoring a movie, you can also import a Google SketchUp model into Director using the `importFileInto()` method.

- Use the following syntax to import the model as a cast member inside the Director movie:

```
member("some3dMember").importFileInto("anypath", [#Linked: 0])
```

- To import the model by reference (Link To External File), set the value of the `Linked` property to 1 instead of 0:

```
member("some3dMember").importFileInto("anypath", [#Linked: 1])
```

The SketchUp Import Options dialog will open. See steps 3 - 9 above for details.

You cannot import a SketchUp file at runtime. The SketchUp importation process can only be performed in the Director authoring application. If you want to link to external files containing content available as SketchUp files, you can select Link to External file when importing, as described above. If you have already imported the file internally you can:

1 Use saveW3D() to export the file in W3D format.

2 Link to the exported W3D file in the usual way.

Cloning from other 3D cast members

If you are creating a very simple 3D world, you may choose to place all your models, motions and other objects in a single W3D file. In many cases, however, you may choose to export your 3D content in a series of separate W3D files. You can use the loadFile() command to load the entire contents of an external W3D file into a 3D cast member.

If you are using linked or internal 3D cast members, you can have finer control of which objects you copy from one 3D cast member to another, or within a single 3D cast member. The following are the "clone" functions:

- node.clone() creates a copy of a model, group, light, or camera, and all its children. Any model resources and `shaderLists` used by the copy are identical to those used by the original. No new model resources, shaders, or textures are created.
- node.cloneDeep() copies the node and all its children, just like the `clone()` function, but it also creates a duplicate of all model resources, shaders, and textures used by the node or its children. This is useful if you want to modify these duplicate objects while leaving the original models unchanged.
- member3D.cloneModelFromCastmember() acts like `cloneDeep()` for a model, except that it copies the model and all its associated objects from one 3D cast member to another.
- member3D.cloneMotionFromCastmember() copies a motion from one 3D cast member to another. You can create an animated character with all its geometry, textures, and shaders in one W3D file. In separate files, you can save different motions for the character. In these separate files, you can omit all the shaders and textures, and reduce the geometry to little more than the bones that are animated by the motion. This will ensure that file size is kept to a minimum.

Export issues

Your first efforts at exporting to the W3D file format may not work out the way you expect. Many 3D designers establish a specific workflow that allows them to avoid conversion problems. This section lists the common issues that you are likely to encounter.

Lights

If there were no lights in the original pre-exported scene, then the W3D file will appear black. This can be simply fixed by adding a light either in the authoring application or in Director.

If your W3D scene appears empty, check whether it is simply a question of lighting.

Tiling and stretching textures

In 3ds Max, avoid making UVW settings to a texture in the Material Editor. UVW is a standard approach to set up tiling, stretching, and so on. Unfortunately W3D ignores settings made in the Material Editor. To use UVWs, you need to add a UVW Map modifier to the model itself and make your settings there.

If the way your texture maps to your model looks different in Director than it did in 3ds Max, check that you used a UVW Map modifier on the model.

Naming

In a W3D file, every object of a given type must have a unique name. Cameras, groups, lights, and models are all considered to be nodes. Every node must have a unique name. For example, you cannot have both a camera named “Main” and a group named “Main”. You can avoid this naming conflict by using a strict naming convention. For example, you can include the type of node in the name of the node. This gives you the names “Main Camera” and “Main Group”.

Depending on your 3D modeling application, you may be able to give the same name to different objects of the same type. Imagine, for example, that you create a leopard model and a zebra model, and that you create a different texture for each model. Your 3D modeling application may allow you to use the same name (for example "animal skin") for both textures. When you export the world as a W3D file, only one of the textures will be exported. The other will be ignored or overwritten. As a result, either your zebra or your leopard will appear with the wrong texture.

Do not rely on the default names given to objects by your 3D modeling application. If you find that some objects are missing in your W3D file, check that you are using a rigorous naming convention.

Hierarchy

The W3D exporter may decide to create a single model resource out of several separate models. This can happen for different reasons:

- If a model is a child of another model, it will become part of the model resource of the parent model not a child of the model. To avoid this, you can build the main hierarchy only with 3ds Max groups (not to be confused with Director groups), dummies, and helpers. Any object that is not a mesh in 3ds Max becomes a group in Director.
- If you group models together in 3ds Max, the group is exported as a single model resource. Grouping models can also cause the exporter to crash if the content is too big or too complex.

Exporting a selection

Your 3D modeling application may propose the option of exporting just the current selection. The W3D exporter is likely to ignore this option: the entire scene will be exported anyway.

Exporting for the #inker modifier

If you intend to apply the `#inker` modifier, include extra information during the export process. If you are working in Cinema4D, enable the option that adds extra information for hypernurbs. If you are working with 3ds Max, then enable the "toon" export option.

Exporting animated characters

In 3ds Max, bones characters will only be exported correctly if all the bones and meshes are selected and grouped. In addition, you need to make sure the envelopes covers the meshes entirely.

To animate human-like characters, you can use bipeds. Bipeds work very well in W3D, but in 3ds Max you must apply a Physique modifier to the biped, and group all bones and geometry into a single group.

Note: *You may find that some of the Physique settings are ignored in the exported W3D file.*

If you do not add the Physique modifier in 3ds Max, then the bonesPlayer will not be added to your animated biped.

An alternative approach is to use a Skin modifier. This can be used with animation data provided through motion capture. You can create all animations in 3ds Max using bones, and then:

- Group the mesh and biped
- Apply a Skin modifier to the mesh
- Adjust the weights of the bone's influence on individual mesh vertices
- Add a transparent dummy (this can be a cube with the rough outer dimensions of the animated mesh)
- Link the mesh/biped to the dummy

This technique allows you to use the dummy to translate and rotate the biped in Shockwave 3D.

Exporting multiple motions

W3D files can contain multiple motions for a given animated character. However, you may find that the W3D exporter accepts only a single motion per character. One solution is to export a separate W3D file for each motion. You can then use cloneMotionFromCastMember() to copy the separate motions to the 3D cast member that will be playing back the scene. See also "Cloning from other 3D cast members" on page 67.

An alternative is to add each motion in sequence to the character's timeline, and then use the millisecond timing of the start and end of each motion when you want to playback a particular motion sequence.

Exporting hidden objects

If you hide objects in your 3D modeling application before you export the scene, everything is exported anyway. Hidden objects will be exported as Director groups. Such a group defines a position, rotation, and a scale, but it includes no geometrical information.

Basic Model Preparation in 3ds Max

1. Create a model, or purchase a model from suppliers, such as, TurboSquid.
2. Convert to polygons if necessary.
3. Create a biped at the same origin point as the mesh, the same height as the mesh (Create > Systems > Biped)
4. Put the biped in figure mode so that you can edit the biped.
5. Get the overall height of the biped right, and then position his hips. Starting from there work outwards using non-uniform scale to fit the bones to the mesh. Widen the bones so that they nearly cover the mesh, so that they will influence the mesh when we add the skin modifier.
6. Put the viewport in wireframe and zoom in on the hips. Add a skin modifier to the mesh, click the Attach to Node button on the rollout, and select the biped's hips. This will open the skin initializer dialog, so click initialize.
7. Open the skin modifier and select vertex. Marquee select the whole mesh and look for blue vertices. These are not linked to a bone, so select them and click Assign to Link.
8. Create the dummy, which is a simple transparent cube that you use to translate and rotate the model. The dummy should be bigger that the mesh, but centered on the mesh center.
9. Go into the hierarchy browser, select both the mesh and biped, and select Group > Group to create a new group with them as the members. Give the group a good name. (If you do not do so, the mesh does not animate with the biped in Shockwave, even though it does in Max)
10. After you group, link the group to the dummy. To do so, select the group, and then select the link, and drag into an area where you only hit the dummy.

Applying motion capture data

Applying motion capture data is really no different for Shockwave3D than for any 3ds Max character. Motion capture data is a substitute for handmade animation motions, and is usually more fluid and natural than handmade animated motions, but may sometimes be less efficient at runtime. Motion capture files are available for purchase from many suppliers.

1. Select the mesh and click Display > Hide > Unhide All to show the biped.
2. Select the biped, Motion > Biped Apps > Figure Mode to take it out of Figure Mode.
3. Motion > Motion Capture > Load File to load the file. Initialize dialog box appears.

4 Select Footstep Extraction > Fit to Existing in initialize dialog box. If you do not do so, your mesh gets reshaped to fit the motion capture biped from the file.

5 Select the mesh, and click Display > Hide > Hide Unselected to hide the biped.

6 Set the animation length to match the motion capture (little clock icon in the lower right).

Native 3D content

You can create basic geometrical shapes in Director itself, without the need for a third-party 3D design application. The following are the simple shapes that are available:

- Sphere
- Plane
- Box
- Cylinder

These shapes are called *primitives*. Using groups and parent-child relationships you can use these simple shapes to create more complicated contraptions. For an example of such grouping, see "Using a parent to group several objects together" on page 33. For more information on the various geometrical shapes, see "Regular primitives" on page 71.

The following are the more complicated primitive types:

- Mesh
- Extruder
- Particle emitter

You can use a *mesh* primitive to define any kind of complex shape you wish. For more details, see "Mesh resources" on page 72.

You can use the `extruder3d()` function to extrude a 2D shape into a 3D object. For more details, see "Extruder resources" on page 74.

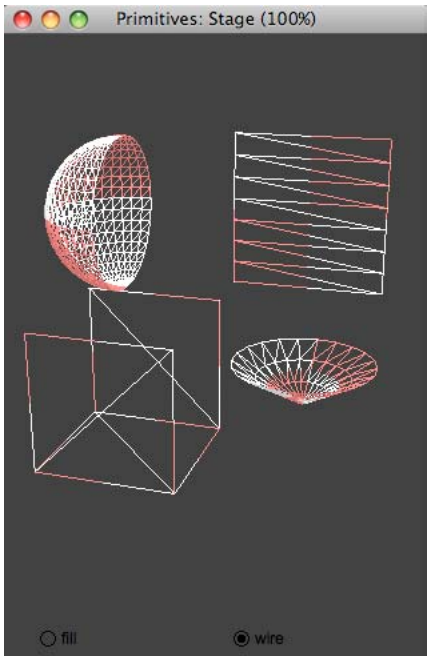
Particle systems create cascades of moving particles. See "Particle emitters" on page 74 for more details.

Regular primitives

Shockwave 3D provides four types of primitives with regular geometry:

- Sphere
- Plane
- Box
- Cylinder

The simplest way to explore all the properties of these primitives is to download the movie Primitives.dir, and launch it.



In Primitives.dir, use the PI to modify the properties of the different primitive resources

Note: *The default dimensions of the* `#sphere`*,* `#box` *and* `#cylinder` *primitives give them a width, height and length of 50 world units. The default size of the* `#plane` *primitive gives it a width and a length of 1 unit. The* `#plane` *resource's* `length` *property actually determines its height. The* `#plane` *primitive is rotated so that it is visible from the default camera position. To create a horizontal plane, you will need to rotate the model -90° around the world's x-axis.*

See "Using a parent to group several objects together" on page 33 for an example of a 3D contraption made from several primitive models.

Mesh resources

You can create any 3D shape you like using a `#mesh` resource. Creating a `#mesh` resource requires the same amount of concentration as building a model out of matchsticks. To create a `#mesh` resource, you must provide many different pieces of information. The minimum information required is:

- The position of each vertex point in the mesh.
- A list of which vertex points are joined together to create each face.

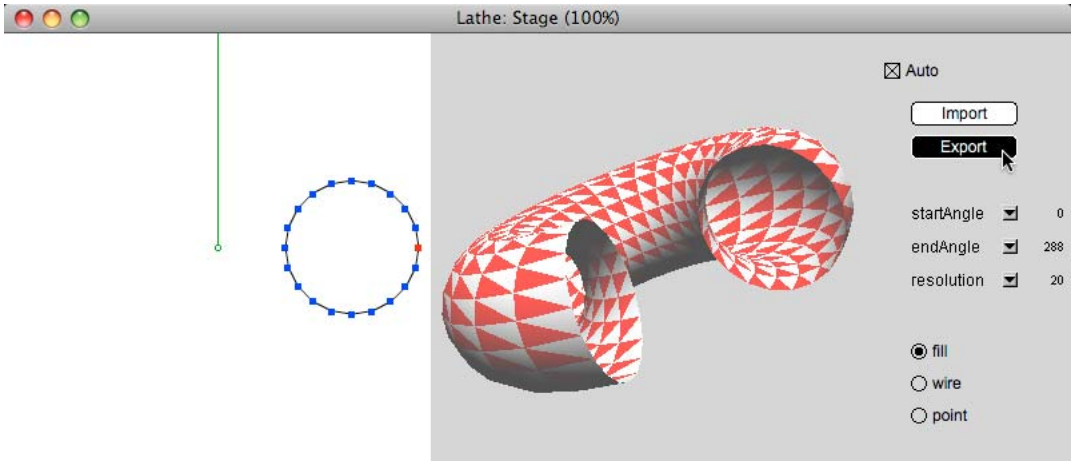
You can also provide information concerning:

- The direction of the normal to the surface at each vertex point.
- The color of each vertex in each face.
- How textures will map to the surface of the mesh.

For each face, you can define a shader. For each shader that your mesh resource uses, the resulting model will have an additional entry in its `shaderList` property.

Examples

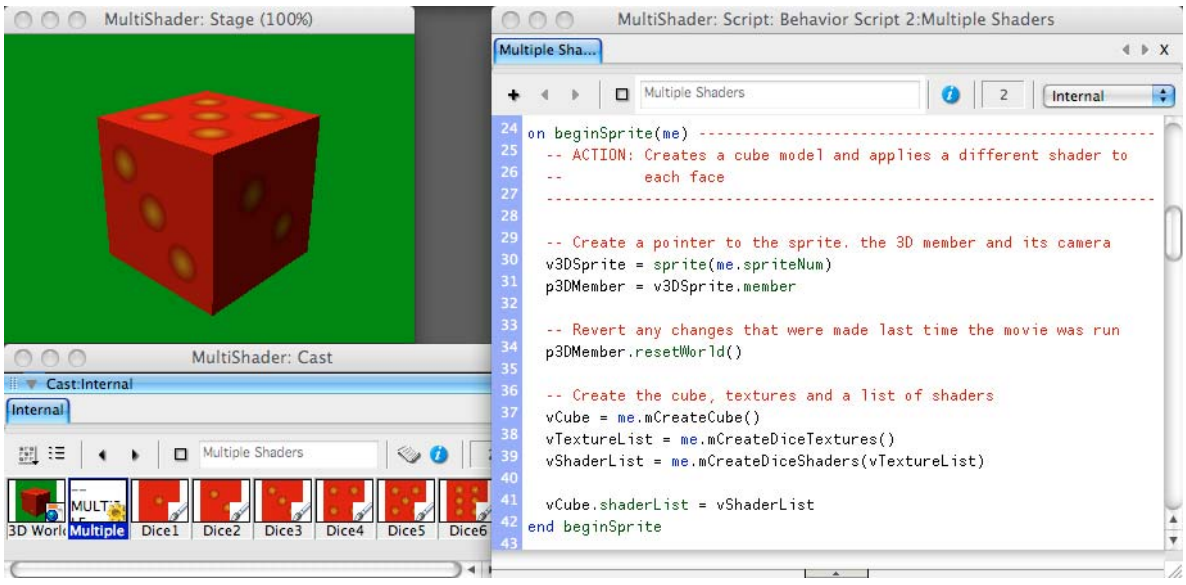
To see the process of creating a mesh resource in real time, download the movie Lathe.dir, and launch it.



With the Lathe.dir movie, you can define a 2D template and extrude it in a circle around an axis.

Click on the 2D pane on the left to create a new vertex point in a 2D plane, or drag an existing point to a new position. Press the Delete key to remove a vertex point. You can export a script that you can use create a new model with the chosen template and settings.

To see an example of a mesh resource with multiple shaders, download the movie Frame.dir, , and launch it.



Frame.dir creates a set of mesh resources with two shaders: one for the frame, one for the picture

Uses of mesh resources

Mesh resource can be used:

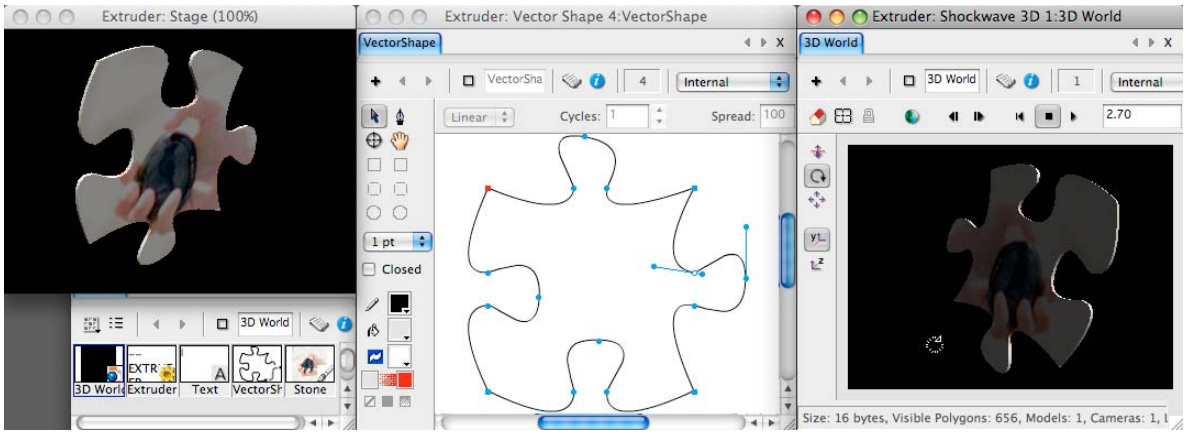
- To create a terrain using a bump map
- To create the walls of a building from a floor plan

- To create a simple collision wall around more complex geometry
- To create objects with geometry that you can modify on the fly, or for any other purpose where pre-defined models cannot be used.

Extruder resources

The extrude3d() function is designed to create an `#extruder` resource from a text cast member. However, you can change the vertexList property of an `#extruder` resource after it has been created.

To see an example of this, download the movie Extruder.dir, , and launch it.



You can use a vectorShape member to define the shape of an #extruder model resource

Note: *The 2D co-ordinates of a vectorShape member are measured rightwards and downwards, where the 3D co-ordinates of a model resource are measured rightwards, upwards, and forwards. As a result, the shape of an* `#extruder` *resource appears to be flipped vertically. An* `#extruder` *resource has only one mesh. The same shader is applied to both front and back, and it is smeared along the sides.*

You can modify the smoothness, tunnelDepth, , and displayFaces properties of an `#extruder` resource.

Particle emitters

Particle systems are different from the other primitives. Instead of being shapes, they create cascades of moving particles. You can use these to simulate dust, clouds, rain, smoke, fire, fireworks, and many other randomly animated systems of particles.

A particle is simply a square plane that turns to face the camera. You can place a texture on this plane. You can use transparency in the texture to give it a specific (non-square) shape.

The emitter can be a point, a line, or an area defined by a set of four points in space. The particles can be made to change size, color, size, and blend over time. You can set the maximum and minimum speed. You can make the particles start in a given direction, follow a path, drift with a wind, and fall due to gravity. You can make an emitter create a single burst of particles, like a firework, or produce particles continuously. You can move the emitter about in 3D space, without moving the model itself.

Experimentation is the best way to find which settings give the results that you are looking for.

Often it helps to have several particle systems with slightly different characteristics working together. You can find an example that uses three particle systems and a total of 260 particles to create simulate a fire in the movie Particles.dir.

Smoke can be simulated by a number of slowly drifting and fading particles

Like other primitives, the particle system is created from a model resource and a model. All models that use the same model resource will show the same particles in the same random states.

3D behaviors

Director includes a library of behaviors that lets you build and control a 3D environment without any knowledge of Lingo or JavaScript syntax. Although scripting is still required for complex projects, you can build simple 3D movies with behaviors alone.

Behavior types

Director provides two types of 3D behaviors: trigger and action. Action behaviors are divided into three types, which are defined in the following table.

Type	**Function**
Trigger behavior	A behavior that sends an event, such as a mouse click, to an action behavior
Local **action** behavior	A behavior that is attached to a particular sprite and that can accept triggers only from that sprite
Public **action** behavior	A behavior that can be triggered by any sprite
Independent **action** behavior	A behavior that needs no trigger

In previous versions of Director, the trigger instruction had to be included as a handler, such as `on mouseDown`, inside the behavior. The trigger behavior type makes it easier to reuse action behaviors in different ways with different triggers. These behaviors can be used with any 3D cast member.

Using the 3D behavior library

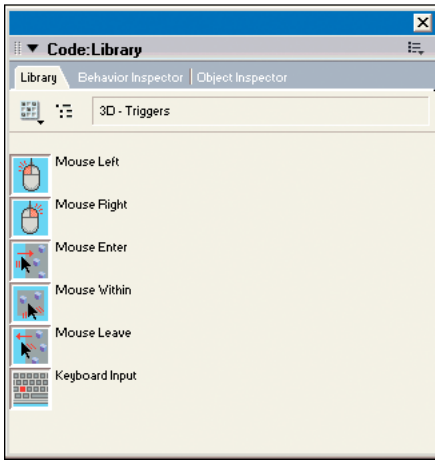
All 3D behaviors are listed in the Behavior Library. The Behavior Library is divided into two sub libraries: actions and triggers.

1 Click the Library Palette button on the Director toolbar.

2 Click the Library List button and select 3D.

3 Select Triggers from the 3D submenu.

The trigger behaviors appear.



3D trigger behaviors

The following table describes the available triggers:

Name	Description
Mouse Left	Triggers action when the user presses, holds down, or releases the left mouse button (WindowsÂ®) or the mouse button (MacÂ®).
Mouse Right	Triggers action when user presses, holds down, or releases the right mouse button. To use on the Mac, you must set the `emulateMultibuttonMouse` property to true; then Control-click is interpreted as a right-click. To use this with ShockwaveÂ® Player, you must disable the context menu and pass right-clicks through to the player.
Mouse Within	Triggers an action when the pointer is inside a sprite.
Mouse Enter	Triggers an action when the pointer enters a sprite.
Mouse Leave	Triggers an action when the pointer leaves a sprite.
Keyboard Input	Lets the author specify a given key as a trigger.

You can add modifier keys to any trigger. By doing so, a given trigger can launch two actions, depending on whether the modifier key is pressed. For example, you can Mouse Left and Mouse Left+Shift as separate triggers.

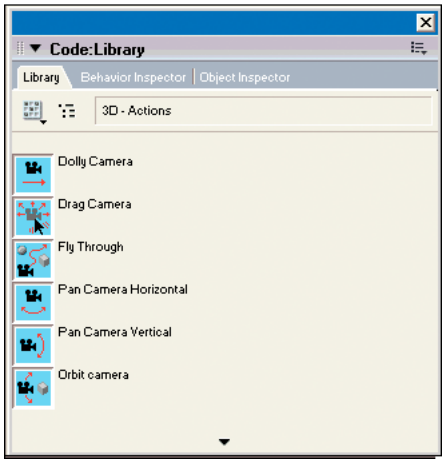
View 3D action behaviors

1 Click the Library Palette button on the Director toolbar.

2 Click the Library List button and select 3D.

3 Select Actions from the 3D submenu.

The action behaviors appear, as shown in the following figure:



3D action behaviors

Local actions

When you attach a local action to a sprite, that action responds only to a trigger that is attached to that same sprite. The following table describes the available local actions:

Name	Effect	Description
Create Box	Primitive	Adds a box to the 3D world each time the trigger action occurs. The author can set the dimensions and texture.
Create Particle System	Primitive	Creates a particle system whenever the trigger is activated. The user can set the number of particles; the life span of each particle; the starting and finishing color of particles; and the angle, speed, and distribution of particles on emission. The user can also set gravity and wind effects along any axis.
Create Sphere	Primitive	Adds a sphere to the 3D world each time the trigger action occurs. The author can set the diameter and texture.
Drag Camera	Camera	Provides full camera control, including panning (changing the direction in which the camera is pointing), dollying (moving), and rotating, through a single behavior. Use separate mouse triggers for panning, zooming, and rotating.
Drag Model	Model	Lets users move a model in any direction by dragging it with the mouse.

Name	Effect	Description
Drag Model to Rotate	Model	Lets you specify an axis or pair of axes around which you can rotate a model by dragging the model with the mouse.
Fly Through	Camera	Simulates flying through the 3D world with a camera. Accepts separate triggers for forward and reverse travel and for stopping.
Click Model Go to Marker	Model	Moves the playhead to a marker in the Score when a model is clicked.
Orbit Camera	Camera	Circles the camera around a model.
Play Animation	Model	Plays a preexisting animation when the model is clicked. This behavior cannot be used with 3D text.

Public actions

As with local actions, you can add public actions to a movie by attaching them to any 3D sprite. Unlike local actions, public actions are triggered whether the trigger is attached to the same sprite as the action or to any other sprite. Public actions use the same triggers as local actions. The following table describes available public actions:

Name	Effect	Description
Dolly Camera	Camera	Dollies the camera into or out of the 3D scene by a specified amount each time the trigger action occurs. Dollying in and dollying out require separate triggers.
Generic Do	Custom	Lets you use the standard triggers to launch custom handlers or execute specific script methods. Requires knowledge of scripting in Director.
Pan Camera Horizontal	Camera	Pans along the horizontal axis by a specified number of degrees each time the trigger action occurs. Panning left and panning right require separate triggers.
Pan Camera Vertical	Camera	Pans along the vertical axis (up and down) by a specified number of degrees each time the trigger action occurs. Panning up and panning down require separate triggers.
Reset Camera	Camera	Resets the camera to its initial location and orientation when the trigger action occurs.
Rotate Camera	Camera	Rotates the camera around the z-axis by a specified number of degrees each time its trigger is activated. This makes the 3D scene appear to rotate and turn upside down.
Toggle Redraw	Drawing	Toggles the redraw mode on or off. Turning redraw off produces visible trails as a model moves through space. Turning redraw on causes the trails to disappear.

Independent actions

Independent actions do not require triggers. The following table describes the available independent actions:

Name	Effect	Description
Automatic Model Rotation	Motion	Automatically rotates a model around a given axis and continues rotating it while the movie plays. To rotate the model around multiple axes, attach multiple instances of the behavior to the sprite, and select the desired axis for each one.
Level of Detail	Model	Enables the level of detail (LOD) modifier for the model. Dynamically lowers the number of polygons used to render the model as its distance from the camera increases. Reduces demands on the CPU.
Model Rollover Cursor	Model	Changes the mouse pointer to the pointer of your choice when the mouse rolls over the given model.
Show Axis	Debugging	Establishes red, green, and blue lines along the x-, y-, and z-axes, respectively, letting you see them in the 3D scene.
Subdivision Surfaces	Model	Enables the subdivision surfaces (SDS) modifier for the given model, which synthesizes additional detail to smooth out curves as the model's distance from the camera decreases.
Toon	Model	Enables the toon modifier, which renders the model in a cartoon style, with a reduced number of colors and distinct boundaries. The user can set the toon style precisely, by selecting the number of colors, line color, brightness, and darkness of highlights and shadows, and anti-aliasing.

Applying 3D behaviors

Apply 3D behaviors in the same way as you apply standard behaviors in Director. You can attach as many behaviors to a sprite as needed, but each behavior that requires a trigger must have a unique trigger to activate it.

1. Open the Library palette.
2. Open the 3D library.
3. Attach an action behavior to the sprite, either on the Stage or in the Score. The Parameters dialog box appears. Use it to control the behavior.
4. Specify options in the Parameters dialog box, and click OK.
5. For local behaviors, attach a trigger behavior to the same sprite. For public behaviors, attach a trigger behavior to a specific sprite.

   The Parameters dialog box appears. Use it to control when the trigger should work; what modifier keys, if any, are associated with the trigger; and to which sprite group the trigger is assigned. For more information about groups, see "About behavior groups" on page 79.
6. Specify the options in the Parameters dialog box, and click OK.

About behavior groups

The Parameters dialog boxes of the local and public action behaviors give you the option to assign the behavior to a group of behaviors. Groups let a single trigger initiate actions across multiple sprites. To establish a group, select a name for the group, and enter that name in the Parameters dialog box of each behavior that you attach to the sprites in the group.

Note: *A behavior group is not the same as a group node inside a 3D cast member.*

Because the triggers are sent to the group name rather than to a specific sprite number, there are no reference changes to update when a sprite moves from one Score channel to another.

Programming issues

A Shockwave 3D movie makes intensive use of memory and processing time. Before you start developing your own 3D worlds and writing the scripts that will make them interactive, there are a number of programming issues that you need to be aware of.

- The 3D world can take some time to load into memory. Certain operations are not possible until it has loaded. For more information, see "Preload requirements" on page 80.
- If you plan to use JavaScript syntax to control your 3D world, you need to understand how to refer to the properties of 3D objects. See "Lingo and JavaScript access to 3D objects" on page 82 for more details.
- All nodes share the same namespace. You cannot have a group and a model (for example) with the same name. This is treated in more detail in "3D namespace" on page 85.

Preload requirements

During authoring and in a projector, the media of a 3D cast member can be read from the local hard disk. This usually happens almost instantaneously.

When you play back a 3D cast member in a Shockwave movie in a browser, the media of the 3D cast member are initially stored on a remote server. Before the Shockwave movie can display any of the models, textures, and other objects in the 3D world, a local copy of the media must be transferred to the end-user's computer.

If the local copy were written to the local hard disk, the process would be called *downloading*. However, for security reasons, the local copy of a Shockwave movie is stored only in the end-user's Random Access Memory (RAM). This process is known as *preloading*.

Preload

There are two ways that you can handle playback of a 3D world from a remote server:

- **Preloading**: You can preload all the data for the 3D cast member before starting playback. The user will not be able to start interacting with the 3D world until a local copy of all data from the remote server has been created in the computer's RAM.
- **Streaming**: You can set a 3D cast member so that it starts to display objects as soon as the data for those objects have been preloaded. This means that parts of the 3D world appear very quickly, and the user can start interacting with these parts before the entire world is available.

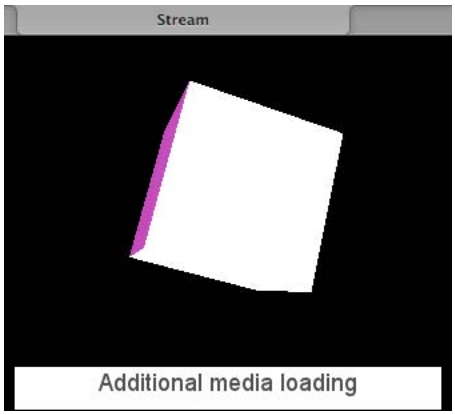
To determine which of these options is used, you can set the 3D cast member's preload property.

Internal and external 3D cast members

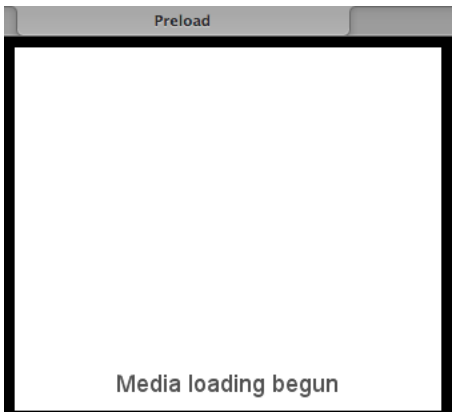
Internal 3D cast members always have their `preload` property set to `TRUE`. You cannot stream internal 3D cast members. External cast members have their `preload` property set to `FALSE` by default.

To see the difference between preloading and streaming an external W3D file, compare the following two Shockwave movies:

- preload.html
- stream.html



The same external W3D file can be set to preload or to stream



The same external W3D file can be set to preload or to stream

Both these movies to the same external.w3d file. The only difference between the movies is that in stream.html, the `preload` property of the linked Shockwave 3D cast member is set to `FALSE`. As a result, the 3D media starts to play back as soon as it can, even before all the textures have finished streaming into your computer's RAM.

You can find the original files for these movies here. You will notice that the CreateExternalW3D.dir movie wastefully creates too many textures, and only applies the last textures that it creates to the shaders for the cube. This ensures that even on a fast connection, you can see the cube with empty shaders when streaming starts.

State

A 3D cast member has a state property. This indicates how far the streaming or preloading process has progressed. If you are using streaming for an external W3D file, you should ensure that the `state` property has the appropriate value before using Lingo or JavaScript syntax to control the objects in the 3D world.

Lingo and JavaScript access to 3D objects

Lingo is the original proprietary programming language for Director movies. Lingo access to 3D objects and their properties is often very flexible.

You can also use JavaScript syntax to write your scripts. However, in order to respect the specifications of JavaScript, some expressions are longer to write in JavaScript than in Lingo. Certain functions and methods that are available in Lingo are not available in JavaScript. Other functions are formulated very differently in the two languages.

To realize the full potential of Shockwave 3D, use Lingo wherever you can.

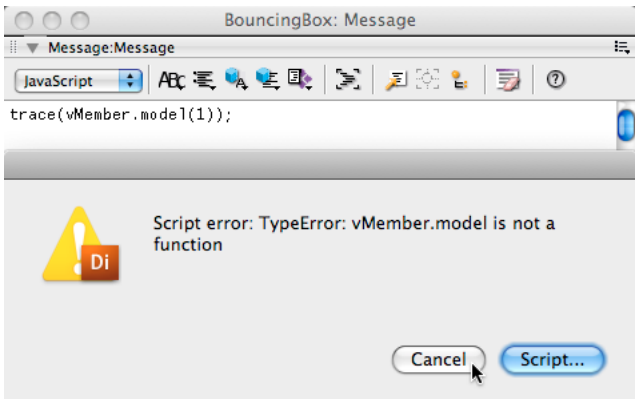
The examples in this article assume that the 3D cast member "3D World" contains a model created from a `#box` primitive, which has 6 meshes.

Using getPropRef() and getProp()

Compare these two examples:

```
-- Lingo
vMember = member("3D World")
put vMember.model(1).resource
-- box("Box")
// JavaScript
vMember = member("3D World");
<(member 1 of castLib 1)>
trace (vMember.getPropRef("model", 1).getProp("resource"));
// <box("Box")>
```

The following incorrect JavaScript statement attempts to mimic the Lingo syntax. It will throw a script error.



Not all functions in Lingo are implemented as functions in JavaScript syntax

Using symbol()

You cannot use the hash (#) character to define a symbol in JavaScript syntax. You need to use the `symbol()` function instead.

```
-- Lingo
vShader = vMember.newShader("Mesh 1 shader", #standard)
// JavaScript
vShader = vMember.newShader("Mesh 1 shader", symbol("standard"));
<shader("Mesh 1 shader")>
```

Using setProp()

To change the value of the property of an object with JavaScript syntax, there are some case where you must use the `setProp()` command. These two examples continue from the examples above:

```
-- Lingo
vMember.model(1).shader = vShader
put vMember.model(1).shaderList
-- [shader("Mesh 1 shader"), shader("DefaultShader"), shader("DefaultShader"),
shader("DefaultShader"), shader("DefaultShader"), shader("DefaultShader")]
// JavaScript
vMember.getPropRef("model", 1).setProp("shader", vShader);
trace(vMember.getPropRef("model", 1).getProp("shaderList"))
// <[shader("Mesh 1 shader"), shader("DefaultShader"), shader("DefaultShader"),
shader("DefaultShader"), shader("DefaultShader"), shader("DefaultShader")]>
```

Cases where getProp and setProp are not required

Not all objects require the use of `getProp` and `setProp` in JavaScript. For example:

```
// JavaScript
sprite(1).camera.getPropRef("overlay", 1).scale = 1.5;
trace(sprite(1).camera.getPropRef("overlay", 1).scale);
// 1.5
```

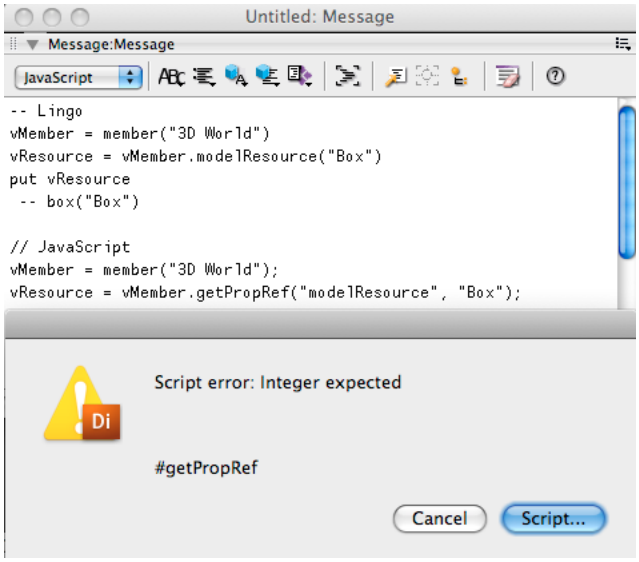
Functions and methods missing from JavaScript syntax

Not all 3D features have been fully implemented in JavaScript syntax. For example, it is not possible to obtain a pointer to an object by using its name. This is possible in Lingo:

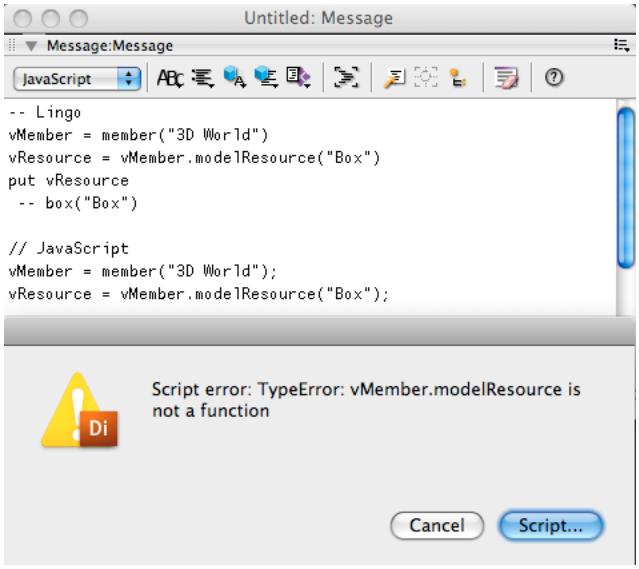
```
-- Lingo
vResource = vMember.modelResource("Box")
put vResource
 -- box("Box")
```

However, as the examples below show, it is only possible to access the object by its index number through JavaScript. Access by name will lead to errors.

```
// JavaScript
vMember = member("3D World");
vResource = vMember.getPropRef("modelResource", 2); // integer
<box("Box")>
trace(vResource);
// <box("Box")>
```



Attempting to access an object by its name in JavaScript will lead to script errors



Attempting to access an object by its name in JavaScript will lead to script errors

Using count to count objects

Compare the following examples

```
-- Lingo
put sprite(1).camera.overlay.count
-- 2
// JavaScript
trace(sprite(1).camera.count("overlay"));
// 2
```

3D namespace

A Shockwave 3D cast member can contain objects of the following types:

- Nodes
  - Cameras
  - Groups
  - Lights
  - Models
- Model Resources
- Motions
- Textures
- Shaders

Within any of these object types, you may not have two objects with the same name. For example, you cannot have a light named “Blue” and a model named “Blue” in the same 3D cast member, because both lights and models are nodes. You can have a light named “Blue” and a shader named “Blue” in the same 3D cast member, because lights and shaders are different types of objects.

Your third-party 3D design software may allow you to use a more flexible naming system. If you find that some objects are missing when you export to W3D, it is worth checking if there are duplicate names in the original source files. See “Export issues” on page 68 for more details.

Note: *You may find that the names of objects in an exported W3D file have a different case than they had in the 3D design software that you used to create them objects. A model named “MyModel” in your 3D design software may be exported as “mymodel”.*

When you use commands such as aNode.cloneDeep(), member3D.cloneModelFromCastMember(), or member3D.loadFile(), new objects may be added to the 3D cast member. These objects will be renamed during the process, in order to avoid naming clashes.

The `cloneDeep()` and `loadFile()` commands will add “`_copyX`" (where X is an integer) to the end of the names of renamed objects. The `cloneModelFromCastMember()` command will add "`_cloneX`" (where X is an integer) to the end of the names of renamed objects. You may find that the new names have all been converted to lower case characters.

Ensuring that a name is unique

Below are three handlers that you can use to make sure that a given name is unique in the namespace in which it is to be used. The first is written for textures. You can modify it for shaders, modelResources or motions, by replacing the string “texture” with the appropriate object name everywhere.

```
on GetUniqueTextureName(a3DMember, aName)
 vTexture = a3DMember.texture(aName)
 if not vTexture then
 return aName
 end if
 vIndex = 0
 repeat while TRUE
 vIndex = vIndex - 1
 vUniqueName = aName&vIndex
 vTexture = a3DMember.texture(vUniqueName)
 if not vTexture then
 return vUniqueName
 end if
 end repeat
end GetUniqueTextureName
```

Example usage:

```
-- Lingo
vName = GetUniqueTextureName(member("3D"), "George")
put vName
 -- "George-2"
vTexture = member("3D").newTexture(vName)
//JavaScript
vName = GetUniqueTextureName(member("3D"), "George");
George-2
vTexture = member("3D").newTexture(vName)
<texture("George-2")>
```

The following handlers ensures that no node (camera, group, light or model) has the name that you are planning to use:

```
on GetUniqueNodeName(a3DMember, aName)
 if NodeNameIsUnique(a3DMember, aName) then
 return aName
 end if
 vIndex = 0
 repeat while TRUE
 vIndex = vIndex - 1
 vUniqueName = aName&vIndex
 if NodeNameIsUnique(a3DMember, vUniqueName) then
 return vUniqueName
 end if
 end repeat
end GetUniqueNodeName
on NodeNameIsUnique(a3DMember, aName)
 vNode = a3DMember.camera(aName)
 if not vNode then
 vNode = a3DMember.group(aName)
 if not vNode then
  vNode = a3DMember.light(aName)
  if not vNode then
  vNode = a3DMember.model(aName)
  if not vNode then
   return TRUE
  end if
  end if
 end if
 end if
 return FALSE
end NodeNameIsUnique
```

Chapter 3: 3D: Controlling appearance

A 3D cast member contains the objects that provide access to 3D functionality. This chapter deals with the features that create a graphic representation of the 3D world.

For information on how to create action and interactions between objects, see “3D: Controlling action” on page 202.

Nodes

All objects in a 3D world are based on a basic object known as a node. The simplest form of a node in a 3D world is a Group object. A Group object is essentially the most basic node. All other objects in a 3D world are based on a Group object, which means that the other objects inherit the functionality of a Group object in addition to containing functionality that is specific to those objects.

- Groups are the most basic kind of nodes. All other nodes share all the methods and properties of a group. See “Groups” on page 88 for more information.
- You can link nodes together in parent-child relationships. The two nodes will then move together as if an invisible bar linked them. For example, if you have a model called "Head", and a model called "Hat". You can make the hat model the child of a head model. When the head moves, the hat will move with it. See “Node hierarchy” on page 91 for more details
- Every 3D cast member contains a group named "World". A model that is not a child of group("World") will not be visible to a camera that is a child of group("World"). See “Group("World")” on page 93 for more details.
- Each node in the world has a spatial relationship to all others. This relationship is defined by a combination of the node's transform and its position in a parent-child hierarchy. See “Frame of reference” on page 97 for more details on how to move nodes relative to each other.
- All nodes have a property list attached to them, called userData. See “userData” on page 99 for some ideas on how to benefit from this feature.
- You can make copies of any node, and optionally of all its children and the graphic objects that it uses. See “Cloning” on page 101 for more details.
- Certain actions, such as playing a motion or the collision between two objects, occur at specific times and generate events which can be handled by scripts registered for the purpose. See “Events” on page 355 for more details.

Groups

A group is the most basic node. It is merely a point in space that is represented by a transform. You can assign children and parents to this node in order to group models, lights, cameras, or other groups. Any translation, rotation, or scale applied to the parent group affects all its children. Removing the group from the world removes all its children.

Accessing the groups in a 3D member

```
-- Lingo
put member("3D").group.count
-- 1
// JavaScript
trace(member("3D").count("group"));
// 1
```

To access a particular group, you can use its name (Lingo only) or its index number.

```
-- Lingo
put member("3D").group[1]
-- group("World")
put member("3D").group(1)
-- group("World")
put member("3D").group("hug")
-- group("Hug")
// JavaScript
trace(member("3D").getPropRef("group", 1));
// <group("World")>
```

If a group with the given name or index number does not exist, Director will return VOID (Lingo) or undefined (JavaScript); no error occurs.

Creating a group

Use the member3D.newGroup() function to create a new group with a given unique name.

```
-- Lingo syntax
vGroup = member("3D").newGroup("Musicians")
put vGroup
-- group("Musicians")
// JavaScript
vGroup = member("3D").newGroup("Musicians");
<group("Musicians")>
```

Ensure that the name is unique, which means that the name is not already used by any other node (model, light, group or camera). If you try to create a new group with the same name as an existing node, a script error occurs: "Object with duplicate name already exists".

See "3D namespace" on page 85 for more details and a script that ensures that you have a unique name for the group that you are about to create.

Deleting a group

To delete a group, use the member3D.deleteGroup() function. You can identify the group either by its name or its index number. Deleting a group may change the index number of other groups. If you attempt to delete a non-existent group, no error occurs.

```
-- Lingo syntax
put member("3D").deleteGroup("Disbanded")
-- 1
put member("3D").deleteGroup("Disbanded")
-- 0
put member("3D").deleteGroup(2)
-- 1
// JavaScript syntax
trace(member("3D").deleteGroup("Disbanded"));
// 1
member("3D").deleteGroup("Disbanded");
0
member("3D").deleteGroup(2);
1
```

Using a group

Like every other node, a group has a transform property, which allows it to be positioned in 3D space, rotated, and scaled. It can also be placed in a parent-child relationship with other nodes (see "Node hierarchy" on page 91 for more details).

These two features allow you to use groups in a variety of ways.

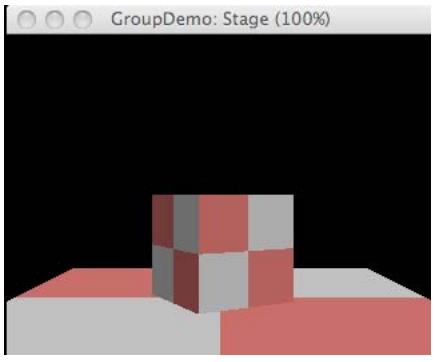
- As parent of a group of nodes
- Scaling
- To provide an anchor point for a model
- To provide an extra degree of freedom. See "Looking around" on page 222 for an example where the parent group for a camera determines the position of the camera and the axis of the camera's movements. This allows the camera to tilt without affecting its movement in space.

Examples

The following Lingo code creates a "Box model" and a "Plane model" from primitives with default values. It makes each the child of its own group and modifies the transform of the group. The result is a box which appears to stand at the surface of the plane.

```
v3DMember = member("3D")
vName = "Plane"
vResource = v3DMember.newModelResource(vName, #plane)
vModel = v3DMember.newModel(vName&" model", vResource) vModel.rotate(-90, 0, 0)
pPlane = v3DMember.newGroup(vName)
pPlane.addChild(vModel, #preserveWorld)
pPlane.scale(200)
vName = "Box"
vResource = v3DMember.newModelResource(vName, #box)
vModel = v3DMember.newModel(vName&" model", vResource)
vModel.translate(0, vResource.height / 2, 0)
pBox = v3DMember.newGroup(vName)
pBox.addChild(vModel, #preserveWorld)
pBox.rotate(0, 30, 0)
```

To find the original code, download the movie GroupDemo.dir.



Using a group to provide a model with a more appropriate scale or origin point.

The following Lingo code creates four models named "Wheel_1" to "Wheel_4" at the corners of a rectangle. It makes each wheel the child of a group called "Chassis", and then scales and rotates the group.

```
v3DMember= member("3D")
pChassis = v3DMember.newGroup("Chassis")
vName = "Wheel_"
vResource = v3DMember.newModelResource(vName, #cylinder)
vResource.height = 10
vRadius = vResource.topRadius
repeat with ii = 1 to 4
vModel = v3DMember.newModel(vName&ii, vResource)
vModel.rotate(0, 0, 90)
vX = (ii mod 2) * 50 - 25
vZ = (ii > 2) * 70 - 35
vModel.translate(vX, vRadius, vZ, #world)
pChassis.addChild(vModel, #preserveWorld)
end repeat
pChassis.scale(0.20)
pChassis.rotate(0, 30, 0)
```

Node hierarchy

Each node in a 3D world may have one parent and any number of children. By default, each new model is set as a child of the group("World"). See "Group("World")" on page 93 for more details.

Each child node inherits the transform of its parent node as a frame of reference. The final position, rotation, and scale of a model in the world depends on the transform of each node in its hierarchy of parents. See "Frame of reference" on page 97 for more details.

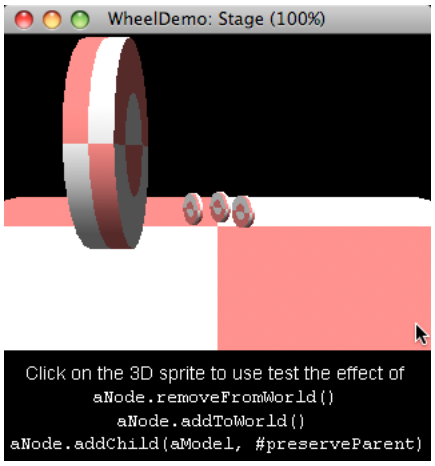
The primary benefit of these parent-child relationships is that they make it easier to move complex models around in the 3D world and to have the component parts of those models move together in the proper way. In the example of a car, if the wheels of the car are defined as children of the car model, then moving the car will cause the wheels to be moved with the car in the expected manner. If no parent-child relationship is defined between the car and the wheels, moving only the car causes the wheels to be left behind in their original position in the world.

Scripting terms to use with parent-child relationships

You can use these scripting terms when you are working with parent-child relationships:

- node.addChild()
- node.child.count (Lingo) or aNode.count("child") (JavaScript)
- node.child() (Lingo) or node.getPropRef("child", aIndex) (JavaScript)
- node.parent

To test these terms, download and launch the movie WheelDemo.dir.



Clicking the 3D sprite changes the parent of model("Wheel_4")

What to preserve with addChild()

When you set a node to be the child of another node, you can determine whether the child node remains in the same place in the world after its adoption, or whether it remains in the same relative position to its parent. In the latter case, the position, rotation, and scale of the child node may change with respect to the world. On `beginSprite()`, the Wheel Demo behavior uses a code similar to the following:

```
-- Lingo syntax
vGroup = member("3D").group("Chassis")
vModel = member("3D").model("Wheel_4")
vGroup.addChild(vModel, #preserveWorld)
-- JavaScript syntax
vGroup = member("3D").getPropRef("group", 3);
vModel = member("3D").getPropRef("model", 5);
vGroup.addChild(vModel, symbol("preserveWorld"));
```

This code alters the transform of the child model so that it remains in the same position relative to the world. Click twice on the 3D sprite. The second time you click, a code similar to the following is executed:

```
-- Lingo syntax
vGroup.addChild(vModel, #preserveParent)
// JavaScript syntax
vGroup.addChild(vModel, symbol("preserveParent"));
```

This code changes the parent of the Wheel_4 model, but does not alter its transform. The model's position, rotation, and scale relative to the world change.

Testing in the Message window

To test the other scripting terms, try typing these commands in the Message window:

```
-- Lingo syntax
vGroup = member("3D").group("Chassis")
put vGroup.child.count
-- 4
put vGroup.child(1)
-- model("Wheel_1")
put vGroup.child("Wheel_2")
-- model("Wheel_2")
put member("3D").model("Wheel_3").parent
-- group("Chassis")
// JavaScript syntax
vGroup = member("3D").getPropRef("group", 3);
<group("Chassis")>
trace(vGroup.count("child"));
// 4
trace(vGroup.getPropRef("child", 1));
// <model("Wheel_1")>
trace(member("3D").getPropRef("model", 5).parent);
// <group("Chassis")>
trace(member("3D").getPropRef("model", 5).getProp("parent"));
// <group("Chassis")>
```

Moving a node without reference to its parent

Set aNode.worldPosition to a position vector. This will alter the position property of the child node's transform. The value for the new position will depend on the transforms of all the child node's parents. You can alter the world rotation or scale of a child node in this way. See "Frame of reference" on page 97 for more details.

Group("World")

Each 3D cast member contains a group object called "World", which may contain a tree-like parent-child hierarchy of nodes, such as models, groups, lights, and cameras. Nodes that have group("World") as an ancestor are rendered by the default camera.

A cast member may also contain nodes that do not have 'world' as an ancestor, such as nodes with a parent property set to VOID. To improve performance, you can remove any models from the world that are not currently visible. For example, while the user is observing a scene inside a room, all models outside the room can be temporarily removed from the world.

Nodes that are not children of the group("World") are not normally rendered. However, you can set the rootNode of a camera to a node that is not in the world. This technique is useful for creating a sky box, for example. See "Sky box" on page 108 for more details.

Scripting terms to use with reference to group("World")

- node.isInWorld() is TRUE if group("World") is a parent or ancestor of the node, FALSE if not.
- node.addToWorld() sets the parent of the node to group("World") if node.isInWorld() is FALSE. Does not do anything if group("World") is already a parent or ancestor of the node.
- "Sky box" on page 108node.removeFromWorld() sets the parent of the node to VOID.
- node.getWorldTransform() returns the transform that need to be applied to the node to display at in its current position. This property returns rotation and scale if the parent is group("World").
- node.worldPosition gets and sets the position of the node in world space. The value of node.transform.position may be different from its worldPosition if the node's parent is not group("World").

- node.boundingSphere gives the center and radius of the smallest sphere that completely encloses the node and all its children. The values are in world co-ordinates.
- node.pointAt() turns the node so that its `pointAtOrientation` is facing towards the given node or position, given in world coordinates.
- node.pointAtOrientation gets and sets the directions that the node uses for its forward and upwards directions.

To test these terms, download and launch the movie WheelDemo.dir.

removeFromWorld() and addWorld()

When you click on the 3D sprite, the Wheel Demo behavior uses a code similar to the following:

```
-- Lingo syntax
vModel = member("3D").model("Wheel_4")
put vModel.parent
-- group("Chassis")
vModel.removeFromWorld()
put vModel.parent
-- <Void>
vModel.addToWorld()
put vModel.parent
-- group("World")
// JavaScript syntax
vModel = member("3D"     ).getPropRef("model", 5);
trace(vModel.parent);
/ /   < g roup("Chassis")>
 v M o d el.remove F r o m World();
trace(vM     odel.parent);
// undef     ined
vModel.addToWorld();
trace(vModel.parent);
// <group("World")>
```

When a node is removed from the world, its transform property remains unchanged. If you later reset its parent to its original parent, it returns to exactly the same position in the world.

If you use node.addToWorld(), you set the parent of the node to group("World"). If this was not the original parent, the node may appear in a new position. See "Node hierarchy" on page 91 for details on how to set the parent of a node to a node other than group("World"). See "Returning a node to its original place" on page 96 for an alternative solution.

Testing in the Message window

Try executing the following commands in the Message window.

Note: *The layout of the transform data is modified to make it more readable.*

```
-- Lingo syntax
vModel = member("3D").model("Wheel_4")
put vModel.parent
-- group("Chassis")
put vModel.isInWorld()
-- 1
vModel.addToWorld()
put vModel.parent
-- group("Chassis")
put vModel.getWorldTransform()
-- transform(-0.00000,  0.20000, 0.00000, 0.00000,
            -0.17321, -0.00000, 0.10000, 0.00000,
            0.10000,  0.00000, 0.17321, 0.00000,
            -0.83013,  5.00000, 8.56218, 1.00000)
put vModel.worldPosition
-- vector( -0.8301, 5.0000, 8.5622 )
vModel.removeFromWorld()
put vModel.isInWorld()
-- 0
put vModel.worldPosition
-- vector( -0.8301, 5.0000, 8.5622 )
vModel.addToWorld()
put vModel.parent
-- group("World")
put vModel.worldPosition
-- vector( -25.0000, 25.0000, 35.0000 )
// JavaScript syntax
vModel = member("3D").getPropRef("model", 5);
trace(vModel.parent);
// <group("Chassis")>
trace(vModel.isInWorld());
// 1
vModel.addToWorld();
1
trace(vModel.getWorldTransform());
// <transform(-0.00000,  0.20000, 0.00000, 0.00000,
        -0.17321, -0.00000, 0.10000, 0.00000,
        0.10000,  0.00000, 0.17321, 0.00000,
        -0.83013,  5.00000, 8.56218, 1.00000)>
trace(vModel.worldPosition);
// <vector( -0.8301, 5.0000, 8.5622 )>
vModel.removeFromWorld();
1
trace(vModel.isInWorld());
// 0
trace(vModel.worldPosition);
// <vector( -0.8301, 5.0000, 8.5622 )>
vModel.addToWorld();
1
trace(vModel.parent);
// <group("World")>
trace(vModel.worldPosition);
// <vector( -25.0000, 25.0000, 35.0000 )>
```

Points to note:

- If you use `addToWorld()` on a node that is already in the world, nothing happens.

- If you use `addToWorld()` on a node that is not in the world, its parent is set to group("World").
- Both `getWorldTransform()` and `worldPosition` retain their values after a node has been removed from the world.
- When `addToWorld()` is used, the node's transform does not change, but the `worldPosition` and the result return by `getWorldTransform()` are updated if the node's original parent was rotated or scaled, or was not at the center of the world.

Returning a node to its original place

Even after a node has been removed from the world, its `getWorldTransform()` returns the value that was last valid when it was still in the world. When you use `node.addToWorld()`, the value returned by `getWorldTransform()` is reset. If you store a copy of the model's world transform, you can return the node to its original location after using `addToWorld()`.

Here is an example to try in the Message window after relaunching the WheelDemo.dir movie.

```
-- Lingo syntax
vModel = member("3D").model("Wheel_4")
vModel.removeFromWorld()
vTransform = vModel.getWorldTransform()
vModel.addToWorld()
vModel.transform = vTransform
// JavaScript syntax
vModel = member("3D").getPropRef("model", 5);
vModel.removeFromWorld();
vTransform = vModel.getWorldTransform();
vModel.addToWorld();
vModel.transform = vTransform;
```

Note that, to achieve the same effect without actually removing the node from the world, you can use:

```
-- Lingo syntax
v3DMember = member("3D")
vModel = v3DMember.model("Wheel_4")
v3DMember.group(1).addChild(vModel, #preserveWorld)
// JavaScript syntax
v3DMember = member("3D");
vModel = v3DMember.getPropRef("model", 5);
v3DMember.getPropRef("group", 1).addChild(vModel, symbol("preserveWorld"));
```

boundingSphere

The boundingSphere of a node is a list that gives the center and radius of the smallest sphere that completely encloses the node and all its children. The values are in world coordinates.

```
-- Lingo syntax
put member("3D").model("Wheel_4").boundingSphere
// JavaScript syntax
trace( member("3D").getPropRef("model", 5).boundingSphere);
```

The output depends on whether Wheel_4 is a child of group("Chassis")

```
-- [vector( -0.8301, 5.0000, 8.5622 ), 5.0990]
```

… or of group("World"):

```
-- [vector( -25.0000, 25.0000, 35.0000 ), 25.4951]
```

For models with no children, you can visualize the boundingSphere by setting the debug property of the model to TRUE. `member("3D").model("Wheel_4").debug = TRUE`

If you need to visualize the boundingSphere of a node that is not a model, or which has children, you can create an invisible plane with the appropriate dimensions and place it at the center of the node's boundingSphere.

```
on ShowBoundingSphere(a3DMember, aNode)
    vBoundingSphere = aNode.boundingSphere
    vPosition = vBoundingSphere[1]
    vRadius = vBoundingSphere[2]
    vSquareSide = (vRadius * vRadius) / 2.0
    vSide = sqrt(vSquareSide) * 2
    vName = aNode.name&"_debugModel"
    vPlane = a3DMember.model(vName)
    if not vPlane then
    vPlane = a3DMember.newModel(vName)
    end if
    vPlane.resource = a3DMember.modelResource(1) -- "DefaultPlane"
    vPlane.visibility = #none
    vPlane.transform.scale = vector(vSide, vSide, vSide)
    vPlane.worldPosition = vPosition
    vPlane.debug = TRUE
end ShowBoundingSphere
```

Example usage:

```
ShowBoundingSphere(member(2), member(2).group("Chassis"))
```

Pointing at positions in the world

The target of the `node.pointAt()` command refers to a position in world coordinates, or to the worldPosition of a node. Try the commands below in the Message window.

```
-- Lingo syntax
member("3D").model("Wheel_4").pointAt(0, 5, 0)
member("3D").model("Wheel_4").pointAt(member("3D").model(3))
// JavaScript syntax
member("3D").getPropRef("model", 5).pointAt(0, 5, 0);
member("3D").getPropRef("model", 5).pointAt(member("3D").getPropRef("model", 3))
```

For more details of the `pointAt()` command and the use of `pointAtOrientation`, see "Using pointAt() to rotate a node" on page 212.

Frame of reference

Every node has a transform property. This defines the position, rotation, and scale of the node, with reference to the node's parent. If the node's parent is group("World"), then the node's transform and the value returned by node.getWorldTransform() contains identical values. If you alter the values of the node's transform, then the changes will affect the node. The value returned by getWorldTransform() is no longer connected to the node, so changes made to that transform does not affect the node.

Scripting terms that affect the node's transform:

- node.transform
- node.rotate()
- node.translate()

- node.scale()
- node.worldPosition
- other Node.addChild(node, #preserveWorld)

Determining the frame of reference transform

The transform of a node indicates a modification from a reference transform. A node's transform says: "Starting with the transform of the node's parent, apply this translation, this rotation and this scale".

A node may be at the end of a chain of parent-child relationships. Each node in the chain applies its transform to the transform of its parent. Imagine that the default position of your arm is dangling down by your side. In order to put your finger in your ear, you may need to rotate your upper arm around your shoulder, rotate your forearm about your elbow, and rotate your hand around your wrist. The final "transform" of your finger depends on the transforms of all its "parents". Altering any one of the parents' transforms moves your finger.

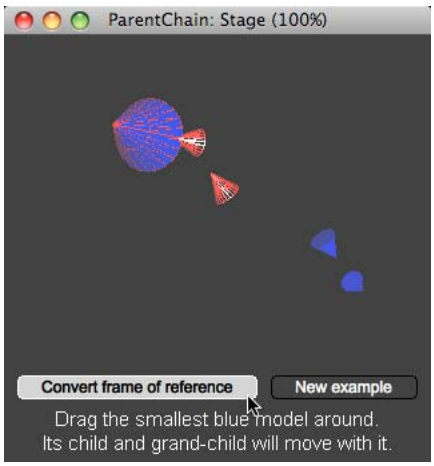
To determine the reference transform of a node, you can use the following expression:

```
vReferenceTransform = aNode.parent.getWorldTransform()
```

Converting from one frame of reference to another

Imagine that you have two nodes with different parents. Imagine that you want to place one node at exactly the same position as the other, without changing the parents. How do you work out what transform to apply?

To see a demonstration of this idea, download and launch the movie ParentChain.dir. This creates two parent-child hierarchies, one in red the other in blue. The smallest blue cone near the center of the sprite is at the root of the hierarchy of blue models. You can drag this model around, to see its child and grand-child follow it. The largest blue and red models are the grand-children of each chain. The `GetRelativeTransform()` handler converts the transform of the big red cone into the frame of reference of the big blue cone.



Click Convert Frame Of Reference to set the transform of the big blue cone

You can perform 3D mathematics on transforms. See "3D mathematics" on page 361 for more details. The `GetRelativeTransform()` handler works out the inverse of the world transform of the big blue cone's parent. This is the transform that moves the parent back to the origin of the world. It then applies the world transform of the big red cone to that inverse transform. In other words, it creates a transform that first sends the big blue cone back to the center of the world and then it moves it out again to the same location as the big red cone.

```
on GetRelativeTransform(aNode, aTargetNode)
    vTargetTransform   = aTargetNode.getWorldTransform()
    vFrameOfReference  = aNode.parent.getWorldTransform()
    vInverse           = vFrameOfReference.inverse()
    vRelativeTransform =  vInverse * vTargetTransform
    return vRelativeTransform
end GetRelativeTransform
```

An alternative approach is to change the parent of the big blue cone twice. Try the following commands in the Message window:

```
-- Lingo syntax
v3DMember = member("3D")
vTarget = v3DMember.model("Red3")
vModel = v3DMember.model("Blue3")
vParent = vModel.parent
vModel.parent = v3DMember.group("World")
vModel.transform = vTarget.getWorldTransform()
vParent.addChild(vModel, #preserveWorld)
// JavaScript syntax
v3DMember = member("3D");
vTarget = v3DMember.getPropRef("model", 3);
vModel = v3DMember.getPropRef("model", 6);
vParent = vModel.parent;
vModel.parent = v3DMember.getPropRef("group", 1);
vModel.transform = vTarget.getWorldTransform();
vParent.addChild(vModel, symbol("preserveWorld"));
```

userData

All nodes have a property list named `userData` attached to them.

```
-- Lingo syntax
put member("3D").model(1).userData
-- [:]
// JavaScript syntax
trace(member("3D").getPropRef("model", 1).userData);
// <[:]>
```

Permanence of the contents of the userData list

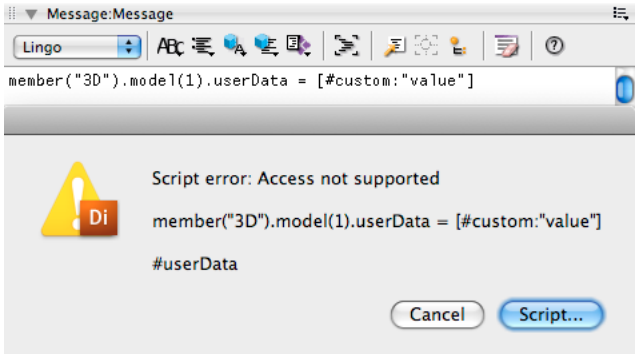
Depending on the third-party 3D design software used, your 3D designer may be able to add data to the `userData` lists of the various nodes when exporting the 3D world to W3D format. In this case, that data will be available to you permanently both during authoring and at runtime.

In Director 11.5, any data that you add to a node's `userData` list while authoring will not be saved with the 3D world, even if you use the member3D.saveWorld() command first. In other words, changes made to a node's `userData` list are available only for the current session.

In Director 11.5, you cannot set the `userData` list to a Lingo property list. This provokes an "Access not supported" script error. To change the contents of the `userData` list, you must act directly on the built-in list.

Also, in Director 11.5, you cannot use bracket access to get or set values in the `userData` list. If you try to set a value for bracket access, a script error occurs (Access not supported). When you use bracket access to retrieve a value, a pointer to the entire list is returned. However, you can create a pointer to the list and use all the standard list methods on that.

When working directly on a Node.userData lists, some standard list methods are not supported.



When working directly on aNode.userData lists, some standard list methods are not supported.

Accessing the content of the userData list

To avoid confusion with the limited range of methods that the userData list supports, you may find it easier to create a pointer to a userData list. You will be able to use all the standard list commands on this de-referenced list. For example:

```
-- Lingo syntax
vUserData = member("3D").model(1).userData
vUserData[#custom] = "value"
put member("3D").model(1).userData
-- [#Custom: "value"]
// JavaScript syntax
vUserData = member("3D").getPropRef("model", 1).userData;
<[:]>
vUserData[symbol("custom")] = "value";
0
trace(member("3D").getPropRef("model", 1).userData);
// <[#Custom: "value"]>
```

Uses of the userData list

You can use a node's userData list to store any information that you want. You can, for example, store a list of objects that an avatar is carrying in the avatar model's userData list. One very practical use is as a means to add script instances directly to a node, in much the same way that you can add behaviors to a sprite. You can then send events to the node's userData list, using a command similar to the following:

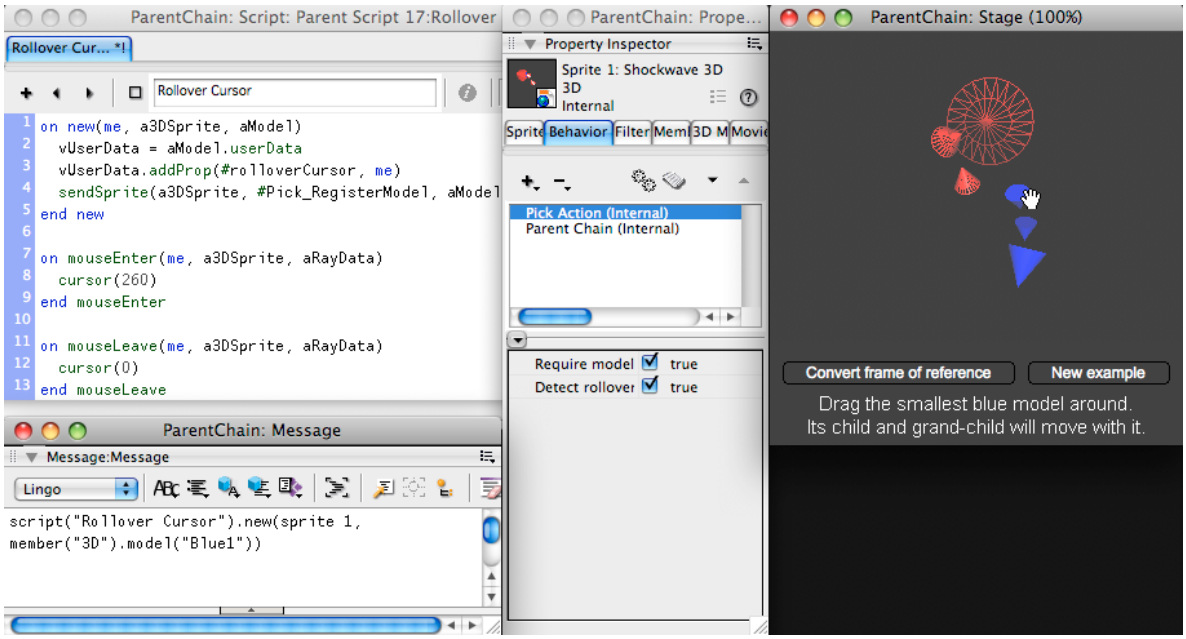
```
-- Lingo syntax
call(#CustomEvent, aNode.userData{, parameter, …})
```

Note: *In Director 11.5, the call() method does not work in JavaScript syntax*

To see a full description of this technique, see "Pick Action behavior" on page 244. To see an example, download and launch the movie ParentChain.dir. In the Message window, type the following commands:

```
-- Lingo syntax
vScript = script("Rollover Cursor")
vScript.new(sprite 1, member("3D").model("Blue1"))
```

This adds an instance of the rollover cursor script to the userData list of the smallest blue cone. When you move the mouse pointer over that model, the cursor changes to an open hand to indicate that you can drag the model.



A script instance added to the userData list of a model can handle events like a behavior on a sprite

Cloning

You can create duplicate copies of any node in a 3D member, using the node.clone() function. To duplicate all the node's children and all modelResources, shaders, and textures used on any of the models in the hierarchy, use node.cloneDeep().

When you use node.cloneDeep(), new objects are added to the 3D cast member. These objects are automatically renamed during the process, in order to avoid naming clashes. See "3D namespace" on page 85 for more details.

Cameras

Cameras act as windows into a 3D world and define a view into the 3D world. Each camera that exists in a 3D cast member offers a different view into it, and each sprite that uses a 3D cast member uses one of these cameras.

A camera's position can be moved with the Property inspector or the Shockwave® 3D window. You can also use the Adobe® Director® 3D behaviors, Lingo, or JavaScript syntax to add a camera and manipulate its positions.

In a movie camera in the real world, you can change the view by altering:

- The position of the camera
- The direction in which you point the camera
- The tilt of the camera (to create a sloping horizon)
- The focal length of the camera lens (zoom). See "Perspective" on page 102 for more details.

For a detailed treatment of these features, see "Moving the camera" on page 216.

Virtual controls

A Shockwave 3D camera gives you additional controls for the view:

- You can choose an orthographic projection to remove all trace of perspective. For an overview of camera projection features, see "Perspective" on page 102. For details of the specific properties, see projection, projectionAngle, its synonym fieldOfView, and orthoHeight.
- You can determine which objects are visible. See "RootNode" on page 107 and rootNode.
- You can choose which slice of the 3D world is visible. See "Hither and yon" on page 110, hither and yon.

Additional features

You can also control the appearance of the view:

- If you use multiple cameras in your 3D sprite, you can define the insert rect of the additional cameras. See "Multiple cameras" on page 103 and "MiniMap" on page 233
- You can set the background color for each camera. See "Color buffer" on page 105.
- You can make distant objects appear in muted colors, using fog. See "Fog" on page 111.
- You can add overlays and backdrops. See "Overlays and backdrops" on page 112.

Properties shared with other nodes

Like groups, lights and models, all cameras share a set of properties and methods with all other nodes. These include:

- A unique node name
- A transform to define its position, orientation, and scale.
- The ability to pointAt() another node or a point in space.
- A position in the world's parent-child hierarchy, with a single parent and possibly multiple children. See "Node hierarchy" on page 91.
- A userData property list which can store any kind of information

For more details on 3D nodes, see "Nodes" on page 88.

Perspective

On many movie and photo cameras, you can adjust the zoom. In the real world, you change the focal length of the camera's lens system and the angle of view changes. With a virtual 3D camera, you simply set the angle of view directly.

Shockwave 3D uses two synonymous terms: projectionAngle and fieldOfView. You are free to use either term. To simplify this article, projectionAngle is used exclusively throughout.

A Shockwave 3D camera can also be set to display an orthographic view. This is the view that is used for drafting architectural and technical designs. In a perspective view, distant objects appear smaller, and parallel lines in the world appear to converge. In an orthographic view, all objects appear at their actual size and parallel lines remain parallel.

When the camera is in orthographic mode, if you move the camera forwards or backwards, the shape and size of the models do not change. In the perspective mode, moving the camera forwards and backwards gives you a natural impression of movement. The camera's projectionAngle determines the "zoom" of the image.

In the orthographic mode, the camera's orthoHeight determines how many world units are visible vertically in the sprite. In an orthographic view, changing the distance of the camera from the target does not change the shape and size of the models.

Default values

By default, the camera of a new #shockwave3d member is set to show a #perspective projection, with a projectionAngle of 34.5160°. This makes for a natural-looking view.

```
v3DMember = new(#shockwave3d)
vCamera  = v3DMember.camera(1)
put vCamera.projection
-- #perspective
put vCamera.projectionAngle
-- 34.5160
```

The camera's orthoHeight is already set to a default value, even if it is not yet in use:

```
put vCamera.orthoHeight
-- 200.0000
```

A camera created by the member3D.newCamera() function has a different projectionAngle of 30° by default:

```
vNewCamera = v3DMember.newCamera("Test")
put vNewCamera.projection
-- #perspective
put vNewCamera.projectionAngle
-- 30.0000
put vNewCamera.orthoHeight
-- 200.0000
```

Examples

The following example changes the projection mode of the camera of sprite("3D") to #perspective and sets the camera's projectionAngle to 30°:

```
-- Lingo
vCamera = sprite("3D").camera
vCamera.projection = #perspective
vCamera.projectionAngle = 30.0 -- degrees
// JavaScript
vCamera = sprite("3D").camera;
vCamera.projection = symbol("perspective");
vCamera.projectionAngle = 30.0; // degrees
```

The following example changes the projection mode of the camera of sprite("3D") to #orthographic and sets the camera's orthoHeight to 200:

```
-- Lingo
vCamera = sprite("3D").camera
vCamera.projection = #orthographic
vCamera.orthoHeight = 200.0 -- world units
// JavaScript
vCamera = sprite("3D").camera;
vCamera.projection = symbol("orthographic");
vCamera.orthoHeight = 200.0; // world units
```

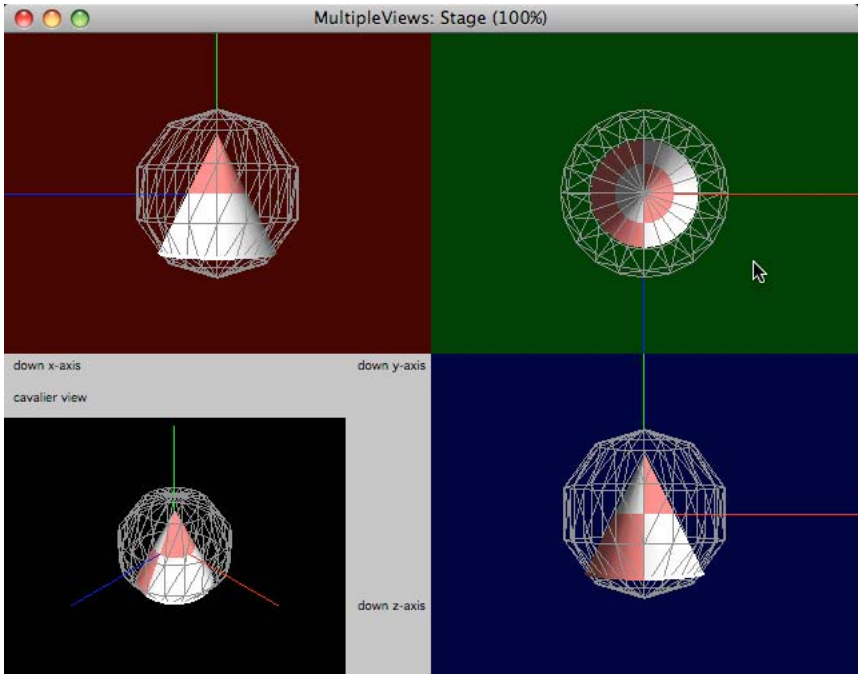
Multiple cameras

By default, a 3D sprite displays the view from only one camera. Every 3D member has a camera named "Default View". The view from this camera fills the entire area of the sprite. In a 3D design application, you can create more than one camera and export it as an object in the W3D file. You can change the camera that is used to create the main view for the 3D sprite to any of these cameras.

Imagine that a sprite named "3D" contains a 3D cast member with 3 or more cameras. This example will set the camera of sprite("3D") to the third camera of the cast member:

```
-- Lingo
v3DSprite = sprite("3D")
v3DMember = v3DSprite.member
v3DSprite.camera = v3DMember.camera(3)
// JavaScript
v3DSprite = sprite("3D");
v3DMember = v3DSprite.member;
v3DSprite.camera = v3DMember.getPropRef("camera", 3);
```

You can use several sprites to display the same 3D cast member, and use a different camera for each sprite. This will give you several views into the same 3D world. To see an example, download the movie MultipleViews.dir and launch it.



Viewing a 3D cast member with different cameras in different sprite

Creating a new camera on the fly

You can use the member3d.newCamera() function to create a new camera. This example creates a new camera in the 3D cast member that is displayed in sprite("3D"), and calls it "Top Down". It then places the camera above the center of the world and points it downwards:

```
-- Lingo
v3DSprite = sprite("3D")
v3DMember = v3DSprite.member
vCamera = v3DMember.newCamera("Top Down")
vCamera.projectionAngle = 45.0
vCamera.worldPosition = vector(0, 250, 1)
vCamera.pointAt(0, 0, 0)
// JavaScript
v3DSprite = sprite("3D");
v3DMember = v3DSprite.member;
vCamera = v3DMember.newCamera("Top Down");
vCamera.projectionAngle = 45.0;
vCamera.worldPosition = vector(0, 250, 10);
vCamera.pointAt(0, 0, 0);
```

Rendering a secondary camera in the 3D sprite

The view from the new camera is automatically rendered to the 3D sprite. You need to provide two additional instructions. Here are the scripting terms you need to use: sprite3d.addCamera() and camera.rect.

The following example creates an inset at the top left corner of sprite("3D") to show the output of the new camera ("Top Down"):

```
-- Lingo
v3DSprite = sprite("3D")
vCamera = v3DSprite.member.camera("Top Down")
v3DSprite.addCamera(vCamera)
vCamera.rect = rect(0, 0, 160, 120)
// JavaScript
v3DSprite = sprite("3D");
vCamera = v3DSprite.member.getPropRef("camera", 2);
v3DSprite.addCamera(vCamera);
vCamera.rect = rect(0, 0, 160, 120);
```

All sprites that display the same 3D cast member display the same secondary camera views. It is not advisable to use the same camera as the main view for one sprite and as a secondary camera for another sprite. The dimensions of the sprite where the camera is used for the main view affects the camera.rect for the secondary view in the other sprite. The two views will have the same dimensions. You can set the rect of the primary camera of a 3D sprite, but you may get unexpected results if you make the dimensions of the camera.rect smaller than the dimensions of the sprite.

Deleting a secondary camera view

To remove a camera that you have added to a 3D sprite, you can use sprite3d.deleteCamera(). This will remove the secondary camera view from all sprites that display the same 3D cast member. It will not delete the camera from the 3D cast member itself.

Color buffer

Every camera has a colorBuffer property. You cannot access this directly, but you can get and set the values of its two properties:

- camera.colorBuffer.clearValue
- camera.colorBuffer.clearAtRender

The clearValue property defines a fill color and clearAtRender determines whether that color is used to fill in any blank spaces behind visible models.

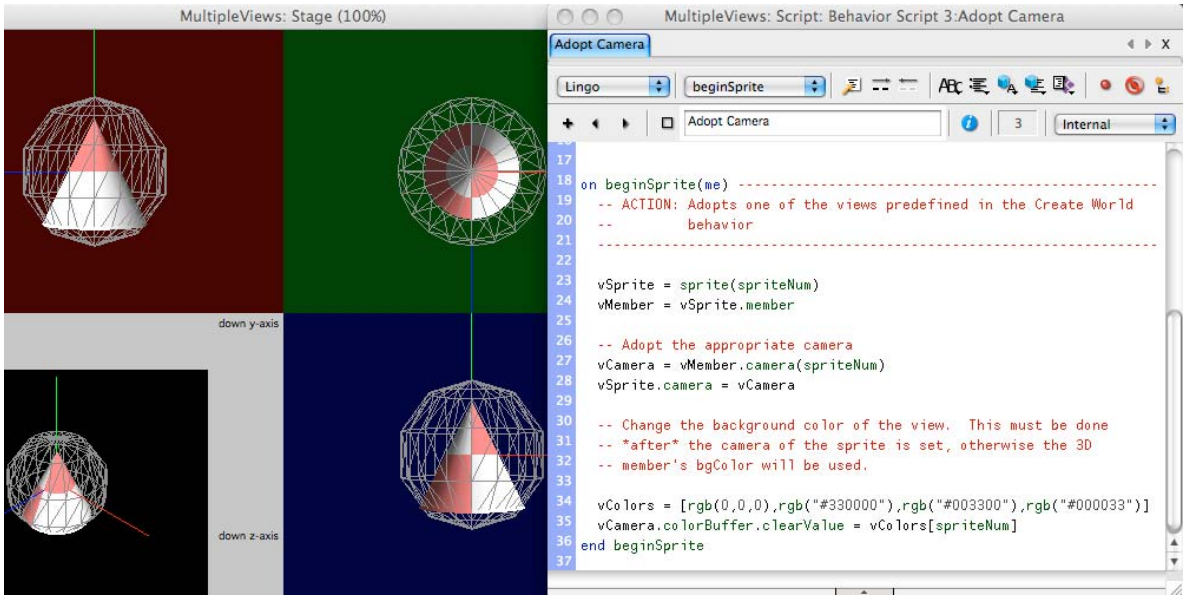
clearValue

The following example shows that the colorBuffer.clearValue is closely related to the 3D cast member's background color. Changing the member's bgColor will change the colorBuffer.clearValue for all cameras of that member.

```
put member("3D").bgColor
-- color( 0, 0, 0 )
put member("3D").camera(1).colorBuffer.clearValue
-- color( 0, 0, 0 )
vSecond = member("3D").newCamera("Second")
member("3D").bgColor = color( 111, 22, 3 )
put member(1).camera(1).colorBuffer.clearValue
-- color( 111, 22, 3 )
put vSecond.colorBuffer.clearValue
-- color( 111, 22, 3 )
```

You may want different cameras in the same 3D cast member to have different background colors. If so, you must first adopt the camera for the 3D sprite, and then change the camera's colorBuffer.clearValue. If you set the colorBuffer.clearValue first, then its value will be overwritten by the member's bgColor when you adopt the camera for the sprite.

To see an example, download the movie MultipleViews.dir and launch it. Each of the sprites shows a view from a different camera with a different colorBuffer.clearValue.



Set a camera's colorBuffer.clearValue after adopting the camera for a sprite

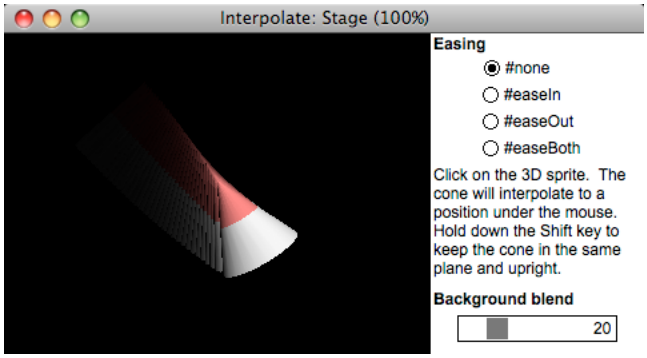
clearAtRender

When a camera refreshes its view of the world, the render process follows this order:

- Fill view with background color
- Show backdrops
- Render visible models
- Show overlays

You can tell a camera not to fill its rectangular area with the color defined by its colorBuffer.clearValue. To do this, you set its colorBuffer.clearAtRender property to FALSE. This means that the camera will start with whatever image it already holds in its display buffer and render the current view of the scene on top of that. If you are using only one camera, this approach leads to trails. You can make these trails fade over time by using a semi-transparent backdrop.

To see an example, download the movie Interpolate.dir and launch it. Try different values for the blend of the backdrop.



Using colorBuffer.clearAtRender = FALSE to create trails, and a backdrop to make the trails fade out

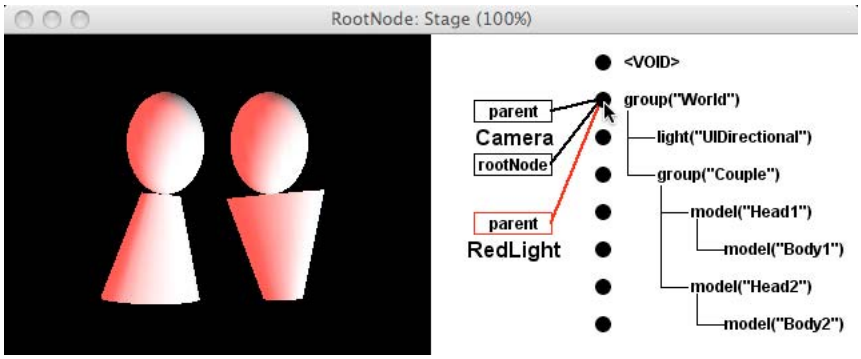
See "Overlays and backdrops" on page 112 for more information on backdrops. See "Sky box" on page 108 for an example of using `colorBuffer.clearValue = FALSE` with multiple cameras.

RootNode

A camera's rootNode property determines which models are rendered by the camera, and which lights are used in the rendering process. By default, a camera's rootNode property is set to the 3D cast member's group("World"). This means that:

- The camera can "see" all the models that are children of group("World")
- All the lights, which are children of group("World"), affect all these models.

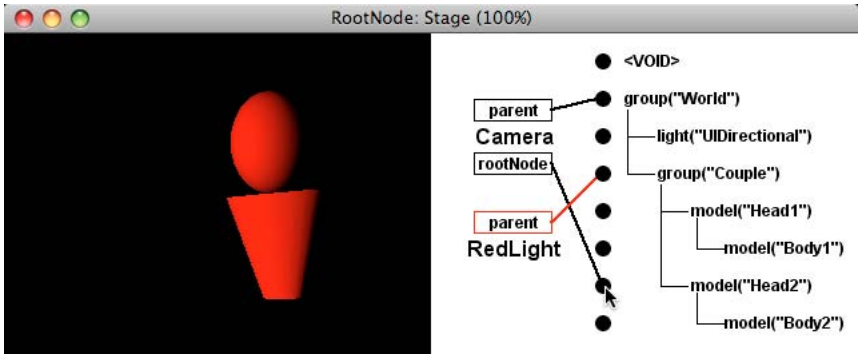
You can set the rootNode of the camera to any node you like. The node does not even need to be a child of group("World"). To see the effect that this has, download the movie RootNode.dir and launch it. On the left, you will see a 3D sprite. On the right, you will see the hierarchy of nodes in the 3D world.



A camera will render its rootNode and all its children using lights that are on the same branch in the hierarchy

You can drag the lines from the parent and rootNode boxes to any of the dots that indicates a node in the 3D cast member. This will change the relationship between the camera and the literality") in the 3D world. Experiment with different settings.

In the following illustration, the parent of the light("RedLight") has been set to the parent of the model("Head2"). These two nodes share the same parent. However, the white light("UIDirectional") is nearer the head of the hierarchy. The model("Head1") is not in the same branch as the camera's rootNode.



Only the model("Head2") and its children are rendered, and they are lit only by light("Red Light")

For more information on parent-child hierarchies, see "Node hierarchy" on page 91.

Sky box

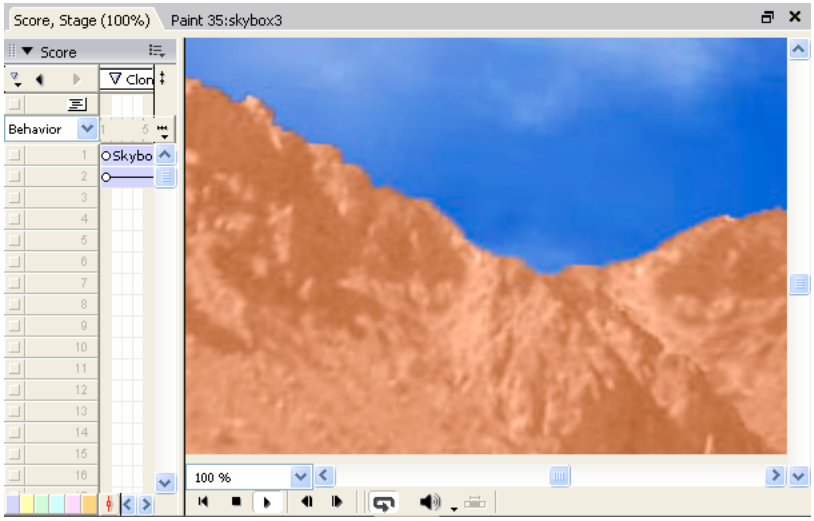
Imagine that you are traveling in a fast car along a straight road. Objects that are close to you, such as telephone poles along the sides of the road, appear to move past very fast in the opposite direction. The sky, on the other hand, is a huge distance away. Even if you travel fast, the moon and the clouds appear to travel with you. If you create a 3D world with an outdoor scene, you will want to be able to simulate this effect of a motionless sky. The solution is to create a sky box. The principle is to create a specific camera to render the image of the sky, and then to render the rest of the world on top of the sky image, using a different camera. You can do this with the techniques that you saw earlier in this section:

- sprite3d.addCamera() in "Multiple cameras" on page 103
- camera.rect in Multiple cameras
- camera.colorBuffer.clearAtRender in "Color buffer" on page 105
- camera.rootNode in "RootNode" on page 107

You also need to:

- Synchronize the node.rotation of the camera that shows the sky with the rotation of the camera that shows the rest of the world. For more information, see "Rotating around an object" on page 218 and "3D mathematics" on page 361
- Change the natural parent-child hierarchy of objects. For more information on this, see "Node hierarchy" on page 91.

To see a demonstration of a sky box, download the movie Skybox.dir and launch it. You can use the arrow keys to rotate the main camera or you can click on the 3D sprite, and then drag the mouse pointer in the direction in which you want the camera to turn. To move forward or back, you can use the I and O (in and out) keys, or the W and S keys.



A skybox is simply a box seen from the inside. Low-quality textures destroy the illusion of a perfect sky.

The Skybox movie uses a mesh in the shape of a box, viewed from the inside. Each wall of the box has a different shader. The textures in each shader are carefully designed to fit together perfectly at the seams. The illusion is shattered if textures and shaders have not been properly prepared in a third-party application. The example creates a sky box from a model named "Skybox_mesh" in the 3D cast member found in sprite 1. You can place this code in a beginSprite() handler in a behavior on the sprite.

```
vSprite = sprite(1)
vMember = vSprite.member
pMainView = vMember.camera(1) -- "DefaultView"
-- Create a new camera and make sure that its view is
-- rendered first
pSkyCam = vMember.newCamera("Skybox")
pSkyCam.fieldOfView = pMainView.fieldOfView
vSprite.addCamera(pSkyCam, 1)
-- As vSprite.camera(1), pSkyCam.rect is set
-- automatically to fill the sprite
-- Ensure that the SkyCam only sees the sky
vSkyBox = vMember.model("Skybox_mesh") -- <HARD-CODED>
-- No other cameras can see vSkyBox...
vSkyBox.removeFromWorld()
-- ... and pSkyCam sees no other models
pSkyCam.rootNode = vSkyBox
-- Ensure that the view from the DefaultView camera is
-- rendered over the top of the view from the SkyCam
pMainView.colorBuffer.clearAtRender = FALSE
```

This code creates a sky that is good enough for a static world. However, the illusion will be broken as soon as first-person player moves. You still need to ensure that the camera that renders the sky and the camera that renders the rest of the world always point in the same direction. You may have noticed the special variables pMainView and pSkyCam. If you declare these variables as properties in your behavior, you can use an exitFrame() handle to synchronize the rotation of the two cameras.

```
on exitFrame(me) ----------------------------------------
-- ACTION: Ensures that the Skybox camera always faces in
--         the same direction as the DefaultView camera,
--         to give the illusion that the sky is part of
--         the world.
-------------------------------------------------------
pSkyCam.transform.rotation = pMainView.transform.rotation
end exitFrame
```

For information on how to constrain a first-person camera to the appropriate part of your world, see “Collisions” on page 279 and “Physics” on page 293. See also this post from Duck.

Hither and yon

A 3D sprite is a window through which you can watch what is happening in a virtual world. You can imagine that the camera of the 3D sprite represents your eye. Your eye cannot see the 3D sprite window. There is always a slight gap between the two. There may be models in the virtual world that you cannot see. Models that are behind the camera or too far to the side are not shown. There may also be objects that come between your eye and the "window". These objects will not be shown either. If a model comes so close to the camera that it intersects with the window, you may see gaps appearing in the model.

You can set the distance between your eye and the “window” by setting the hither property of the camera. A more technical term for this “window” is the hither plane. The hither plane is a plane that is perpendicular to the direction in which the camera is pointing.

Minimum value of hither

The hither value cannot be less than 1.0 world unit. This has important implications concerning the size of your models. You will not be able to move the camera closer than 1.0 world unit to any model without holes appearing in the model.

Note: *If your models are very small, and you move the camera close to see them in close-up, you may find holes appearing in their surface.*

If you find yourself stuck with tiny models, you can increase the scale of all models in your world by changing the scale of group(“World”). However, if you do this, you will probably need to change many other settings in the world.

X-ray vision: using hither creatively

In some situations, a wall or some other object may block the view of the camera. You can adjust the camera's hither value so that no part of the wall is visible, and the scene on the far side is clearly visible. You can also use hither to create cut-away views of objects.

Yon

The yon plane is further away than the hither plane. The 3D sprite will not show any model faces that are further away from the camera than its yon plane. Imagine a spaceship flying through a field of asteroids. If the value of yon is high, the Director playback engine renders every asteroid in front of the spaceship. Distant asteroids may appear smaller than a pixel, but the playback engine will process them anyway. You can improve playback performance by setting yon to a lower value. You may need to experiment to get the value just right. If you set the value of yon too low, big asteroids may suddenly appear out of nowhere.

Fog

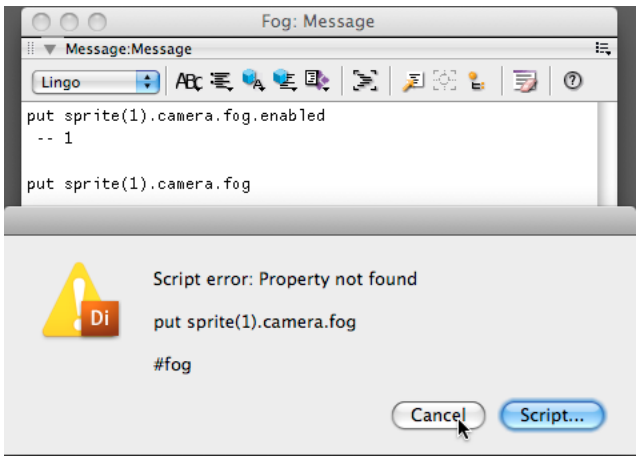
In the real world, you can hold your hand up in front of you and look at a distant scene behind your hand. Your hand will be out of focus. If you focus on your hand, the distant scene will become a blur of colors. You can judge the distance to an object that is close to you by how much work your eyes have to do to focus on it.

If there is a lot of dust or moisture in the air, the brightness of colors in the distance is decreased. You can use the vividness of colors to judge just how far is a distant object.

In a virtual 3D world, everything is always in focus, regardless of the distance, and there is no dust. Nonetheless, you can use camera.fog to blur distance objects and attenuate their colors. In a Shockwave 3D world, fog can be any color. The most subtle effects occur when you choose a color very close to the background color of the member. Fog is applied to the faces of models, and not to background, backdrops, or overlays.

Properties

You cannot access camera.fog directly. If you try to do so, a script error occurs.



You can access the properties of camera.fog, but not the fog object itself

- `fog.enabled` can be TRUE or FALSE. It allows you to toggle on and off the use of fog. The default value is FALSE.
- `fog.color` defines the color of the fog. The default value is color(0, 0, 0) or black. For most natural results, use a value that is close to the member3D.bgColor of the 3D cast member.
- `fog.far` defines the distance in world units from the camera where the fog reaches 100% opacity. Any model face that is more than fog.far away from the camera will be set to fog.color. The colors applied to the face by its shader and texture will be completely ignored. The default value is 1000.0.
- `fog.near` only has an effect if fog.decay mode (see below) is set to #linear. In this case, fog.near defines the distance in world units from the camera at which fog starts to be applied to models. If fog.near is greater than fog.far the foreground will be filled with fog but the zone beyond fog.near will be clear. The default value is 0.0.
- `fog.decayMode` can take three different values:
  - #linear: the fog density is linearly interpolated between fog.near and fog.far.
  - #exponential: fog.far is the saturation point; fog.near is ignored.
  - #exponential2: as with #exponential, fog.far is the saturation point; fog.near is ignored. The difference is that fog in the foreground will be less dense.

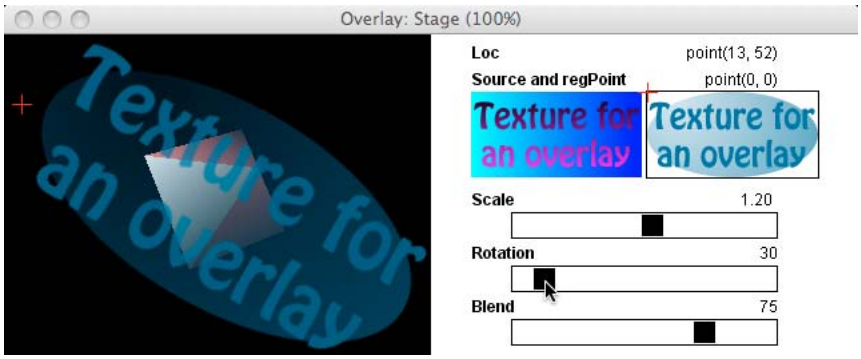
The default setting for this property is #exponential.

Note: *If you use the same color for fog as for the bgColor of your 3D cast member, you may wish to set the yon property of the camera to the same value as fog.far. See "Hither and yon" on page 110 for more details.*

Overlays and backdrops

Overlays and backdrops are rectangular 2D areas that you can place in a camera view of the world. Backdrops are rendered to the camera view after the background fill color, but before any of the 3D models. Backdrops thus appear behind all models.

Overlays are rendered to the camera view after all the 3D models have been rendered. Overlays thus appear on top of all 3D models.To experiment with all the properties of a camera.overlay, download the movie Overlay.dir and launch it.



You can set the loc, regPoint, source, scale, rotation, blend of an overlay or backdrop.

D sprites rotate around their regPoint. Overlays and backdrops always rotate around their center. The regPoint property of overlays and backdrops is a misnomer. The regPoint property of an overlay indicates indirectly where the center of the overlay is compared to the loc of the overlay within the camera.rect. The same is true for backdrops. See "Properties for overlays and backdrops" on page 114 for more details. You can find a Parent Script called "Rotatable Overlay" in the Overlay.dir movie. Create an instance of this script and use it to control the rotation of an overlay around a point other that its center.

The properties and methods of overlays and backdrops are very similar. In this article, all the examples are written for overlays. To perform the same operations with backdrops, simply replace the string "overlay" with "backdrop" wherever it appears in the examples.

Displaying textures

Overlays and backdrops display textures. Textures are always rectangular and always have dimensions that are a power of 2 (1, 2, 4, 8, 16). To create overlays and backdrops of different shapes and proportions, you can use textures with alpha transparency. For more information on textures, see "Textures" on page 141.

Adding an overlay to a camera

To add an overlay to a camera, you can use either camera.addOverlay() or camera.insertOverlay(). The addOverlay() method places the overlay on top of all existing overlays. The insertOverlay() method inserts the overlay at the given layer in the overlay rendering sequence. Overlays at that layer and in higher-numbered layers are moved up one layer to make room for the inserted overlay.

The following example creates an overlay in the camera of sprite(1), using a texture named "Overlay1" (texture 2 in the 3D cast member). It places the top left corner of the texture at point(32, 56). The overlay is not rotated.

```
-- Lingo
v3DSprite = sprite(1)
v3DMember = v3DSprite.member
vTexture  = v3DMember.texture("Overlay1")
vLoc      = point(32, 56)
vRotation = 0
v3DSprite.camera.addOverlay(vTexture, vLoc, vRotation)
// JavaScript
v3DSprite = sprite(1);
v3DMember = v3DSprite.member;
vTexture  = v3DMember.getPropRef("texture", 2);
vLoc      = point(32, 56);
vRotation = 0;
v3DSprite.camera.addOverlay(vTexture, vLoc, vRotation);
```

Counting overlays or backdrops for a given camera. When you add a new overlay or backdrop in Lingo you can use the camera.overlay.count or camera.backdrop.count property to determine which layer the overlay or backdrop was added to.

In JavaScript, the equivalent expressions are camera.count("overlay") and camera.count("backdrop"). The following example obtains the index number of the most recently added overlay, and sets the scale of that overlay to 0.5:

```
-- Lingo
vCamera = sprite(1).camera
vCamera.addOverlay(vTexture, vLoc, vRotation)
vIndex = vCamera.overlay.count
vCamera.overlay[vIndex].scale = 0.5
// JavaScript
vCamera = sprite(1).camera;
vCamera.addOverlay(vTexture, vLoc, vRotation);
vIndex = vCamera.count("overlay");
vCamera.getPropRef("overlay", vIndex).scale = 0.5;
```

Inserting an overlay

The following example inserts an overlay with the same texture as used in the previous example at the same loc. It inserts the overlay into layer 1, pushing the previous overlay up into layer 2. The new overlay (in layer 1) is rotated through 30 degrees around its center. It is partly hidden behind the non-rotated overlay.

```
-- Lingo
v3DSprite = sprite(1)
v3DMember = v3DSprite.member
vTexture  = v3DMember.texture("Overlay1")
vLoc      = point(32, 56)
vRotation = 30
v3DSprite.camera.insertOverlay(1, vTexture, vLoc, vRotation)
// JavaScript
v3DSprite = sprite(1);
v3DMember = v3DSprite.member;
vTexture  = v3DMember.getPropRef("texture", 2);
vLoc      = point(32, 56);
vRotation = 30;
v3DSprite.camera.insertOverlay(1, vTexture, vLoc, vRotation);
```

Removing an overlay

Use camera.removeOverlay() to remove an overlay from a camera. The following example removes the overlay in layer 1 from the camera of sprite 1:

Note: *If the camera of sprite 1 does not currently display any overlays, this code fails silently.*

```
-- Lingo
sprite(1).camera.removeOverlay(1)
// JavaScript
sprite(1).camera.removeOverlay(1);
```

Adding, inserting and removing backdrops

The equivalent methods for backdrops are:

- camera.addBackdrop()
- camera.insertBackdrop()
- camera.removeBackdrop()

Properties for overlays and backdrops

When you add or insert an overlay of backdrop, you manually set the following properties:

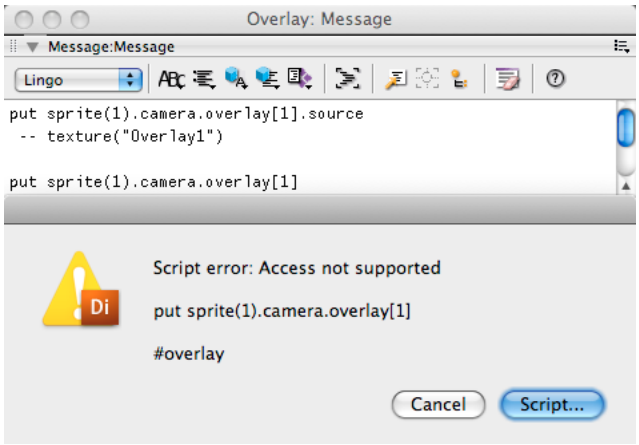
- source: The texture displayed in the overlay or backdrop
- loc: The position within the camera.rect which corresponds to the regPoint (see below) of the overlay or backdrop
- rotation: the rotation of the backdrop around its center

The following properties are set to default values:

- scale: A positive floating-pointing number. You can scale a 3D model along three axes (x, y and z). The scale of an overlay or backdrop is a single number that applies to both the horizontal and vertical axis. You cannot change the proportion of the texture. The default value is 1.0.
- regPoint: A point relative to the top left corner of the unrotated overlay or backdrop. The overlay or backdrop is positioned first with its regPoint at the chosen loc within the camera view, and then it is rotated around its center. See "Rotating overlays and backdrops" on page 116 for more information on this feature.
- blend: A floating-point number between 0.0 (completely transparent) and 100.0 (completely opaque).

Accessing an overlay or backdrop in a particular layer

You cannot obtain a pointer to a particular overlay or backdrop in Lingo or JavaScript syntax. The following illustration shows that doing so leads to a script error:



You can access the properties of an overlay or backdrop, but not the overlay or backdrop itself

If you need to make several changes to the same overlay, you need to refer to `camera.overlay[layerIndex]` (Lingo) or `camera.getPropRef("overlay", layerIndex)` (JavaScript) each time.

The following example below adds a new overlay, determines which layer the overlay was added to, and sets the scale, regPoint and blend properties of the new overlay:

```
-- Lingo
v3DSprite = sprite(1)
v3DMember = v3DSprite.member
vCamera = v3DSprite.camera
vTexture = v3DMember.texture(2)
vLoc = point(50, 50)
vRotation = 90
vCamera.addOverlay(vTexture, vLoc, vRotation)
vIndex = vCamera.overlay.count
vCamera.overlay[vIndex].scale = 0.5
vCamera.overlay[vIndex].regPoint = (vTexture.member).regPoint
vCamera.overlay[vIndex].blend = 25 -- %
// JavaScript
v3DSprite = sprite(1);
v3DMember = v3DSprite.member;
vCamera = v3DSprite.camera;
vTexture = v3DMember.getPropRef("texture", 2);
vLoc = point(50, 50);
vRotation = 90;
vCamera.addOverlay(vTexture, vLoc, vRotation);
vIndex = vCamera.count("overlay");
vCamera.getPropRef("overlay", vIndex).scale = 0.5
vCamera.getPropRef("overlay", vIndex).regPoint = (vTexture.member).regPoint
vCamera.getPropRef("overlay", vIndex).blend = 25;
```

Note: *The expression* `(vTexture.member).regPoint` *gives an error if parentheses are not used.*

Finding a particular layer

There is no built-in method for keeping track of which layer contains a given overlay. If you use overlays and backdrops intensively, you can create your own "layer manager" scripts. For less intensive use, where the only one layer uses a particular texture, you can use a handler such as this:

```
-- Lingo
on GetOverlayIndex(aCamera, aTexture) --------------------------
-- INPUT: <aCamera> must be a 3D camera object
--        <aTexture> must be a texture object
-- OUTPUT: Returns the index number of the topmost overlay
--         that contains the given texture
-------------------------------------------------------------
    ii = aCamera.overlay.count
    repeat while ii
        if aCamera.overlay[ii].source = aTexture then
            exit repeat
        end if
        ii = ii - 1
    end repeat
    return ii
end GetOverlayIndex
```

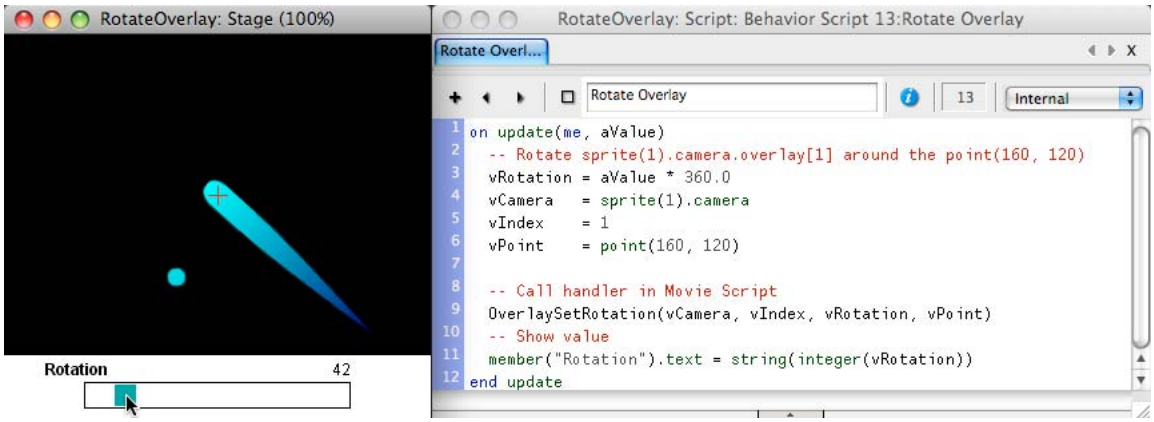
Dynamic overlays and backdrops

You can create overlays and backdrops that display animated 2D images. All you need to do is to update the texture that is displayed by the overlay or backdrop, and its image will update automatically. You can see an example of this with editable text in "Text in overlays and backdrops" on page 116.

Rotating overlays and backdrops

The built-in rotation feature is fine if you simply want to place a static overlay or backdrop at a particular angle. For dynamic rotation, you may need a different solution. To simulate the needle on a speedometer, for example, you may need to rotate the overlay or backdrop around its regPoint. The solution is to roll your own rotation code.

To see an example script, download the movie RotateOverlay.dir. This includes a Movie Script with a handler named OverlaySetRotation(). You can use this handler to rotate an overlay around its regPoint.



A call to OverlaySetRotation() rotates an overlay about its regPoint

Drag the slider button to rotate the overlay. The regPoint of the blue overlay is set to the regPoint of the Needle vectorShape member. If you change the regPoint of the member and relaunch the movie, the overlay rotates around its new regPoint.

Text in overlays and backdrops

For most of your text requirements, Director's 2D text and field members provide everything you need. Sometimes, however, you may need to display 2D in a 3D environment. To do so, you can use overlays and backdrops. Overlays and backdrops display textures. You can set the member of a texture to any cast member that has an image property, including Text members. To make the text look crisp when it is used in a texture, ensure that the rect of the Text member has dimensions which are a power of 2.

The following example sets the rect of the Text member "Text" to rect(0, 0, 128, 256):

```
-- Lingo
vMember = member("Text")
vMember.boxType = #fixed
vMember.width = 128
vMember.height = 256
put vMember.rect
-- rect(0, 0, 128, 256)
// JavaScript
vMember = member("Text");
vMember.boxType = symbol("fixed");
vMember.width = 128;
vMember.height = 256;
trace(vMember.rect);
// <rect(0, 0, 128, 256)>
```

Note: *If you try to set the rect of a text member directly, the width of the cast member gets set, but its height remains unchanged. If you change the width and the height explicitly, then the rect of the text member is set correctly.*

You can even simulate editable text in an overlay. To do this, place a text sprite over the 3D sprite at the position where you want to see the text. Create an overlay containing the image of the text member inside the 3D camera view. Each time the user presses a key, update the texture that is displayed in the overlay. To see this technique in action, download the movie EditableOverlay.dir and launch it. Click in the text area and start typing.

An overlay in the 3D sprite displays a texture that uses the image of an editable text member

When the mouse pointer moves over the editable text sprite, the cursor changes to the text insert cursor.

Note: *When you click in the text sprite or make a selection, the insertion point or current selection is not displayed. You can create a more elaborate version of the Editable Overlay behavior if the insertion point information is necessary. See Manipulating text with Lingo or JavaScript syntax for more information.*

In the behavior extract below, pTexture is a pointer to the texture in the 3D member which is displayed in the overlay, and pTextMember is the Text member which is placed on top of the 3D sprite. This extract is found in a behavior attached to the Text sprite.

```
on keyUp(me) ------------------------------------------------------
-- ACTION: Refreshes the texture of the overlay to show the
--         latest text in the editable text member.  The
--         overlay will update automatically.
-----------------------------------------------------------------
pTexture.member = pTextMember
end keyUp
```

The paler rectangle around the text is created by a separate overlay.

Interacting with overlays and backdrops

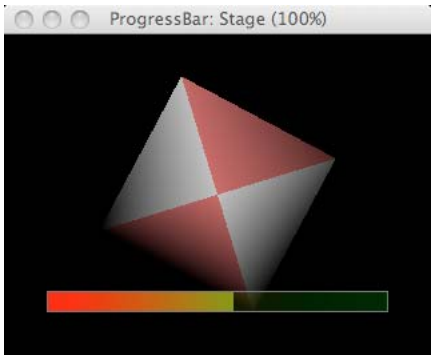
You can create features that allow end-users to interact with overlays and backdrops in a variety of ways. This article presents four examples:

- Progress bar
- Moving an overlay with the mouse
- Rollover tool tip
- 2D buttons in a 3D world

There are many other possible uses for backdrops and overlays.

Progress bar

You can use progress bars to indicate a waiting period or a value for a property such as Life Force. To see an example, download the movie ProgressBar.dir and launch it.

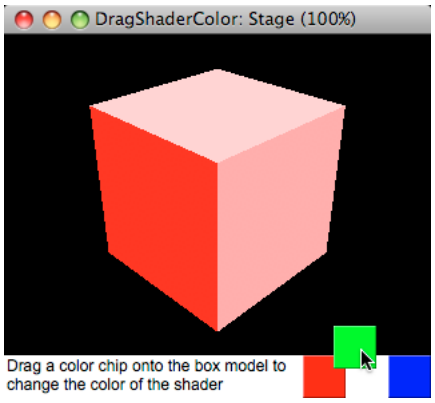


A progress bar created from a texture with a semi-transparent image

The progress bar is created from an image object. The alpha channel of the image object is filled with black for the opaque parts and light gray for the semi-transparent part. The image is applied to a texture, and the texture is used in an overlay. See the OverlaySetProgress() handler in the Progress Bar behavior for details.

Moving an overlay with the mouse

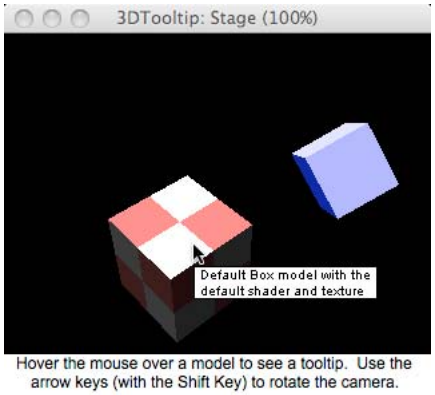
You can move an overlay around by changing the value of its loc property. To see an example where an overlay follows the position of the mouse pointer, download the movie DragShaderColor.dir and launch it.



A 2D sprite and a 3D overlay with the same image move together to give the appearance of a single element

Rollover tool tip

You can use overlays and backdrops to display text that is generated on the fly and placed at arbitrary positions in the 3D camera view. To see this technique used to create rollover tool tips, download the movie 3DTooltip.dir and launch it.



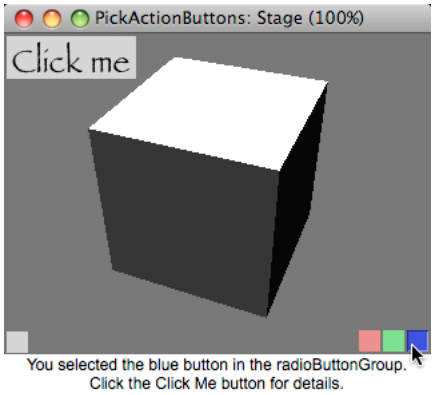
The movie 3Dtooltip.dir generates images for an overlay on the fly with the help of a text cast member

In the 3DTooltip.dir movie, the text for the tool tips is stored in the userData list of each of the models.

2D buttons in a 3D world

The easiest way to create buttons in the 3D world is to use ordinary 2D sprites. You can place these on top of a 3D sprite that is set to display directToStage. The 3D sprite appears in front of the buttons, but the buttons receive any mouse events such as mouseEnter or mouseDown. To make the buttons visible, you can tell the 3D camera view to display an overlay at the same location as the 2D button.

To see this in practice, download the movie PickActionButtons.dir and launch it.



2D sprites in higher-numbered channels tell the 3D camera view to display overlays based on user interaction

The movie PickActionButtons.dir contains a set of scripts that can be applied to buttons in either a 2D or 3D environment.

The overlay button behavior set

These include the following behaviors:

- Overlay Push Button
- Overlay Toggle Button
- Overlay Radio Button

There are also two Parent Scripts. One or other of these scripts is used as the ancestor to each button behavior.

- Overlay Button: Creates an overlay in a 3D sprite if it happens to overlap the button
- Overlayer Button: Uses a sprite in the next highest-numbered channel for text or icon. (The Overlayer Button behavior is not demonstrated in this movie).

The PickActionButtons.dir movie also demonstrates the Pick Action behavior. This allows you to add script instances to models in much the same way that you add behaviors to 2D sprites. For more information on this technique, see "Picking" on page 242.

Lights

Lights illuminate the 3D world and the models in it. Without lights, the world exists, and actions can take place, but users see nothing. You can add lights to your 3D world in your 3D modeling application and with the Property inspector or with Lingo or JavaScript™ syntax.

You can find an overview of how light is simulated in a 3D world at "Simulated light" on page 41. Shockwave 3D provides the following types of light sources:

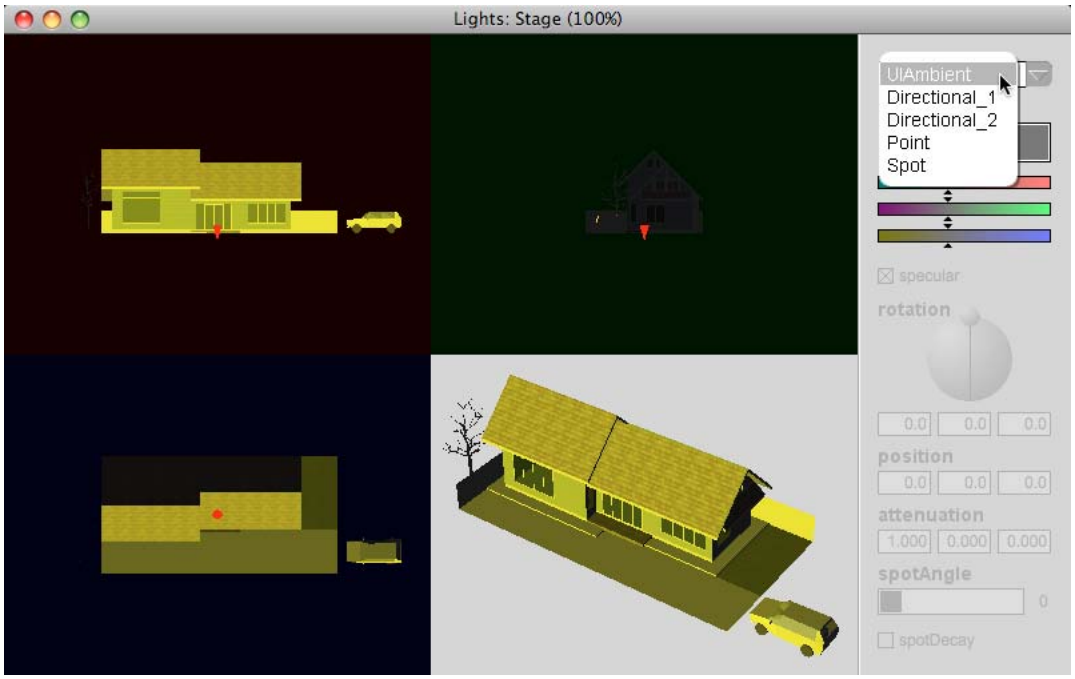
- "Ambient light" on page 123
- "Directional lights" on page 124
- "Point lights" on page 125
- "Spot lights" on page 126
- "Interactions with shaders" on page 123
- "How faces are lit" on page 127
- "Specular Light" on page 130

You can find an overview of the different types of light used in Shockwave 3D at "Light sources" on page 42. This section provides you details about how each type of light functions. It also explains how shaders and meshes react to light sources, and about creating the illusion that objects are shiny.

Note: *Lights in Shockwave 3D pass through all models and do not cast shadows. For more information on the limitations of the Shockwave 3D light simulation process, see "The shortcomings of lighting in Shockwave 3D" on page 44.*

Demo movie

To experiment with lights, download the movie Lights.dir and launch it.



Use the pop-up menu to select one of the five lights in the movie Lights.dir and set its properties

The movie Lights.dir shows four views into the same 3D world:

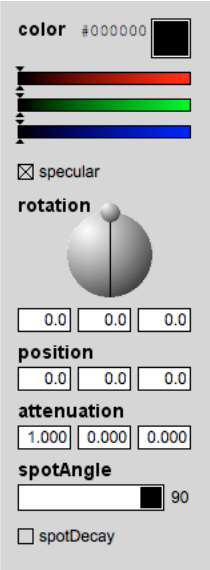
- Top left: Orthographic front elevation view down the x-axis
- Top right: Orthographic side elevation down the y-axis
- Bottom left: Orthographic plan view down the z-axis
- Bottom right: Perspective cavalier view

Note: *The z-axis points upwards.*

You can use the pop-up menu at the top right to select which light you wish to study. In the three orthographic views, a red cone shows the position and orientation of the selected light. The tip of the cone points in the same direction as the selected light. In the illustration above, the light("UIAmbient") is pointing directly downwards along the z-axis. You can drag the red cone around to change the position of the light in the world.

Light properties

The column at the right allows you to set all the properties for the selected light. All lights have the same seven properties. Most lights ignore at least some of the properties that you can set. Only #spot lights react to all the properties of a light object.



The control panel for light properties in the Lights.dir movie

Lights have five light properties and two node properties that can alter their effect on models:

- color: Determines the hue and the intensity of the light.
- specular: Determines if the light produces highlights when light rays bounce off a shiny surface. Ignored by #ambient lights.
- attenuation: Determines how the light gets dimmer with distance. Applies only to #point and #spot lights.
- spotAngle: Determines the angle of cone of light from a #spot light. Ignored by all lights except #spot lights.
- spotDecay: Determines whether the outside edge of the cone of #spot light is hard or soft. Ignored by all lights except #spot lights.
- rotation: (node property) Determines the direction in which the light is pointing. Ignored for #ambient and #point lights. For #directional lights, all light rays have the same rotation. For #spot lights, the rotation gives the direction of the rays at the center of the cone
- position: (node property) For #point and #spot lights, the position of the light in world space determines the angle at which the light's rays strike each face of each model. Ignored for #ambient and #directional lights.

Demo controls

In the Lights.dir demo movie, when you select a light in the pop-up menu, certain controls may appear grayed out. These are the controls that are ignored for that type of light. All controls work even if they are grayed out. You can use this to check that setting these properties has no effect on lights of the selected type.

You can set the color of a light by dragging the red, green, and blue sliders separately. If you hold the Shift key down while you are dragging, all three sliders adopt the same value. This will give you “white” light of varying intensity.

Drag the pinhead in the Rotation control to set the rotation of the selected light. Watch the front elevation view at the top left as you drag the pinhead. The red cone in this view will point in the same direction. This control allows you to rotate the pinhead around one hemisphere only. You cannot use this control to point the light towards the camera in top left view. Drag the red cone around in the orthographic views in order to change the position of the selected light. To change the attenuation, type numbers in to the appropriate fields. An attenuation of vector(0.0, 0.0, 0.0) is not permitted. If you enter such a value, the last acceptable value will be restored.

Click here for further reading on this topic.

Interactions with shaders

Lights interact with the shaders on models in order to produce the rendered image. Specific light properties interact with specific properties of #standard and #painter shaders.

The result of the interaction depends on the overlap between the light color and the color of the shader property. For instance, a light color of rgb("#FF9900") and a shader property color of rgb("#33FF66") result in a color of rgb("#339900"). In this case, the minimum of each color will be used.

Note: *The #engraver and #painter shaders ignore the color of lights altogether. Their diffuse property only reacts to the rotation of #directional lights, not their color.*

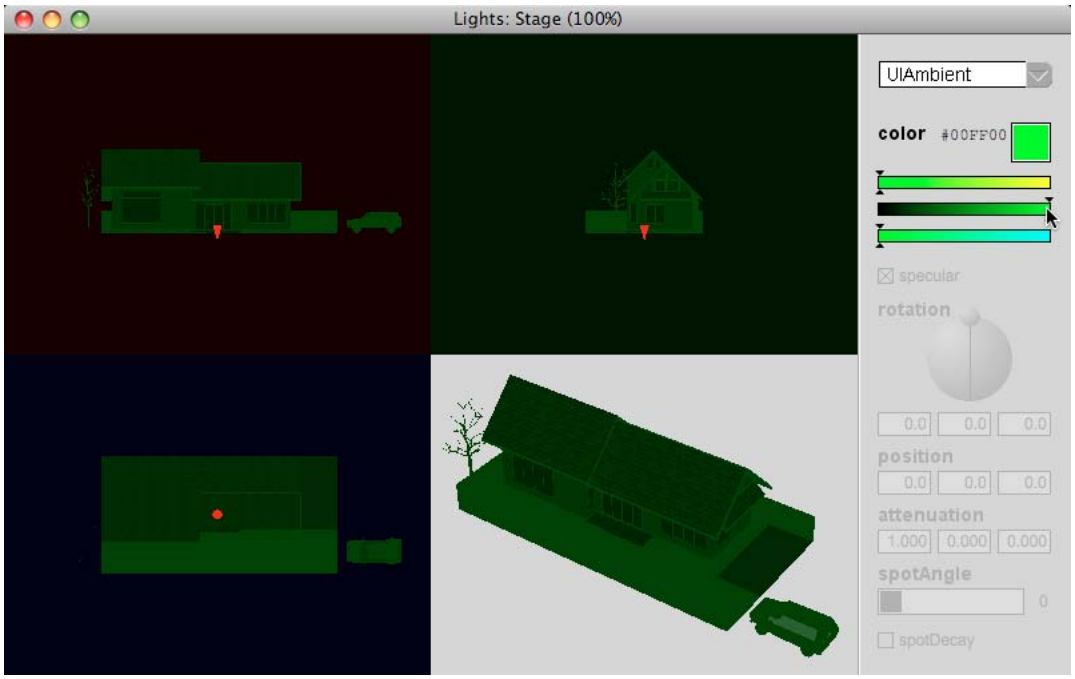
Ambient light

By default, every 3D cast member has one ambient light object: light("UIAmbient"). Generally, you do not need more than one ambient light. The ambient light determines how dark the shaded areas will appear.

By default, the UIAmbient light is black. This means that shaded areas will be 100% black. If you increase its brightness, shaded areas will appear gray. The position and orientation of the ambient light has absolutely no importance.

For a given shader, ambient light affects all faces equally, regardless of their orientation in space. If a scene is lit only with ambient light, there will be no change shading between surfaces. The scene will appear flat.

To experiment with lights, download the movie Lights.dir and launch it. The controls for the ambient light will be selected by default. You may wish to make the light("Directional_1") black in order to see the effect of the ambient light on its own.



Ambient light colors all surfaces equally, regardless of their orientation

The image above uses an unnatural green ambient light to illustrate how the color is applied equally.

Ambient property

The only active property for ambient light is light.color.

To get the most realistic effect in your 3D world, the ambient light must be the same color as your main directional light. Ambient lights ignore all other properties that you set for a light object.

Ambient color of #standard shaders

Only #standard shaders react to ambient light. All other shaders ignore it. Standard shaders have an ambient color property. This determines how the shader reacts to ambient light. If the ambient light is yellow (red + green) and the shader's ambient color is magenta (red + blue), then all faces with that shader will be tinted red. The green channel of the ambient light will be ignored, and there will be no blue light for the shader to react to.

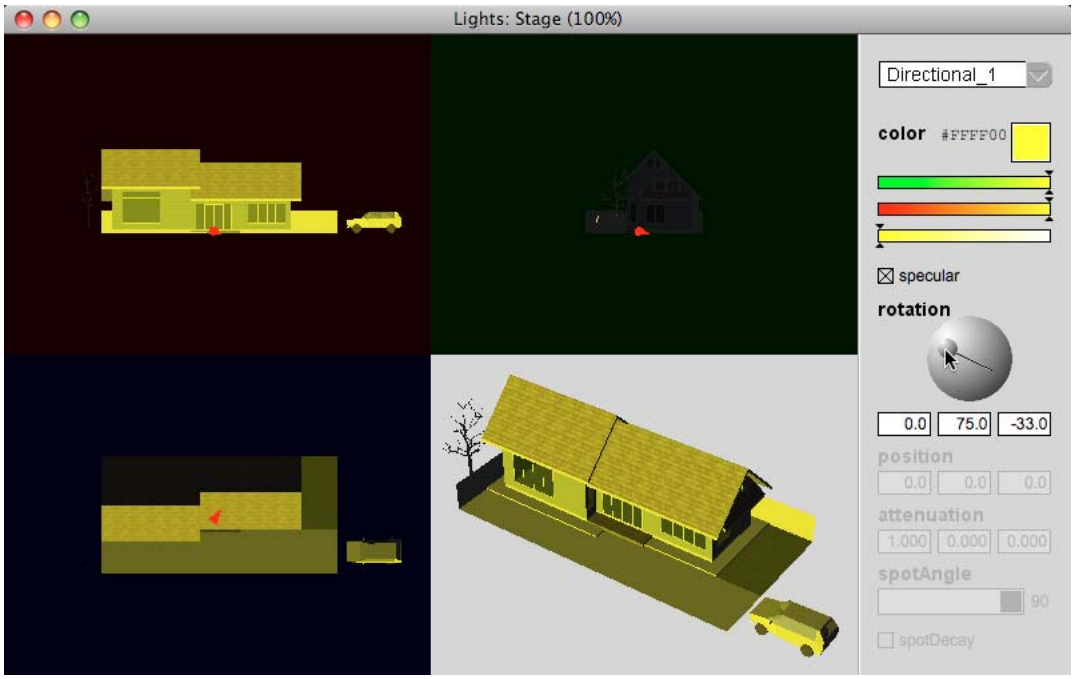
If the ambient light is a shade of gray (equal amounts of red, green, and blue), then all faces with the shader will be tinted magenta. Normally the ambient color of a #standard shader will be a shade of gray. The default value is rgb("#3F3F3F). For more information on the properties of shaders, see "Shaders and appearance modifiers" on page 130.

Directional lights

A directional light acts as if its rays travel in parallel from infinity in the direction of the light's negative z-axis (from positive to negative). A directional light creates an effect of light and shade that enhances the illusion of three dimensions.

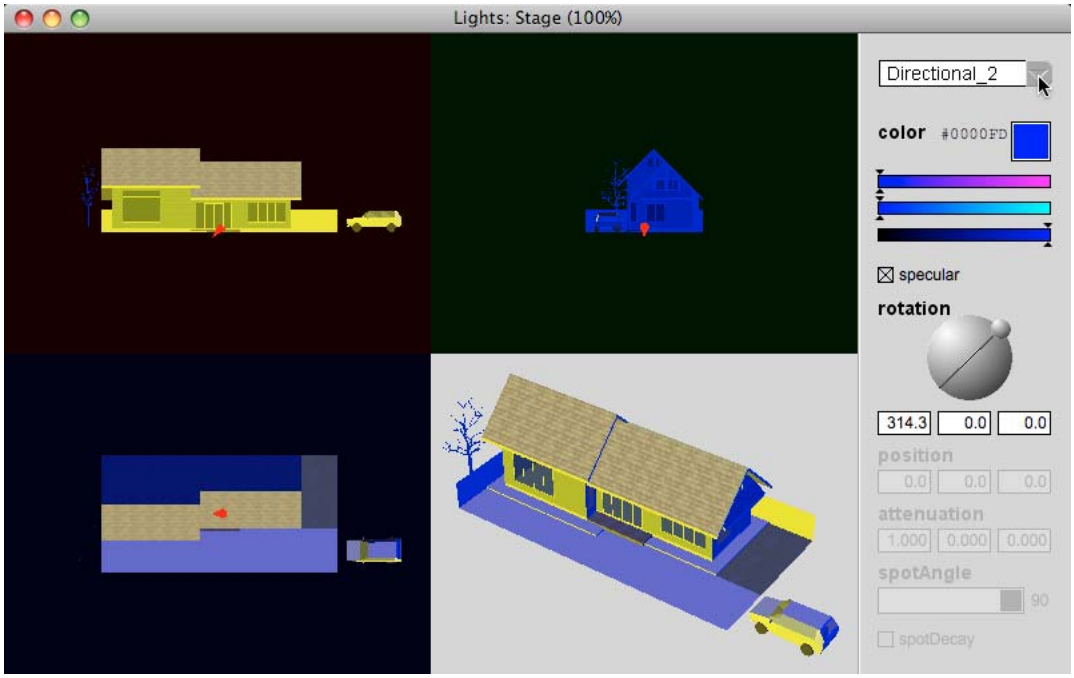
Changing the position of a directional light has no effect. Changing the rotation of a directional light will change the way it illuminates each triangular face of the models in the 3D world. To get the most realistic effect in your 3D world, the ambient light needs to be the same hue as your main directional light.

To experiment with directional lights, download the movie Lights.dir and launch it.



Directional lights create an effect of light and shade, due to the angle of incidence of the light on each surface.

The movie contains two directional lights: Directional_1 and Directional_2. You can set these to different colors at different rotations to observe the effect.



Directional lights at different rotations have different effects on different faces

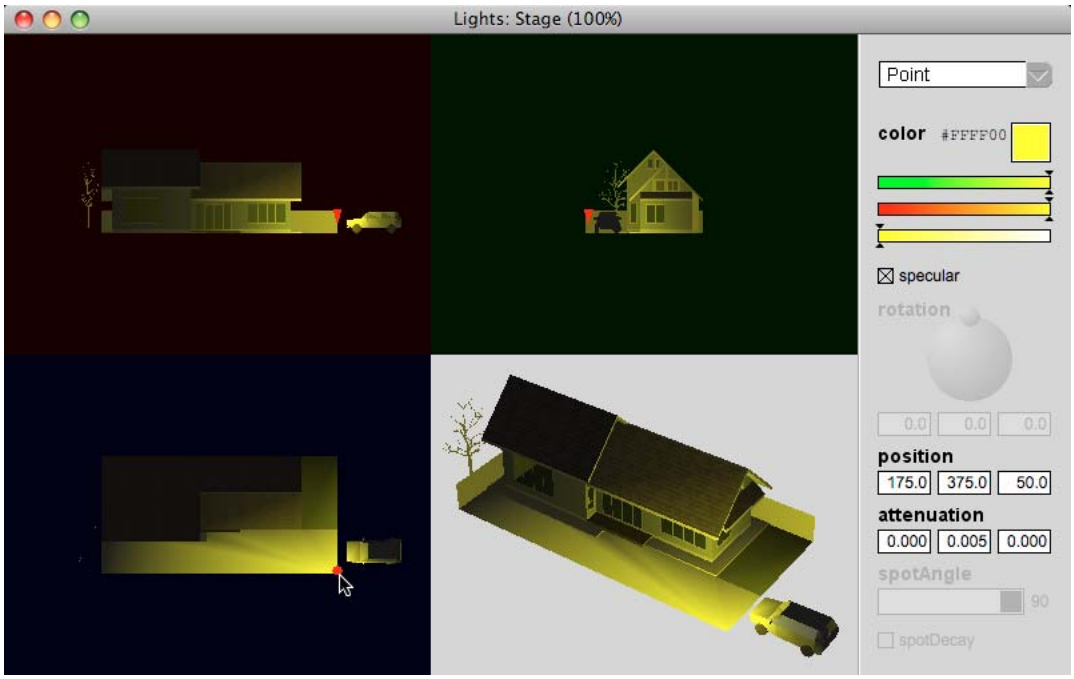
Directional light properties

You can set three properties for directional lights objects:

- light.color
- light.specular (See also "Specular Light" on page 130)
- node.rotation

Point lights

A point light sends out light rays in all directions from a given position in 3D space. A point light's attenuation property can be set so that its effect diminishes with distance. Changing the rotation of a point light has no effect. Changing the position of a point light will change the angle at which its rays fall on each triangular face of the models in the 3D world. To experiment with a point light, download the movie Lights.dir and launch it.



Point lights illuminate the world like a naked electric bulb, sending out light rays in all directions from a point.

Before selecting the Point light in the pop-up menu, you may wish to select the UIAmbient and Directional_1 lights, and set their color to black. This will allow you to see the effect of the point light on its own.

Point light properties

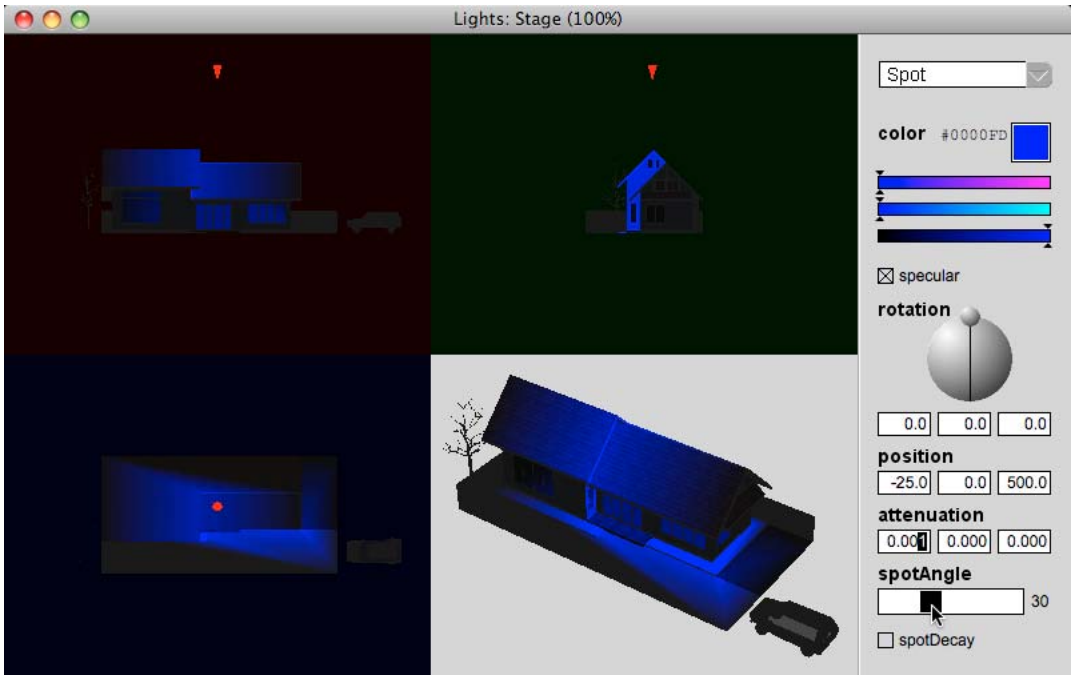
You can set four properties for directional lights objects:

- light.color
- light.specular (See also "Specular Light" on page 130)
- node.position
- node.attenuation

The attenuation property takes the form of a vector because it has three numerical values. Its default value is vector(1.0, 0.0, 0.0). This makes it maintain its brightness regardless of its distance from the target surface. To make a Point light act more like a real light bulb, try setting its attenuation to a value such as vector(0.0 0.01, 0.0) or vector(0.0 0.0, 0.0001). These values make its effect less noticeable the further away it is from a given surface.

Spot lights

A spot light sends out a cone of light from a given position in 3D space. You can set a spot light's attenuation property so that its effect diminishes with distance. You can set its spotAngle property to make its cone of light wider or narrower. Changing the rotation of a spot light changes the direction of the cone of light. Changing the position of a point light determines which faces of the models in the 3D world its rays will fall on. To experiment with a spot light, download the movie Lights.dir and launch it.



Spot lights illuminate the world like a flashlight, sending out light rays in a cone from a point.

Before selecting the Spot light in the pop-up menu, you may wish to select the UIAmbient and Directional_1 lights, and set their color to black. This allows you to see the effect of the spot light on its own.

Spot lights are the most complex light source in Shockwave 3D. Calculating the effect of a spot light requires more computing time than for any other type of light. So, use spot lights only when their additional features are essential. As you experiment with the spot light in the Lights.dir movie, you may notice that certain triangular faces light up more readily than others. To know more about why this happens and the spotDecay property, see "How faces are lit" on page 127.

Spot light properties

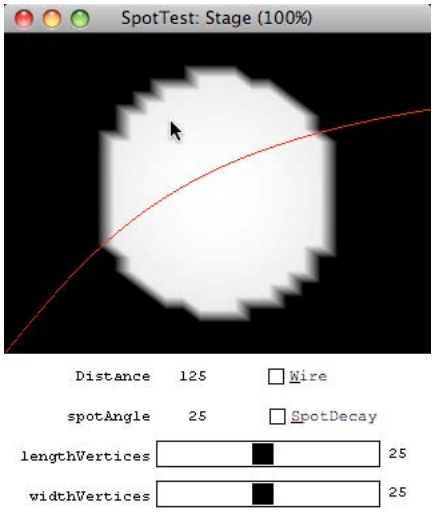
You can set seven properties for directional lights objects:

- light.color
- light.specular (See also "Specular Light" on page 130)
- light.attenuation
- light.spotAngle
- light.spotDecay (See also How faces are lit)
- node.rotation
- node.position

How faces are lit

When you use a spot light, sometimes faces that you expect to be lit remain dark. If this happens, you may need to divide the faces up into smaller triangles. A face is lit if at least one of its vertex points is visible to a light source. If all vertex points are lit, then the brightness is spread evenly across the face.

If only one or two vertex points are lit, then the other vertices appear dark; the lighting creates a gradient across the face. This is only an issue with spot lights. A spot light produces a cone of light. One vertex of a face may fall inside the cone while the other vertices fall outside it. If a face is particularly big and if it is directly in front of the spot light, the face can fail to light up at all. The cone of light can miss all three vertex points, even if the middle of the triangular face is directly in line with the beam. To see a demonstration of this effect, download the movie SpotTest.dir.



A spot light lights up faces whose vertices are within the cone of light

The SpotTest.dir movie shows a plane model that is perpendicular to the camera and to a spot light. Using the mouse you can change two parameters:

- The distance from the plane to the spot light by moving the mouse pointer horizontally.
- The spotAngle of the spot light by moving the mouse pointer vertically.

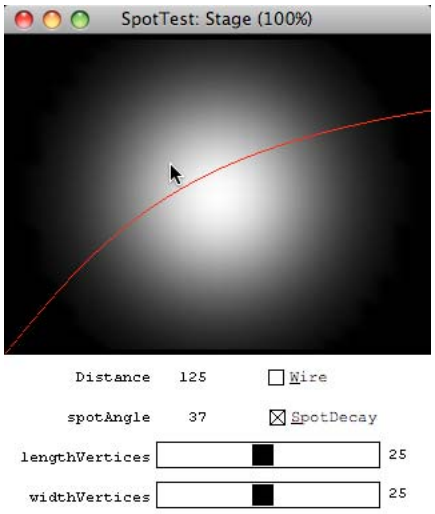
The red line across the 3D sprite indicates the combination of distance and spotAngle at which the corners of the plane are just lit. If the mouse pointer is above or to the left of this line, the corners of the plane model are not lit, and the spot light fails to light the plane completely.

When you launch the movie, the plane's modelResource has only two vertices horizontally and two vertices vertically. In other words, its only vertices are in the corners. If you move the mouse pointer below the red line, the plane model will be fully lit. If you move it above the red line, it will not be lit at all.

You can create a more graduated response by dividing the plane into smaller triangles. The screenshots above shows which triangular faces will be lit by a particular cone of light from a spot light.

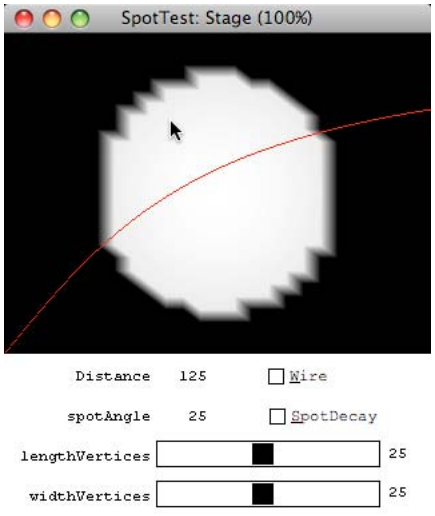
SpotDecay

Even with fairly small faces, the edges of the cone look very hard and rough. You can soften the edges by setting the spotDecay property of the spot light to TRUE. The following screenshots show the same distance and spotAngle settings as in the screenshots above, but with the spotDecay set to TRUE.



Setting a spot light's spotDecay property to TRUE softens the edges of its beam

If you look closely at the two screenshots that show the wireframes, you will see that the lit area is the same, but outer edges have been reduced to darkness. To give the effect of a soft-edged cone with the same illuminating power from the same distance, you can increase the spotAngle of the spot light, as the comparison below shows.



To illuminate the same area with spotDecay set to TRUE, increase the spot

To increase the number of faces displayed by a model, you can use the Subdivision Surfaces modifier (SDS). See "Modifiers" on page 50 for more details.

Using shader layers to simulate and enhance lighting

You can create a brightly-lit 3D world with no lights at all by using baked textures and shaders whose emissive property is set to rgb("#FFFFFF").

A 3D world with baked textures and no lights

You can also use the multiple layers of a shader object to provide additional cues about light sources. This article provides you with an example of what you can achieve.

Specular Light

Specular light means light that is reflected from a mirror-like surface. (The Latin word for “mirror” is “specularis”). Specular highlights are created by interaction between a light and a shader. All lights except ambient lights have a specular property. This is set to TRUE by default. If you set it to FALSE for a particular light, that light stops being “shiny”. It will not create any specular highlights on any shiny objects.

Directional lights, point lights, and spot lights all send out rays of light that have a direction. These types of light can produce both diffuse light and specular light. The color and intensity of the diffuse light for a given surface depends only on the angle that the surface makes to direction of the light's rays. For specular light, the angle of observation is also important. Specular light requires more calculations. It simulates the shininess of objects, and so makes them look more real.

Specular light does not make objects look more three-dimensional. If your 3D movie is running on a slow computer, you can save processor cycles by:

- Setting individual light objects so that they trigger no specular calculations
- Setting individual shaders not to react to specular light

Using specular light

To get specular light to work in Shockwave 3D, several conditions need to be met:

- At least one light in the scene must have its light.specular property set to TRUE (the specular property of ambient lights is ignored).
- The shader must have a non-zero value for its shader.shininess property.
- The shader's shader.specular color must have a non-zero value for all channels (red, green and blue).
- The model must be oriented so that the rays of light from the specular light source bounce from at least one of the model's faces into the camera.

Note: *If you use a color such as rgb("#FF00000") instead of rgb("#FF0101") for the shader's specular property, the specular feature will no longer function. All color channels (red, green and blue) for the shader's specular property must have a non-zero value.*

Shininess

If a shader's shininess property is set to 0 specular highlights do not appear. If you set its shininess value is set to 1, the shader will show a large highlit area. Increasing the value to a maximum of 100 will in fact decrease the area of the highlight.

Performance

Calculating the shape and position of specular highlights requires significant computer processing power. If you are delivering an application for low-end computers, consider limiting your use of specular highlights.

Shaders and appearance modifiers

A shader is a programming object that defines how the surface of a 3D model appearance reacts to virtual lights.

Shaders define the model’s surface colors and reflectivity. You can use just one shader or more than one. Each mesh in a model resource can have its own shader. For example, a box may have six different shaders, one for each mesh (a box is actually composed of six plane meshes carefully arranged).

If you do not specify a shader, the default #standard shader is used. If the shader's properties are modified, the change affects all models that use that shader. Models that are created with script are assigned the standard shader. You can replace the default shader of a model with any of the other types of shaders.

The following are the different types of shaders:

- "Standard shaders" on page 131
- "Painter shaders" on page 135
- "Engraver shaders" on page 137
- "Newsprint shaders" on page 138
- "Toon modifier" on page 138
- "Inker modifier" on page 140

You can use a #standard shader to create a realistic or imaginary surface for your models. The #painter, #engraver and #newsprint shaders give more artificial effects.

Director also provides two modifiers that you can add to a 3D model to customize its appearance. These both give a cartoon-style quality to your 3D world.

Diffuse property for shaders

All four different types of shaders share one property: #diffuse. You can set the value of this property to any color. However, the #diffuse property reacts differently to light for different shader types. For #standard and #painter shaders, the #diffuse property determines which color of light is reflected from the surface of the model.

Imagine that a light whose color is set to yellow (rgb("#FFFF00")) shines on a #standard or #painter shader whose #diffuse property is set to cyan (rgb("#00FFFF")). The surface of the model appears green (rgb("#00FF00")) as this is where the colors of the light and the shader's #diffuse value overlap. The amount of green light reflected from the surface from each face depends on the orientation of that face with respect to the light.

Faces turned away from the light shows a darker shade. For #newsprint and #engraver shaders, the #diffuse property behaves more like the #emissive property for #standard shaders. If the #diffuse property of a #newsprint or #engraver shader is set to cyan (rgb("#00FFFF")), then that is the color that the surface of the model will appear. The shading on the sides facing away from light a directional light shows a smaller number of pure cyan pixels and more black pixels. To experiment with this property, see the demo movie at "Shader types" on page 35.

Standard shaders

A #standard shader has five different types of properties, which can be divided into two categories. The properties in the first category affect the shader as a whole:

- Color properties: ambient, diffuse, emissive, and specular.
- Surface properties: shininess, blend, transparent, renderStyle, flat, and useDiffuseWithTexture.

  The surface properties define how the shader interacts with texture images. A shader can display up to 8 different layers of texture. In practice, only one to five layers are normally used. To imitate certain smooth materials like plastic or metal, you may not need to use any textures at all.
- Texture properties: texture, diffuseLightMap, reflectionMap, glossMap, specularLightMap
- Layer one properties: blendFunction, blendSource, blendConstant, textureMode, textureRepeat, textureTransform, wrapTransform

- Layer property lists: blendFunctionList, blendSourceList, blendConstantList, textureList, textureModeList, textureRepeatList, textureTransformList, and wrapTransformList

Color properties

A #standard shader has four color properties. These properties determine how the shader as a whole reacts to light sources.

Director uses three different kinds of light sources:

- "Ambient light" on page 123
- Diffuse light: "Directional lights" on page 124, "Point lights" on page 125, and "Spot lights" on page 126
- "Specular Light" on page 130

See "Interactions with shaders" on page 123 for a comparison table. The ambient, diffuse, and specular properties of a shader determine how the shader reacts to each of these different types of lighting. For example, imagine a shader with the following properties:

```
vShader = member("3D").newShader("Example")
vShade.ambient = rgb("#0000FF")
vShade.diffuse = rgb("#FFFFFF")
vShade.specular = rgb("#808080")
```

This shader only reflects the blue channel of the ambient light. It reflects all of the light from diffuse sources (directional, point, and spot lights). It reflects 50% of all specular light from diffuse sources whose specular property is set to TRUE. In other words, it appears blue on faces that are pointing away from any diffuse light source, and it shows dim specular highlights where a light bounces off its surface directly towards the camera.

Note: *If there are one or more textures attached to the shader, and the shader's useDiffuseWithTexture property is set to FALSE, then the shader does not react to diffuse light. The useDiffuseWithTexture property is FALSE by default on all new shaders.*

Emissive

The emissive property allows you to make a shader glow with its own light. If you want to use a baked texture on a model, then you must set the emissive property of all its shaders to rgb("#FFFFFF").

You can see a demo where the emissive property of a shader is changed to highlight the model under the mouse at "Pick Action behavior" on page 244.

Surface properties

Surface properties are applied to the whole shader.

- shininess changes how shiny the surface of the model appears to be. A value of 0 creates a matte surface. Values from 1.0 and 100.0 make the surface shiny to varying degrees, with 1.0 being the most shiny and 100.0 being the least. A value greater than 0.0, but less than 1.0 is treated is if it were 1.0. The default value for new shaders is 30.0. The effect of shininess will only be apparent if the specular color of the shader is not black. See also the note below concerning the flat property.
- transparent and blend determine whether you can see through the surface of the model. The transparent property can be set to TRUE or FALSE; the blend property can have any value between 0.0 and 100.0. If transparent is FALSE, then the model will appear opaque, regardless of the value of blend. If transparent is TRUE, then the lower the value of blend, the more transparent the model will appear. If blend is set to 100.0 then the model will appear fully opaque, even if transparent is set to TRUE. See the note below about issues with more than one semi-transparent models in a scene.

- renderStyle can take three values: #fill, #wire or #point. A `renderStyle` with a value of #fill will show the surface of all faces. With a value of #wire, only the edges of faces will be shown. When set to #point, only vertex points will be shown as a cloud of dots. This can be useful when debugging.

  Note: *A creative use of the #point render style is to simulate a "Beam me, Scotty" type of animation. You can set the* `renderStyle` *of a character to point, and then use either the #LOD or #SDS modifier to change the apparent number of vertex points used by the model. Reducing the number of visible vertex points will make the model gradually disappear.*

- flat determines what shading algorithm is used to draw each face. If `flat` is TRUE, then each face will appear as a flat triangle. If it is FALSE, curved surfaces will appear to be curved.curved surfaces will appear to be curved. If performance takes priority over appearance, then you may want to set `flat` to TRUE, to reduce the processing time for each face. See Flat shading and smooth shading for more details.

  Note: *If #flat is set to TRUE, specular highlights do not look natural. It makes sense to set the #shininess of flat shaders to 0 and to set the #specular property of lights to FALSE.*

- useDiffuseWithTexture allows you to get or set whether the diffuse color is used to modulate the texture (TRUE) or not (FALSE). When set to TRUE, this property works in conjunction with the blendFunction and blendConstant properties to determine what proportion of the shader's diffuse color and what proportion of the color in the texture is used for each point on the surface of the model.

Texture properties

To display images on the surface of models, you can attach textures to shaders. You may want to use as many as five different textures, each with a different purpose. These textures are stored in the different layers of a shader. When you create a new shader, layer 1 contains the default red-and-white checkered texture, and all the other layers are empty.

- In layer 1, the shader.texture defines the original color of the pixels on the surface of the model, in the absence of any effects of lights or reflections.
- In layer 2, the shader.diffuseLightMap defines a diffuse light map to be used with the shader, so that differences in how the environment is lit can be made visible.
- In layer 3, the shader.reflectionMap defines what colors in the environment will be reflected in the shader. It is used to create metallic and chrome-like effects. You can see examples of what can be achieved with a reflectionMap here:
  - http://www.director3d.de/reflectionmap.htm
  - http://www.fce.at/7senses/shock/shock10.htm. Press the spacebar to show the configuration options.
- In layer 4, the shader.glossMapdefines which areas of the surface are shiny and which are matte. This texture is used as a mask for specular highlights. You can use this to simulate a shiny material which has dirt on it in some places. You must use a grayscale image for this texture.
- In layer 5, the shader.specularLightMap defines the color of the specular highlights at different points on the surface. When you set the value of any of these properties to a texture object, Director automatically set the appropriate values in the blendFunctionList and textureModeList for the given layer, at the same time as it adds the texture. When you set the `shader.reflectionMap` property to a texture blendSourceList and blendConstantList for layer 3 are also modified.

Layer properties

For each layer of texture in a shader, you can specify:

- How the texture in that layer blends with the layer below
- What method is used to map the texture onto the faces of the mesh
- How the texture is positioned and scaled and repeated over the faces of the mesh.

The values for the layer properties are stored in linear lists within the shader. (See Layer property lists, below).

You can access the properties in two different ways:

- `shader.layerPropertyList[n]` provides access to the layerProperty in the nth layer of the shader
- `shader.layerProperty` provides a shortcut for referring to `shader.layerPropertyList[1]`

When setting a property, you can set it either for one particular layer, or use a shortcut to set it for all layers simultaneously. For example, to set the texture in the third layer of the shader aShader to aTexture, use:

```
aShader.textureList[3] = aTexture
```

To set the texture in all eight layers of the shader to VOID with one command, use:

```
aShader.textureList = VOID
```

Blending with the layer below

Each texture layer can blend with the layer below. Layer 1 can blend with the diffuse color for the shader, if useDiffuseWithTexture is set to TRUE. If useDiffuseWithTexture is set to FALSE, white is used instead. Each of the other layers blends with the layers below.

See blendFunction, blendSource, and blendConstant for details.

Mapping onto the mesh

The texture in a given layer can be mapped on to the mesh in a variety of different ways. Some of these reflect the geometrical shape of the mesh, others are projections from the environment around the mesh.

- `#none` uses the texture coordinate values defined for the mesh. See "Mapping a texture to a mesh resource" on page 150 for more details of texture mapping.
- `#wrapPlanar`, `#wrapCylindrical`, `#wrapSpherical` consider the mesh to have certain geometrical properties, and map the texture onto the mesh accordingly. If you choose one of these values for a given layer, then you can use the wrapTransform for that layer to rotate the texture to suit the geometry.
- `#reflection` projects the texture onto the mesh from a fixed orientation to simulate objects in the environment which are reflected in the surface of the model.
- `#diffuseLight`, `#specularLight` create light mapping texture coordinates for each vertex in the mesh. See textureModeList for more details regarding these values.

Positioning on the mesh

You can use textureTransform to rotate, position and scale the texture relative to the model's mesh. With certain settings for textureTransform, the texture image may be mapped to only a part of the surface covered by the shader. You can use textureRepeat to determine whether the texture tiles across the mesh to fill the surrounding blank space, or whether the edges of the texture are smeared out to fill the blank space. Only five properties of the textureTransform for a given texture layer will have any effect:

- Use `textureTransform.position.x` and `textureTransform.position.y` to move the origin point for calculating uv positioning. See "Mapping a texture to a mesh resource" on page 150for more details of texture mapping. The values for uv coordinates range between 0.0 and 1.0. As a result, only the decimal part of the x and y position co-ordinates will be taken into account. Setting textureTransform.position.x to 0.321 or 654.321 has exactly the same effect.
- Use textureTransform.scale.x and textureTransform.scale.y to stretch or shrink the texture.
- Use textureTransform.rotation.z to rotate the texture around the uv point [0.0, 0.0] at the bottom left hand corner of the texture image.

If you want the texture to appear to rotate around its center, you can set the textureTransform.position to vector(0.5, 0.5, 0), to place the uv point [0.0, 0.0] in the center of the shader. You will probably have to cut the texture into four equal parts and swap the parts around to obtain the desired effect.

Adjusting the wrapping

If you set the textureMode of a given layer #planar, #spherical, or #cylindrical, then you can use the wrapTransform for that layer to rotate the texture so that it fits the geometry of the mesh correctly. Only modifying the rotation or axisAngle properties of the wrapTransform will have any effect. Altering the position or scale of a wrapTransform will make no difference to the appearance of the shader.

Note: *`aShader.wrapTransformList[textureLayerIndex]` has an effect only when `aShader.textureModeList[textureLayerIndex]` is set to #planar, #spherical, or #cylindrical.*

Layer property lists

The values for all the texture layer properties are stored in linear lists as part of the shader object.

- blendFunctionList
- blendSourceList
- blendConstantList
- textureList
- textureModeList
- textureRepeatList
- textureTransformList
- wrapTransformList

To get or set a given property for a particular layer, use: `shader.layerPropertyList[n].`

For example:

```
aShader.blendSourceList[3] = #alpha
put aShader.textureRepeatList[4]
```

The objects that store layer properties for a shader are not true Lingo lists. You cannot set the value of one of these lists to a Lingo list. This command gives a Lingo list as the output:

```
put member("3D").shader(1).textureRepeatList -- [1, 1, 1, 1, 1, 1, 1, 1]
```

However, the command below will lead to a "Wrong Type" script error:

```
member("3D").shader(1).textureRepeatList = [1,1,1,1,1,1,1,1]
```

To set the values in all 8 layers of the shader to a given value with one command, use:

```
aShader.layerPropertyList = aValue
```

For example:

```
member("3D").shader(1).textureRepeatList = 1
```

Painter shaders

To explore the properties of the Painter shader, download and launch the movie ShaderTypes.dir.

See "Shader types" on page 35for details on how Painter shaders interact with lights. See "Toon modifier" on page 138 for a different way to achieve similar results.

Styles and properties

A Painter shader can have one of the following styles:

- #blackAndWhite
- #toon
- #gradient

A Painter shader also has six other properties. None of the styles reacts to all of these properties:

- shadowPercentage (#blackAndWhite and #toon styles only)
- highlightPercentage (#toon style only)
- shadowStrength (#toon and #gradient style only)
- highlightStrength (#toon and #gradient style only)
- diffuse (#toon and #gradient style only)
- colorSteps (#gradient style only)

shadowPercentage and highlightPercentage

The properties shadowPercentage and highlightPercentage are linked. Their total value does not exceed 100. If the value of one of these properties is set to a higher value, the other will be reduced accordingly. For example, after the following code has been executed, the value for the highlightPercentage of the Painter shader is reset to 30, to ensure that the total of the two percentage values does not exceed 100.

```
vPainterShader = member("3D").shader("Painter shader")
vPainterShader.highlightPercentage = 70
vPainterShader.shadowPercentage = 70
put vPainterShader.highlightPercentage
-- 30
```

The total of the two percentage values can, however, be less than 100.

```
vPainterShader.shadowPercentage = 30
put vPainterShader.highlightPercentage
-- 30
```

Black and white

A Painter shader with a style of #blackAndWhite shows brightly lit areas in white and shaded areas in black. It reacts to only one property: shadowPercentage. This can take a value between 0 and 100. The higher the value, the higher the cut-off point between light and dark. A high value gives a very small white area and a big area of black.

Altering highlightPercentage may reset the value of shadowPercentage and so it may appear is if the shader is reacting to changes in highlightPercentage. However, only the automatically modified value of shadowPercentage actually has any effect.

Colors for toon and gradient styles

The #toon and #gradient styles react to ambient light. Instead of being completely black, the shadow areas for these styles are calculated from the ambientColor for the 3D member. The shadow color is calculated as follows:

```
shadowColor = a3DMember.ambientColor / 4 * shader.shadowStrength
```

For example, if the ambientColor for the 3D member is rgb(64, 64, 192) and the shadowStrength for the shader is 2.5, the shadow color will be rgb(40, 40, 80). Values for red, green and blue will be pinned to 255. If the ambientColor is rgb(1, 2, 4) and the shadowStrength is 512.0, then the color of the shadow area will be rgb(128, 255, 255). The color of the bright area depends on three values:

- The color of the highest priority non-ambient light (see "Shader types" on page 35 for details)
- The diffuse color of the Painter shader
- The highlightStrength of the Painter shader

The color is calculated according to the following equation, with individual red, green and blue values pinned at 255:

```
brightColor = (light.color AND shader.diffuse) *
shader.highlightStrength
```

Toon

A toon-style shader reacts to the value of five properties:

- shadowPercentage
- highlightPercentage
- shadowStrength
- highlightStrength
- diffuse

The sum of shadowPercentage and highlightPercentage may be less than 100. Unless both values are equal to 50, the shader produces 3 bands of color:

- A bright area with the color (light.color AND shader.diffuse) * shader.highlightStrength
- A middle area with the color a3DMember.ambientColor / 4
- A shadow area with the color a3DMember.ambientColor / 4 * shader.shadowStrength

If shadowStrength is at its default value of 1.0, then the last two areas will have the same color. If shadowStrength is less than 1.0, the third area appears as a darker hue. If you decide to make shadowStrength greater than 1.0, the areas where the shading is supposed to be the darkest appears brighter than the mid zone, which is probably not desirable.

Gradient

A Painter shader with a style of #gradient reacts to four properties:

- shadowStrength
- highlightStrength
- diffuse
- colorSteps

Any faces that are facing away from the highest priority light takes on the shadow color, as described above. All other faces are assigned to one of the color steps depending on how much directional light falls on the face. The brightest faces are colored with the highlight color, as described above. The intermediate faces receive a blend of the highlight and the shadow color, proportional to their exposition to the directional light. The result is a series of bands of color, with a gradient from the highlight color to the shadow color.

Engraver shaders

To explore the properties of the Engraver shader, download and launch the movie ShaderTypes.dir.

The #engraver shader gives a monochrome image with old-fashioned cross-hatchings. See "Shader types" on page 35 for details on how Engraver shaders interact with lights.

Properties

An Engraver shader has four properties:

- diffuse (color)
- brightness (float 0.0 - 100.0, default 0.0)
- density (float 0.0 - 100.0, default 40.0)
- rotation (float 0.0 - 360.0, default 0.0)

The value for the diffuse color determines the highlight color for the shader. Brightness indicates the amount of white blended into the shader. Density adjusts the number of lines used to create the shading. Low values result in few, very thick lines. High values result in many very fine lines. Rotation indicates the orientation of the lines.

Note: *Changing the rotation of an Engraver shader over time makes the model to which the shader is attached appear to buzz. This can be useful for creating a rollover effect, or for making an object look as if contains some magic power.*

Newsprint shaders

To explore the properties of the Newsprint shader, download and launch the movie ShaderTypes.dir.

The #newsprint shader allows you to create monochrome images similar to photos in a newspaper. See "Shader types" on page 35 for details on how Newsprint shaders interact with lights.

Properties

A Newsprint shader has three properties:

- diffuse (color)
- brightness (float 0.0 - 100.0, default 0.0)
- density (float 0.0 - 100.0, default 45.0)

The value for the diffuse color determines the highlight color for the shader. Brightness indicates the amount of white blended into the shader. Density adjusts the number of dots and lines used to create the shading. Low values result in thicker, shorter lines. Values above 8.0 result in many very fine lines.

Toon modifier

The #toon modifier gives a cartoon style of rendering to the model's surface.

Styles and properties

The Toon modifier has a #style property that can be set to one of three values:

- #blackAndWhite
- #toon
- #gradient

It also has 12 other properties. Only the #toon style reacts to all of these properties.

- shadowPercentage (#blackAndWhite style only)
- highlightPercentage (#toon style only)

- colorSteps
- shadowStrength (ignored by #blackAndWhite style)
- highlightStrength(ignored by #blackAndWhite style)
- lineColor
- creases
- creaseAngle
- boundary
- lineOffset
- useLineOffset

Note: *The Inker modifier features lineColor, creases, creaseAngle, boundary, lineOffset and useLineOffset properties which function in an identical way to these properties of the Toon modifier. See "Inker modifier" on page 140 for more details on how these properties work.*

shadowPercentage and highlightPercentage

The properties shadowPercentage and highlightPercentage are linked. Their total value may not exceed 100. If the value of one of these properties is set to a higher value, the other will be reduced accordingly. For example, after the following code has been executed, the value for the highlightPercentage of the Toon modifier is reset to 30 to ensure that the total of the two percentage values does not exceed 100.

```
vModel = member(1).model(1)
vModel.toon.highlightPercentage = 70
vModel.toon.shadowPercentage = 70
put vModel.toon.highlightPercentage
-- 30
```

The total of the two percentage values can, however, be less than 100.

```
vModel.toon.shadowPercentage = 30
put vModel.toon.highlightPercentage
-- 30
```

Note: *The shadowPercentage property only has an effect on Toon modifiers with a style of #blackAndWhite. The higlightPercentage property only has an effect on Toon modifiers with a style of #toon.*

colorSteps

This can take the discrete integer values 2, 4, 8 or 16. If you attempt to set it to a value less than 2, a "Value out of range" script error occurs. If you set it to an integer higher than 2, it takes the next lowest valid value. For example:

```
vModel = member(1).model(1)
vModel.addModifier(#toon)
vModel.toon.colorSteps = 15
put vModel.toon.colorSteps
-- 8
```

A Toon modifier with a style of #gradient creates the number of bands of color defined by the value of colorSteps. If the Toon modifier has a style of #blackAndWhite or #toon, it uses these same bands of color as boundaries between the highlit and shaded areas. The selection of which boundary to use depends on the value of shadowPercentage (for the #blackAndWhite style) or highlightPercentage (for the #toon style). These percentage values change the shading in a stepwise fashion, rather than gradually.

shadowStrength and highlightStrength

For Toon modifiers with a style of #toon or #gradient, these properties determine how much white or black is added to the shadow or highlight areas. A value of 0.0 or below makes the given area completely black; a value of 1.0 gives it its natural color, and a value of 5.0 or above makes the area completely white.

For Toon modifiers with a style of #toon, setting the shadowStrength to a value other than 1.0 may result in 3 bands getting created. These properties are ignored by Toon modifiers with a style of #blackAndWhite. lineColor, creases, creaseAngle, boundary, lineOffset and useLineOffsetSee "Inker modifier" on page 140 for more details on how the lineColor, creases, creaseAngle, boundary, lineOffset, and useLineOffset properties work.

Inker modifier

The Inker Modifier is a simpler version of the Toon modifier. It allows you to add creases, boundary lines, and silhouette lines to the rendered image of a model. These features are shared with the Toon modifier. The Toon modifier provides additional shading features. This article describes the creases, boundary and silhouette features shared by both the Toon and the Inker modifier.

For information on the additional features of the Toon modifier, see "Toon modifier" on page 138.

Properties

The Inker modifier and Toon modifier share the following properties:

- lineColor (color)
- silhouettes (boolean)
- creases (boolean)
- creaseAngle (float -1.0 - +1.0, default 0.01)
- boundary (boolean)
- lineOffset (float -1.0 - +1.0, default -2.0)
- useLineOffset (boolean)

lineColor

The lineColor property sets the color for the crease, boundary and silhouette lines.

creases, boundary and silhouette

The creases, boundary, and silhouette properties take a value of TRUE or FALSE. They determine where lines are drawn on the surface of the model.

- If creases is TRUE, the Toon or Inker modifier draws lines at where two faces meet at an angle. The value of creaseAngle determines how acute the angle needs to be before a crease line is drawn (see below).
- If boundary is TRUE, the Toon or Inker modifier draws lines wherever two meshes in the model meet.
- If silhouette is TRUE, the Toon or Inker modifier draws lines around the outside edges of the model.

creaseAngle

The creaseAngle property determines how acute the angle between two adjacent triangular faces needs to be before a crease line is drawn between them. A low value requires the faces to be at a very acute angle. A value of 1.0 will draw creases even between faces that are in exactly the same plane.

lineOffset and useLineOffset

If you set lineOffset to TRUE, you can alter the apparent thickness of the creases and boundary lines by altering the value of lineOffset. You can use values between -100.0 (thickest) to 100.0 (thinnest lines). If you use a value outside this range, you will provoke a "Value out of range" script error.Note: You may wish to be cautious about using the lineOffset property, because when the camera is in certain positions, a series of apparently random lines may be drawn into the 3D sprite when useLineOffset is set to TRUE.

Textures

Textures are images that are optimized for display in a 3D environment. Each shader can have textures applied to it. Textures are 2D images drawn on the surface of a model. The appearance of the model's surface is the combined effect of the shader and textures applied to it. If you do not specify a texture, a default red-and-white bitmap is used. Textures can also be displayed as 2D overlays and backdrops, or as particles emitted by a particle system.

A texture is an image object stored inside a 3D cast member. Textures can be used in three ways:

- To map an image to the surface of a mesh or to provide information about the material or the environment, inside one of the texture layers of a shader object, See "Standard shaders" on page 131for details on how to use a texture inside a shader. See "Mapping a texture to a mesh resource" on page 150 for information on how to prepare a mesh resource so that the textures inside a shader are displayed correctly.
- As a 2D overlay or backdrop for a camera object. See "Overlays and backdrops" on page 47.
- To provide an image for the particles in a particle emitter object, see "Particles" on page 197.

The treatment of images in a 3D environment makes intensive use of the computer processor, so various techniques are used to improve performance. The most obvious difference between 3D textures and 2D images is that 3D textures only have a limited number of possible dimensions. See "Images for textures" on page 144 for more details. Textures also have a number of properties that tells the video card how much processing to do before displaying them. See "Texture properties and method" on page 147 for more details.

Texture names

All textures in a given 3D cast member must have unique names. If you try to create a new texture with the same name as an existing texture, a script error occurs: "Object with duplicate name already exists". See "3D namespace" on page 85 for more details.

Accessing a texture

A 3D world exported from a third-party 3D design application is likely to have many textures included within it. Textures are stored in the texture palette of the 3D cast member and can be referenced:

- By name (Lingo only)
- By index number

For example:

```
-- Lingo
put member("3D").texture(1)
-- texture("DefaultTexture")
put member("3D").texture("LENOIR")
-- texture("Lenoir")
// JavaScript
trace (member("3D").getPropRef("texture", 1))
// <texture("DefaultTexture")>
```

Note: *You can obtain the name of a texture if you know its index number, but there is no shortcut for finding the index number of a texture from its name.*

When using JavaScript in Director 11.5, there is no way to access a texture by its name. You must use the syntax a3DMember.getPropRef("texture", aIndex).

When you delete a texture, the index numbers of other textures may change. Textures imported in a W3D are already applied to shaders for different models. Textures for overlays, backdrops, and particle systems can only be applied at runtime. You can get and set the textures for any of these objects using these expressions:

- aShader.texture
- aShader.textureList[aIndex]
- aCamera.overlay[aIndex].source
- aCamera.backdrop[aIndex].source
- aParticleResource.texture

Here are some examples:

```
--Lingo
vSprite = sprite(1)
vMember = vSprite.member
put vMember.model(2).shader.texture
-- texture("Wood")
vMember.model(3).shader.textureList = vMember.texture("Stone")
vSprite.camera(2).overlay[3].source = vMember.texture("North")
vMember.modelResource("Fire").texture = vMember.texture("Flame")
// JavaScript
vSprite = sprite(1);
vMember = vSprite.member;
trace (vMember.getPropRef("model", 1).getProp("shader").getProp("texture"));
// <texture("Wood")>
vStoneTexture = vMember.getPropRef("texture", 2);
vMember.getPropRef("model", 1).getProp("shader").setProp("textureList", vStoneTexture;
vNorthTexture = vMember.getPropRef("texture", 3);
sprite(1).camera.getPropRef("overlay", 4).source = vNorthTexture;
Replace sprite(1).camera with an expression exquivalent to sprite(1).camera(2)
vFireParticle = vMember.getPropRef("modelResource", 5);
vFlameTexture = vMember.getPropRef("texture", 6);
vParticle.texture = vFlameTexture
```

Counting textures

To determine how many textures are stored in the texture palette of a 3D member, you can use the member3D.count property (Lingo) or the `member3D.count()` function (JavaScript):

```
-- Lingo
put member("3D").texture.count
-- 12
// JavaScript
trace(member("3D").count("texture"));
// 12
```

Creating textures at runtime

To create a new texture at runtime you can use the function member3D.newTexture(aUniqueName). See "3D namespace" on page 85 for a Lingo handler that returns a name that you can be sure is not already used for a texture in the given 3D member.

Providing an image source when creating a new texture

You must provide a string name for the new texture. Optionally, you can provide two additional parameters:

```
--Lingo
vTexture = member(1).newTexture(aName, #fromImageObject, aImage)
vTexture = member(1).newTexture(aName, #fromCastMember, aMember)
//JavaScript
vTexture = member(1).newTexture(aName, symbol("fromImageObject"), aImage)
vTexture = member(1).newTexture(aName, symbol("fromCastMember", aMember)
```

If you use #fromImageObject as the second parameter, the third parameter must be an image object. If you use #fromCastMemberas the second parameter, the third parameter can be any member with an image property. Possible member types include #bitmap, #flash, #realMedia, #shockwave3d, #text and #vectorShape. See "Images for textures" on page 144 for more details.

You do not need to indicate an image source for a new texture immediately. You can set the image or member properties of texture in the next line or later if you wish.

Deleting a texture

To delete a texture, you can use the member3D.deleteTexture() command. You can use either the name or the index number to identify the texture to delete. If you provide a valid name or index number, the command will return 1. If not, it will return 0.

```
-- Lingo
put member("3D").texture.count
-- 3
put member("3D").texture(3).name
-- "DeleteMe"
put member("3D").deleteTexture("DeleteMe")
-- 1
put member("3D").deleteTexture(3)
-- 0
put member("3D").deleteTexture(2)
-- 1
// JavaScript
trace(member("3D").count("texture"));
// 3
trace(member("3D").getPropRef("texture", 3).getProp("name"));
// DeleteMe
trace(member("3D").deleteTexture("DeleteMe"));
// 1
trace(member("3D").deleteTexture(3));
// 0
trace(member("3D").deleteTexture(2));
// 1
```

Images for textures

The image for a texture can come from one of three different sources:

- #importedFromFile: The texture is imported with a model from a 3D modeling program.
- #fromCastMember: The texture is created from a bitmap cast member using the newTexture() command, or its member property has subsequently been set to a member with an image property.
- #fromImageObject: The texture is created from a lingo image object using the newTexture() command, or its image property has subsequently been set to an image object

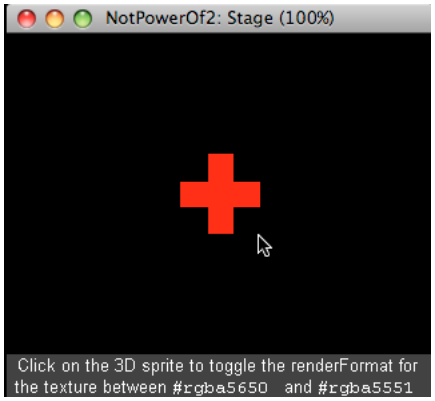
```
put member(1).texture(1).type -- default red-and-white texture
-- #importedFromFile
member(1).texture(1).member = member(2)
put member(1).texture(1).type
-- #fromCastMember
member(1).texture(1).image = member(2).image
put member(1).texture(1).type
-- #fromImageObject
```

Mapping the color information contained in textures onto a dynamic 3D world makes intensive use of the computer processor. To limit the number of calculations made on each frame, textures are always created with height and width dimensions that are powers of 2. Powers of 2 are numbers like 1, 2, 4, 8, 16, 32, and so on. For example, you can create a texture that is 16x256 or 512x64.

If the image or bitmap that you use for a texture does not have pixel dimensions that are a power of 2, both rendering performance and visual quality decreases. The Director player rounds the dimensions of the image to the nearest power of 2. This can lead to both stretching and squeezing. For example, an image that is 96 x 97 pixel is squeezed horizontally and stretched vertically:

```
vTexture = member("3D").texture(1)
vTexture.image = image(96, 97, 1)
put vTexture.width
-- 64
put vTexture.height
-- 128
```

To see a demonstration of poorly dimensioned images for textures, download the movie OverlayDemo.dir and launch it.



An image whose dimensions are not powers of 2 results in a poor quality texture

To ensure that you have the correct dimensions when using text members as the source for a texture image, you can:

- Drag the text member temporarily on to a blank area on the stage
- Use the Text tab in the Property Inspector to set the #boxType of the text member to #fixed
- At the Sprite tab of the Property Inspector, use the W and H fields to set the width and height of the sprite to powers of 2
- Delete the temporary sprite

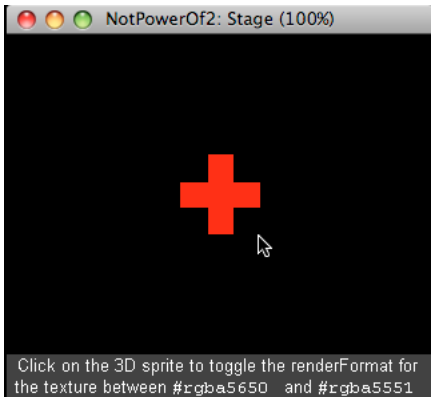
Creating textures that appear to have custom sizes

To avoid the loss of clarity that is associated with this automatic redimensioning, always use bitmaps or images whose dimensions are powers of 2. If you wish to create a texture which appears to have different dimensions, you can use an image with alpha-channel transparency for the parts that you want to make invisible.

The Lingo script below creates an image object that is 64 x 64 and makes most of its pixels red. Next, it creates an alpha channel that contains a black cross with dimensions of 60 x 60. The rest of the alpha image remains white. The script then sets the image of the default texture to the 64 x 64 image, and finally it displays the texture as an overlay at the center of sprite(1).

```
vSprite = sprite(1)
vMember = vSprite.member
vTexture = vMember.texture(1) -- default texture
vImage = image(64, 64, 32, 8)
vImage.fill(rect(0, 0, 60, 60), rgb("#FF0000"))
vAlpha = image(64, 64, 8, #grayscale)
vAlpha.fill(rect(21,  0, 40,  60), rgb("#000000"))
vAlpha.fill(rect( 0, 21,  60, 40), rgb("#000000"))
put vImage.setAlpha(vAlpha)
-- 1
vTexture.image = vImage
put vTexture.width, vMember.texture(1).height
-- 64 64
vSprite.camera.addOverlay(vTexture, point(130, 90), 0)
To see the true size of the texture, execute the following line:
vTexture.renderFormat = #rgba5650
```

This command makes the texture ignore any alpha channel information. To see this script in action, download and launch the movie NotPowersOf2.dir.

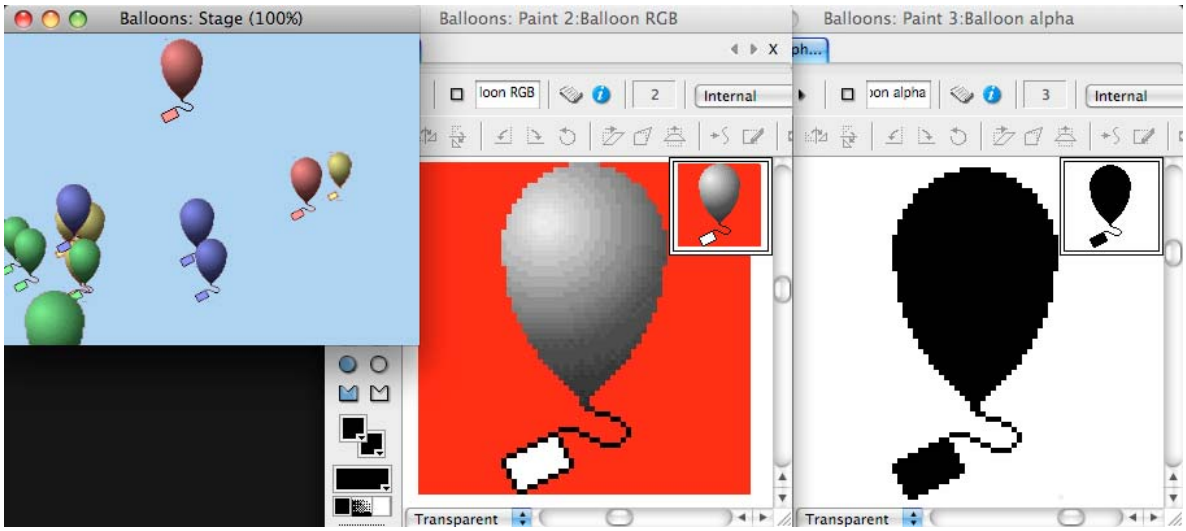


Use an image with a partially transparent alphaChannel to create a texture that appears to be a non-standard size

Images for particle emitters

The texture for a particle emitter resource is mapped onto a square plane. Even if you use a texture whose width and height are different, the texture will be stretched to fill the square plane.

In order to create particles that do not appear to be square, use an image with an alpha channel that creates a non-square opaque area. To see an example of this, download and launch the movie Balloons.dir.



For textures for a particle system, start with a square image and use the alphaChannel to make parts of it invisible

Texture properties and method

Texture objects have nine properties and one method. Three of the properties are read-only:

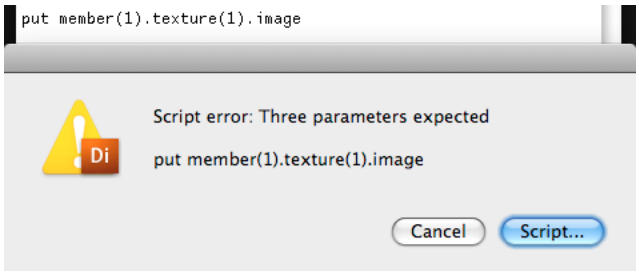
- texture.type (symbol)
- texture.width (integer)
- texture.height (integer)

These properties are set automatically if you change the image or member properties (see below). You can get and set the following five properties:

- texture.member (VOID or member reference)
- texture.renderFormat (symbol)
- texture.quality (symbol)
- texture.nearFiltering (boolean)
- texture.compressed (boolean)

One property that you can set but not read is texture.image.

Note: *If you try to get the value of the texture.image property, a script error occurs.*



Script error when you try to get the value of the texture.image property

scaleDown() method

High quality textures require a lot of space in the video RAM. If the video RAM gets too full, the rendered image can start getting choppy. You can use texture.scaleDown() to reduce the size of the image used by the texture. This results a change of quality that is permanent. You have to delete the texture and recreate it to restore it to its original size. For textures that have a type of #importedFromFile, recreating the texture means reloading the model that uses the texture.

```
vTexture = member("3D").newTexture("Test")
vTexture.image = image(128, 128, 32, 8)
put vTexture.width, vTexture.height
-- 128 128
vTexture.scaleDown()
put vTexture.width, vTexture.height
-- 64 64
vTexture.scaleDown()
put vTexture.width, vTexture.height
-- 32 32
```

Instead of using scaleDown(), you may prefer to provide several different external castLibs containing bitmap members at various sizes. If the user encounters difficulties with rendering all the textures of your 3D world, your application can select a castLib with images at a less challenging scale, and refresh all the textures whose sizes have changed.

Setting the member or image properties

You can change the image displayed by a texture by setting its member property to any member that has an image property. Example member types include:

- #bitmap
- #flash
- #realMedia
- #shockwave3d
- #text
- #vectorShape

You can set the texture.image to any image. For best results, ensure that the dimensions of the member or image are powers of 2. See “Images for textures” on page 144 for details.

renderFormat

The renderFormat property determines how many bits are assigned to displaying each of the color channels, for red, green, blue and alpha. The following are the possible settings:

- #rgba4444
- #rgba5551
- #rgba5550
- #rgba5650
- #default
- #rgba8888
- #rgba8880

The first four values assign 16-bits to each pixel in the texture image. The last 2 assign 32 bits to each pixel and give a higher quality rendering at the cost of 4 times as much video RAM.

Values that end with a 0 create opaque textures. These are suitable for use with images saved in a 24-bit format or lower. The other values provide for transparency, based on information stored in the alpha channel of the image associated with the texture.

The value #rgba5551 provides a single bit for the alpha channel: all pixels in the texture image are either fully opaque or fully transparent. The value #rgba4444 allows 16 different values for each channel: 4096 distinct colors at 16 levels of transparency. The value #rgba8888 gives almost photo-realistic results.

Note: *Older video cards may not support all of these formats. You can use the following code to determine what renderFormats are available on a given machine:*

```
vHardwareInfo = getRendererServices().getHardwareInfo()
put vHardwareInfo.supportedTextureRenderFormats
-- [#rgba8888, #rgba8880, #rgba5650, #rgba5551, #rgba5550, #rgba4444]
```

The value #default is a shortcut to the value set for getRendererServices().textureRenderFormat. By default, this is #rgba5551, but you can set it at any time.

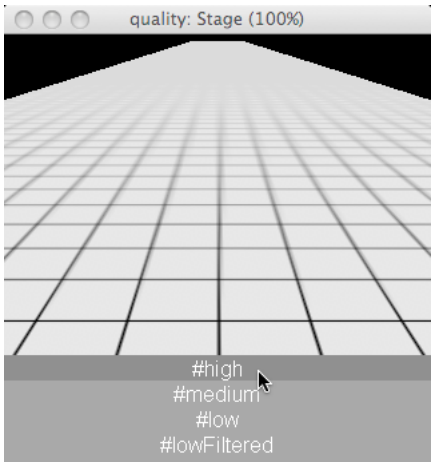
```
put getRendererServices().textureRenderFormat
-- #rgba5551
getRendererServices().textureRenderFormat = #rgba4444
put getRendererServices().textureRenderFormat
-- #rgba4444
```

See textureRenderFormat for more details.

quality

The quality setting determines how a texture appears when it is on a surface that stretches away into the distance. This setting can take four values:

- #low
- #lowFiltered
- #medium
- #high

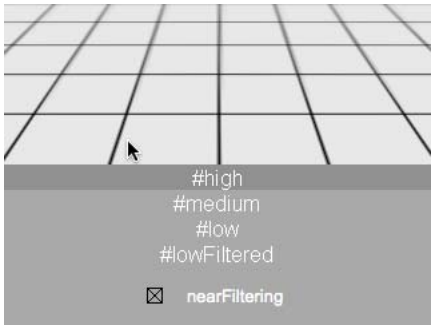


The quality property determines how much processor time is used to deal with foreshortened images

If the mesh on which the texture appears is view squarely by the camera, the value of quality has little to no effect.

nearFiltering

The nearFiltering property determines what happens to texels (the pixels in a texture) that appear bigger than a single pixel on the screen. When nearFiltering is TRUE, oversize texels are antialiased with their neighbors. This makes any potentially jaggy lines appear smoother.



When a texel is so close to the camera that it is bigger than a pixel nearFiltering can smooth its edges

If the camera can never get close enough to a texture for its individual texels to appear bigger than a screen pixel, enabling nearFiltering does not have any visible effect.

compressed

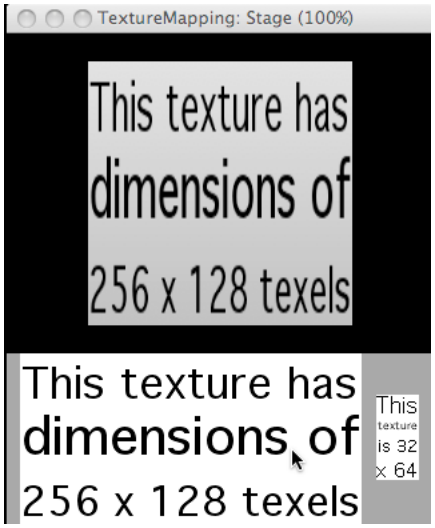
To save video RAM, textures that are currently unused can be compressed. If your scene requires textures to swap in and out of use rapidly, you may want to store them in an uncompressed state, even when they are not currently being used. This reduces the processing time required to decompress them each time they are swapped back into the scene.

Mapping a texture to a mesh resource

The member3D.newMesh() command allows you to create model resources at runtime. These model resources can have any shape that you are able to create through code. See "Creating a mesh resource" on page 170 for more details.

Perhaps you want to use a texture to define the details on the surface of a custom-built #mesh resource. If so, you need to indicate which points within the texture map to which vertex points within the mesh. Different textures can have different width and height dimensions. However, the shader that applies the texture to the mesh will stretch the texture to fit, regardless of the texture's proportions.

Imagine two textures, one with dimensions of 256 x 128 and another with dimensions of 32 x 64. Imagine that each is attached to a shader which is applied to a square mesh. The wide texture will be squeezed horizontally, and the tall texture will be stretched horizontally. For both textures, the bottom left corner will be at the bottom left of the plane and the top right corner will be at the top right of the plane. To test this, download and launch the movie TextureMapping.dir. Click either on the 3D sprite or on one of the texture source sprites to swap the texture in the shader that is applied to the plane.

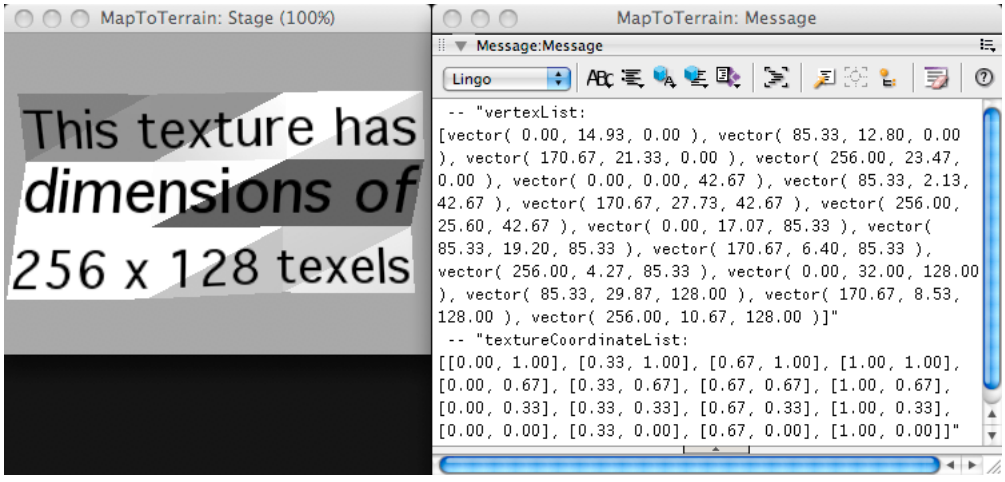


Textures are stretched to fit the mesh regardless of their actual width and height

A 2D image uses pixel positions, measured in integers starting at the top left corner of the image. In an image which is 256 by128 pixels, the bottom left corner is at point(0, 128) and the top right corner at point(256, 0).

A 3D texture measures texel positions in floating point coordinates, starting at the bottom left corner. In a texture with any dimensions, the bottom left corner is at [0.0, 0.0] and the top right corner at [1.0, 1.0].

Each mesh that a texture is mapped to may have a different number and arrangement of vertices. The plane in the image above has four vertices, one at each corner, and two triangular faces. In the terrain image below, the mesh has 16 vertices, in four rows and four columns. This gives a total of 18 faces: (4 - 1) x (4 - 1) x 2 triangles per square. Download and launch the movie MapToTerrain.dir to see how this mesh is created. You will see vertexList and textureCoordinateList information printed in the Message window.

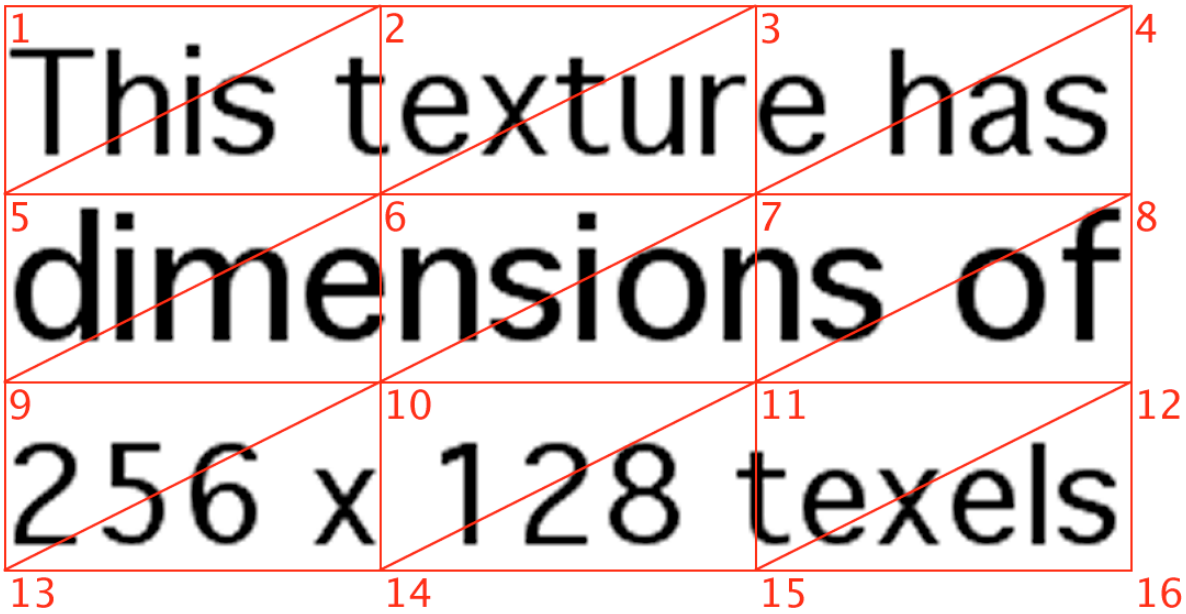


The same texture mapped to a mesh with 18 faces

To map the same texture to different meshes, define which vertex points in the mesh correspond to which points in the texture. This information is stored in the mesh object.

Mapping

The image below shows where the vertex points appear for the terrain mesh, and provides the index number for each vertex point.



The faces in the mesh model are shown as triangles

```
[[0.00, 1.00], [0.33, 1.00], [0.67, 1.00], [1.00, 1.00],
[0.00, 0.67], [0.33, 0.67], [0.67, 0.67], [1.00, 0.67],
[0.00, 0.33], [0.33, 0.33], [0.67, 0.33], [1.00, 0.33],
[0.00, 0.00], [0.33, 0.00], [0.67, 0.00], [1.00, 0.00]]
```

The list above shows how each of the indexed vertex points maps to the floating point coordinates in the texture. To make it easier to follow what is happening, the MapToTerrain.dir creates the vertex points and the texture coordinates in an order that appears logical when printed on a page. The actual order of items in these lists is not important, so long as both lists use the same order.

Note that neither changing the image of the texture, nor changing the texture used by the shader will have any effect on the mapping. The same relative positions within the texture image will be used, regardless of the absolute dimensions of the image.

Defining faces in a mesh

Each face in a mesh is defined by a list of three integers, representing the index numbers of the vertex points at each corner. For example, the face at the top left corner can be defined as [1, 5, 2]. Note that faces are always defined in counter-clockwise order.The faces themselves can be defined in any order. In the MapToTerrain.dir movie, for simplicity, they are defined in the same order as the text is read. The top left face thus has a faceID of 1.

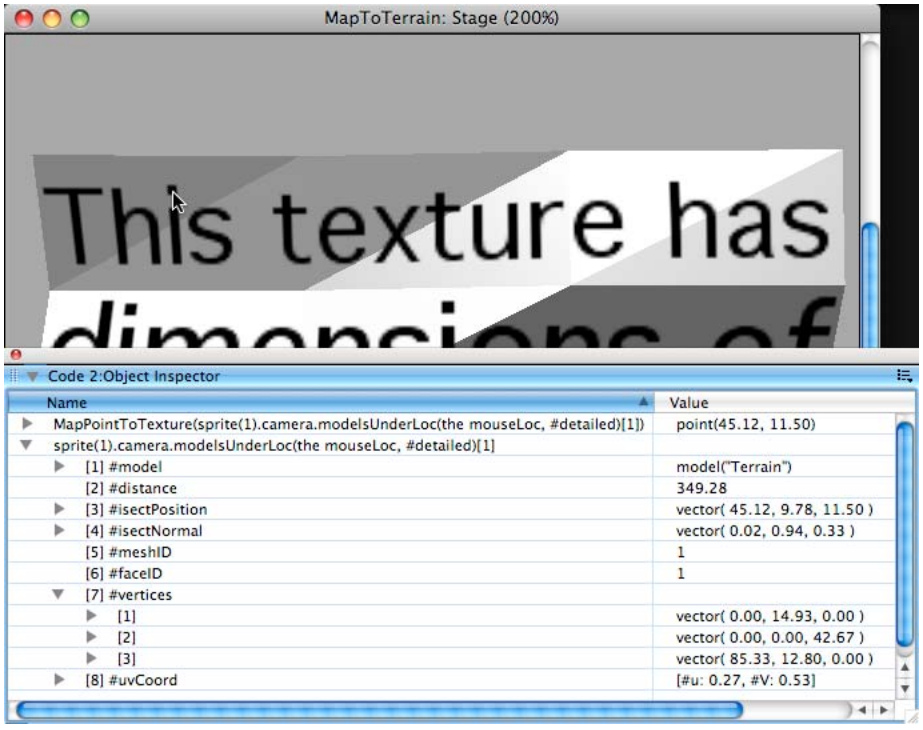
Determining which pixel is at a given uv coordinate

When used with the #detailed option, the and member3D.modelsUnderRay() functions allow you to discover the point where a ray intersects with a given face of a give mesh of a given model. The output from these functions is in the form of a list of property lists:

```
[[#model:         <model intersected by ray>,
#distance:      <float distance to intersection point>,
#isectPosition: <vector worldPosition of intersection point>,
#isectNormal:   <vector normal of the face at intersection>,
#meshID:        <integer id of mesh to which the face belongs>,
#faceID:        <integer id of intersected face>,
#vertices:     [<vector>, <vector>, <vector>],
#uvCoord:      [#u: <float>, #v: <float>]], ...]
```

In this list, note the items #meshID, #faceID and #uvCoord. You can use the values in this list to determine which point in the texture is at the intersection point.

MapToTerrain.dir movie contains a Map Point To Texture script that will perform this calculation for you. You will find a text member named Expression To Watch. Copy the text that it contains and select the menu item Window > Object Inspector. Double-click on an empty space in the Name column and paste the text that you have just copied. Now, run the movie and move your mouse over the terrain texture. In the image below, the mouse is hovering over the dot on the "i" in the word "This" in the triangle whose faceID is 1.



Using MapPointToTexture() to determine which point in a texture is under the mouse

Face 1 is defined as [1, 5, 2] (see diagram above). You can see from the list of textureCoordinates that these points are at [0.00, 1.00], [0.00, 0.67] and [0.33, 1.00] within the texture image, respectively.

```
[[0.00, 1.00], [0.33, 1.00], [0.67, 1.00], [1.00, 1.00],
[0.00, 0.67], [0.33, 0.67], [0.67, 0.67], [1.00, 0.67],
[0.00, 0.33], [0.33, 0.33], [0.67, 0.33], [1.00, 0.33],
[0.00, 0.00], [0.33, 0.00], [0.67, 0.00], [1.00, 0.00]]
```

The uvCoord list reports that the point under the mouse is 0.27 along the u axis and 0.53 along the v axis for this face. The u axis for a face starts at the first corner (here, index 1: [0.00, 1.00]) and points towards the second corner (here, index 5: [0.00, 0.67]). The v axis starts at the first corner and points towards the third corner (here, index 2: [0.33, 1.00]). In this case, the axes are vertical and horizontal, which simplifies the calculation. (The MapPointToTexture() handler takes care of more complex cases where the two axes are not vertical and horizontal and not at right-angles to each other).

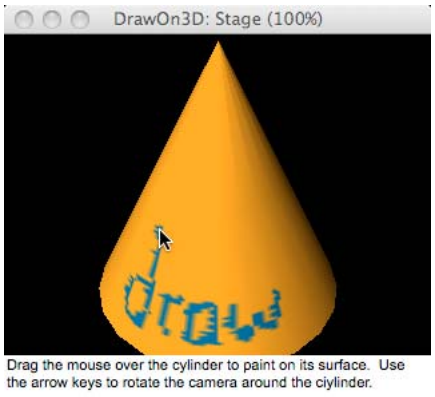
```
vFloatDistanceFromBottom = 1.00 + (0.27 * (0.67 - 1.00))
put vFloatDistanceFromBottom
-- 0.91
vFloatDistanceFromLeft   = 0.00 + (0.53 * (0.33 - 0.00))
put vFloatDistanceFromLeft
-- 0.17
```

To convert these values into a position within the texture image, you need to take into account the dimensions of the texture image, and the fact that the vertical float value is calculated from the bottom:

```
vImageWidth  = 256
vImageHeight = 128
vLocH = integer(vImageWidth * vFloatDistanceFromLeft)
put vLocH
-- 45
vLocV = integer(vImageWidth * (1.0 - vFloatDistanceFromBottom))
put vLocV
-- 11
```

Drawing on a 3D model

You can use this technique to draw on the surface of a model. To see this in action, download and launch the movie DrawOn3D.dir.



The DrawOn3D.dir movie uses the Map Point To Texture script to let you draw on a texture image

Rendering

Rendering determines which software and hardware technologies are applied to the 3D data before it is displayed on the screen.

Not all computers are created equal. What looks good on your development machine may result in poor quality graphics or slow performance on some end-users' computers.

Issues can include:

- The choice of rendering technology. See "Using rendering methods" on page 155.
- The amount of video RAM available. This can have an effect on the number, quality and size of the textures that your application can use. See "Texture properties and method" on page 147.
- The availability on the end-user's machine of hardware accelerated anti-aliasing. See "Antialiasing" on page 157.

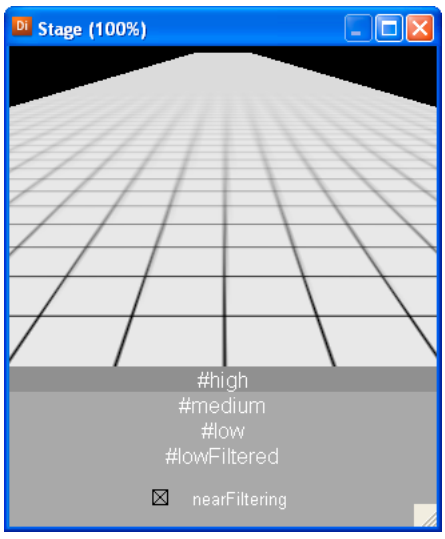
  Another feature that you can use to quickly render 3D objects with textures on your users' devices is 'Render to texture'. See "Render to texture" on page 157 for details.
- The algorithm used for shading. See "Flat shading and smooth shading" on page 197.

Using rendering methods

Director 3D relies on third-party technologies to render a 3D scene to the user's screen. The video cards on different end-users' machines may provide a different range of technologies than those you have available on your development machine.The rendering method refers to the specific way Director displays 3D images on the Stage. The methods available depend on the type of hardware you have. The rendering methods include the following:

- #auto: Director selects the best method based on the client computer's specific hardware and drivers.
- #openGL: OpenGL drivers for a 3D hardware accelerator are used. OpenGL is available for the Mac and Windows platforms.
- #directX9: Specifies the DirectX 9 drivers for hardware acceleration that work only with Windows platforms. #auto sets the renderer to DirectX 9.
- #directX7_0: DirectX7_0 drivers for a 3D hardware accelerator are used. This option is available for Windows only.
- #directX5_2: DirectX5_2 drivers for a 3D hardware accelerator are used. This option is available for Windows only.
- #software: The Director built-in software renderer is used. This option is available on the Mac and Windows platforms.

The rendering method can have a dramatic effect both on performance and on the quality of the rendered image.



A comparison of #software (left) and #openGL (right) for rendering a simple scene

Selecting a preferred 3D renderer in Director

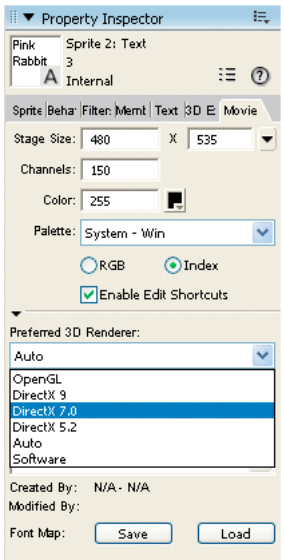
If your hardware permits you to select different methods, use the following procedure:

1 Select the Stage.

2 Open the Property inspector.

3 Click the Movie tab.

4 Select a rendering method from the menu.



Property inspector - Preferred 3D Renderer

If you do not select a rendering method, Director defaults to #auto. The name of the active 3D renderer property appears below the menu. The value of this property indicates which rendering method is currently being used. This is especially useful when you want to know which renderer is active while you have #auto selected.

Selecting a 3D renderer through code

You can achieve the same effect using Lingo or JavaScript syntax, by setting the value of _movie.preferred3dRenderer to one of the symbols given above.

Computer A

```
put getRendererServices().rendererDeviceList
-- [#openGL]
```

Computer B

```
put getRendererServices().rendererDeviceList
-- [#directX5_2, #directX7_0, #directx9, #software]
```

To select a particular renderer, you can set getRendererServices().renderer to the symbol representing the renderer you want to use. For example:

```
getRendererServices().renderer = #directx9
```

You can use the read-only property _movie.active3dRenderer to determine whether the selected renderer has been successfully set.

Texture renderFormat

Textures give precision and detail to the rendering of an image. However, they require significantly more space in the computer's video Random Access Memory than 2D images.

A 2D image normally appears flat on the screen with one pixel of memory space corresponding to one pixel of screen space. A 3D texture may be placed on a surface at any angle to the camera. A texture for a wall, for example may need to be very detailed when it is viewed from directly in front and close up. If the wall is in the distance, or seen at a steeply foreshortened angle, very few pixels on the screen actually uses the pixels stored in the texture. Only a small proportion of the information stored in textures loaded into video RAM is used on any one frame. The graphics card used to render an image to your client's monitor will have a finite capacity for textures. You may want to exploit the available space to the best effect.

textureRenderFormat

You can set the texture.renderFormat property individually for each texture. Alternatively, you can set a default value for all textures, using getRendererServices().textureRenderFormat.

For the best graphic quality, you can choose one of the 32-bit formats (#rgba8880 or #rgba8880). For more economic use of the video RAM, you can use one of the 16-bit formats (#rgba5550, #rgba5650, #rgba5551 or #rgba4444). Any texture whose renderFormat is set to #default will use the value set by getRendererServices().textureRenderFormat. You can alter this value at runtime. See "Texture properties and method" on page 147for a discussion of the scaleDown() method and for alternative techniques for finding the right balance between texture detail and size.

Antialiasing

See "Antialiasing" on page 390.

Render to texture

Let us consider that you are developing a car-racing game in Director. You need the driver of the car to see the other cars and the 'scene' behind the car in the side rear-view mirrors.

The easiest way to achieve this is to use the render to texture feature. This feature helps you render objects with specific 'textures'during run-time and display the rendered scene on your users' devices faster.You can define the 'texture' using a camera view. Everything that is within the camera view is considered as a 'texture', and the whole 'view' is applied to the chosen object. In the car-racing example, you can place the camera behind the car. As the car moves, the camera captures the view behind the car, and this view is applied as a 'texture' to the rear-view mirrors.

When you use camera views for render to texture, the image is copied onto the video RAM and then in the Lingo code, and is then applied to the chosen object at run-time.

You can render the objects quickly by applying the image to the object directly from the video RAM. To do so, you need to set the `updateRTImageOnRender` property to false.

The following are the typical steps that you need to follow for using render to texture:

1 Create a new texture with type rendertexture:

```
member(whichCastmember).newTexture(newTextureName {,#typeIndicator, rendTexWidth,
rendTexHeight})
```

The `updateRTImageOnRender` property is set to '`True`' by default. This means, Director copies the texture image from the video RAM to the Lingo code and then applies to the target object.

If you want to apply the texture directly from video RAM to the target object, set the `updateRTImageOnRender` property to '`False`'.

```
<castmember>.updateRTImageOnRender=0
```

2 Using the renderToTexture method, assign a camera to the new texture that you created.

```
<3dSpriteRef>.renderToTexture(cameraObj, renderTextureObj)
```

If `updateRTImageOnRender` is 'True', you can use the `renderTextureObj.image` property to obtain a Lingo image of the texture on video RAM. You can then apply more effects, for example, filters, on this image, before applying to the target object.

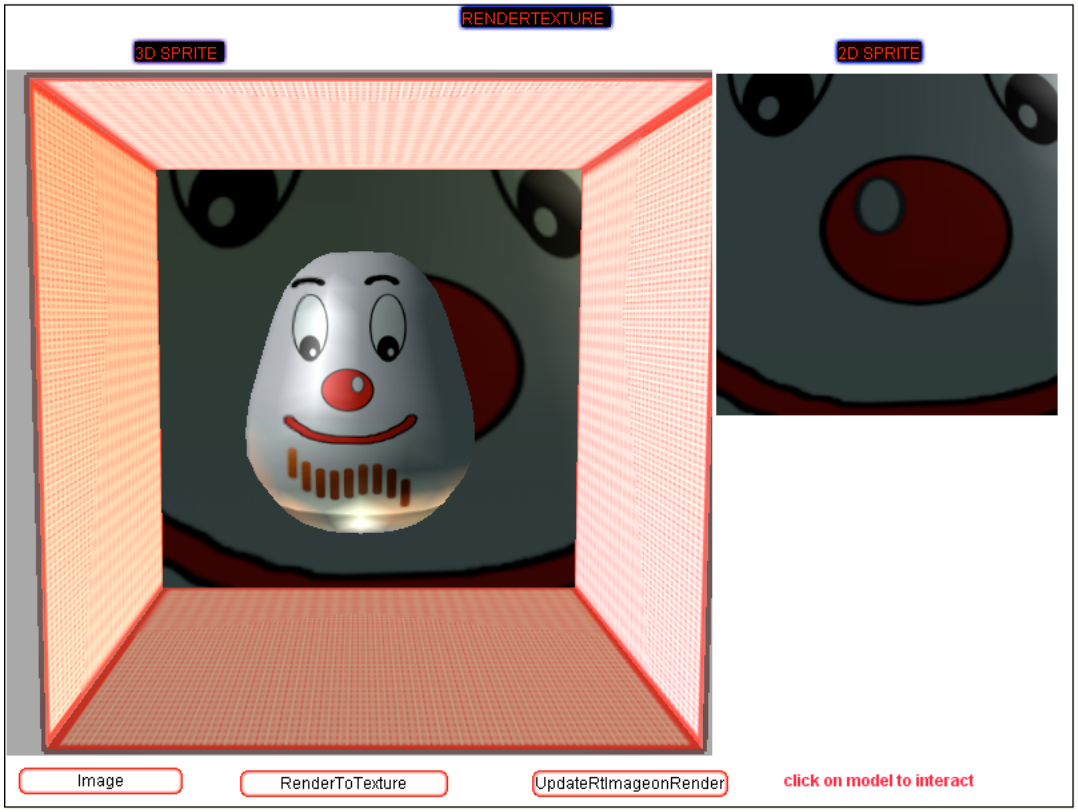
3 Apply the texture to the target object.

```
vMember.model(3).shader.texture = vMember.texture("renderTextureObj")
```

Here is an example that starts with creating a new Render Texture object and then renders a 3D scene into it using the current camera of the sprite. Then, the rendered image is populated into a bitmap cast member named `bitmapRT`.

```
-- Lingo
sprite3D = sprite("3D World")
myRT = member("3D World").newTexture("MyRT1",#renderTexture, 512, 512)
sprite3D.renderToTexture(sprite3D.camera, myRT)
member("bitmapRT").image = myRT.image
```

To see a demonstration of this feature, download and launch the movie RenderTexture.dir.



Render to texture used in a 3D sprite of the RenderTexture.dir movie

When you launch the movie, you see that the wall behind the russian doll in the 3D sprite is rendered with a 2D static image.

Click the Render To Texture button to see the texture rendered on the wall.

The 3D sprite has two cameras:

- The default one that is facing the russian doll and through which you look at the sprite.
- The second camera that is behind the doll and is facing the doll.

The second camera view (the back of the doll) is the texture that you see on the wall behind when you click the Render To Texture button.

You also notice that the 2D image adjacent to the 3D sprite too changes when you click the Render To Texture button. This is because the `updateRTImageOnRender` is set to '`True`', and the texture is obtained as a 2D image before applying to the wall.

Now, click the updateRTImageOnRender button to set the property to 'False'. You now see that the 2D image is not updated, and the texture is directly applied to the wall.

Geometry

The word geometry is often used to describe everything that gives the illusion of three dimensions in a 3D world. It can refer to anything from the invisible numbers used to define the structure of a model resource through the visible position and location of a model to the relationship between the positions of individual nodes. This section looks at how the 3D elements of a virtual world are defined. It covers:

- What a 3D model is. See "Models" on page 159.
- Different ways of moving models and of controlling the interactions between them. See "Manipulating models" on page 163.

  Also see "Motion" on page 260, "Collisions" on page 323and "Physics" on page 293.
- How a model resource defines the shape of a model. See "Model resources" on page 163.
- How to generate custom model resources at runtime. These can have a variety of different shapes. For example:
  - Geometrical figures, see "Primitives" on page 165.
  - Extruded shapes, see "Creating an extruder resource" on page 166.
  - Arbitrary shapes, see "Creating a mesh resource" on page 170.
  - Distorted 2D planes "Creating a terrain mesh" on page 176. Customized mesh resources may also display multiple textures. See "Mesh resources with multiple shaders" on page 182.
- How to change the shape of a model resource at runtime. See "Manipulating a mesh resource" on page 182and "MeshDeform modifier" on page 186.
- How to give the illusion that a mesh of flat triangular faces is a smoothly curved surface. See "Flat shading and smooth shading" on page 197.

Models

The following types of visual content can be displayed inside a 3D sprite:

- The default background color that appears where there is no other visual content
- Backdrops, 2D areas that appear behind any 3D content
- 3D models
- Overlays, 2D areas that appear in front of any 3D content.

- Debugging axes and bounding spheres

In other words, the only way to display 3D shapes is with models. See Model Properties and Model Methods for details.

Accessing the models in a 3D member

To determine how many models a 3D member currently contains, you can use the count property (Lingo) or method (JavaScript).

```
-- Lingo
put member("3D").model.count
-- 1
// JavaScript
trace(member("3D").count("model"));
// 1
```

To access a particular model, you can use its name (Lingo only) or its index number.

```
-- Lingo
put member("3D").model[1]
-- model("Model")
put member("3D").model(1)
-- model("Model")
put member("3D").model("model")
-- model("Model")
// JavaScript
trace(member("3D").getPropRef("model", 1));
// <model("Model")>
```

If a model with the given name or index number does not exist, Director will return VOID (Lingo) or undefined (JavaScript). No error occurs.

Creating and deleting models

Use the member3D.newModel() function to create a new model with a given unique name.

Note: *Unique name means that no other node (model, light, group or camera) with the given name already exists. If you try to create a new model with the same name as an existing node, a script error occurs: "Object with duplicate name already exists".*

You can optionally provide a pointer to a modelResource object in the call.

See "3D namespace" on page 85 for more details and a script that ensures that you have a unique name for the model that you are about to create.

```
-- Lingo
vModel = member("3D").newModel("ResourceFree")
put vModel, vModel.resource, vModel.shaderList
-- model("ResourceFree") <Void> []
vResource = member("3D").modelResource("Sphere")
vModel = member("3D").newModel("ResourceFull", vResource)
put vModel, vModel.resource, vModel.shaderList
-- model("ResourceFull")
// JavaScript
vModel = member("3D").newModel("ResourceFree");
<model("ResourceFree")>
trace( vModel, vModel.resource, vModel.shaderList);
// <model("ResourceFree")> undefined <[]>
vResource = member("3D").getPropRef("modelResource", 5);
<sphere("Sphere")>
vModel = member("3D").newModel("ResourceFull", vResource);
<model("ResourceFull")>
trace(vModel, vModel.resource, vModel.shaderList);
// <model("ResourceFull")> <sphere("Sphere")> <[shader("DefaultShader")]>
```

Note: *If you create a model without defining a modelResource for it, you cannot attach any modifiers to the model. If you set the resource property of a model to VOID, all the modifiers and shaders that were initially attached to the model are deleted.*

To delete a model, use the member3D.deleteModel() function. You can identify the model either by its name or its index number. When you delete a model, the index number of other models may change. If you attempt to delete a non-existent model, no error occurs.

```
-- Lingo syntax
put member("3D").deleteModel("ResourceFree")
-- 1
put member("3D").deleteModel("ResourceFree")
-- 0
put member("3D").deleteModel(2)
-- 1
// JavaScript syntax
trace(member("3D").deleteModel("ResourceFull"));
// 1
member("3D").deleteModel("ResourceFull");
0
member("3D").deleteModel(2);
1
```

Geometry

The shape of a 3D model is defined by a model resource. See Model resources. Several models may share the same model resource. You can set the resource property of a model to change the model resource that is displayed by the model. You can set the resource property of a model to VOID. This will make the model invisible (since it now has no geometry), but all the children of the model remain unchanged.

shaderList

A model's resource can have more than one mesh. The model's model.shaderList property is a list that contains one shader for each mesh in the model's resource. If you set the model's resource for a modelResource with more meshes, the newly created entries in the model's shaderList are filled with the default shader. The same happens if you set the shaderList of a model to a list which does not contain enough shaders for all the meshes in the model.

```
-- Lingo syntax
v3DMember = member("3D")
vModel = v3DMember.model(1)
put vModel.resource.type
-- #cylinder
put vModel.shaderList
-- [shader("DefaultShader"), shader("DefaultShader"), shader("DefaultShader")]
vShader = v3DMember.shader(2)
vModel.shaderList = [vShader, vShader]
put vModel.shaderList
-- [shader("Red"), shader("Red"), shader("DefaultShader")]
// JavaScript
v3DMember = member("3D");
vModel = v3DMember.getPropRef("model", 1);
trace(vModel.resource.type);
// #cylinder
trace(vModel.shaderList)
// <[shader("DefaultShader"), shader("DefaultShader"), shader("DefaultShader")]>
vShader = v3DMember.getPropRef("shader", 2);
vModel.shaderList = list(vShader, vShader);
trace(vModel.shaderList);
// <[shader("Red"), shader("Red"), shader("DefaultShader")]>
```

If you set the shaderList property after changing the resource, then the model keeps a separate record of which shaderList is used by which resource.

Placement in 3D world space

The position, rotation and scale of a model in the world is defined by the model's transform property, in conjunction with the transform of each node in the model's parent chain. To determine exactly how the model appears in the world, you can use the node.getWorldTransform() method. You can use node.worldPosition to return the vector position of the model's origin point in world space. You can use node.removeFromWorld() to set the model's parent to VOID. Its transform will remain unchanged. If you restore the model to its original parent, it will reappear in the same place in the world. If you use node.addToWorld(), the parent of the model will be set to the group("World"). If this was not the model's original parent, then it may find itself in a new location. See "Nodes" on page 88 for more details.

Appearance

For a complete discussion on how to control the surface characteristics of a model, see "Shaders and appearance modifiers" on page 130 and "Textures" on page 141.

Properties and methods shared with other nodes

Models are one of the four types of node objects. For details on how to link nodes together into parent-child hierarchies, and on the properties and methods shared by all nodes, see "Nodes" on page 88.

Features unique to models

Apart from being physically visible in a 3D sprite, models differ in a number of other ways from groups, lights, and cameras.

- Modifiers: You can attach appearance modifiers, behavior modifiers, and animation modifiers only to models. See "Modifiers" on page 50.
- Animation: You can animate models using keyFrame animations or bones animations. A key Frame animation can alter the position, rotation, and scale of a model over time. A bones animation can manipulate virtual bones inside the model to alter the model's shape over time. See "Pre-defined animations" on page 270 for more details.

- Visibility: You can make a model visible only from the outside, only from the inside, visible from any position, or completely invisible. An invisible model remains in the 3D world, and can participate in collision detection and other physical interactions. See model.visibility.
- Cloning from a different cast member: You can clone a model (and optionally all the objects in its child hierarchy and those used to define its appearance), from a different cast member. See member3D.cloneModelFromCastMember().

Sources

3D models can be created in a third-party 3D design application. See "Sources of 3D content" on page 63. You can also use Director's built-in primitives to create model resources for models on the fly. For geometric shapes, see "Primitives" on page 165. For extruded shapes, see "Creating an extruder resource" on page 166. To create meshes with arbitrary shapes, see "Creating a mesh resource" on page 170. To create distorted 2D planes for use as terrains, see "Creating a terrain mesh" on page 176.

Manipulating models

Models can behave in different ways.

- Models can remain at a static location (like a building), or they can move around (like a ball). See "Translation" on page 206.
- Models can have fixed geometry or they can be animated or deformed (like a running avatar, a windmill, a crashing car or a shower of particles). See "Pre-defined animations" on page 270, "Manipulating a mesh resource" on page 182and "Particles" on page 197.
- Models can mutate from one object to another. See "Model resources" on page 163.
- Models can allow other models to pass through them or be made to react to collisions with other models. See "Collisions" on page 323.
- They can behave as if they had physical properties, such as mass and friction. See "Physics" on page 293.

Model resources

Director recognizes eight types of model resources:

- #fromFile: Any model created in a third-party 3D design application, converted to W3D format and imported into Director, has a model resource of the type #fromFile. The other seven types are Primitive resources that can be generated on the fly at runtime in Director.
- #sphere
- #plane
- #cylinder
- #box
- #extruder
- #mesh
- #particle

See "Primitives" on page 165 for more information on the first four, geometrical types. See "Creating an extruder resource" on page 166 for information on the #extruder type. "Creating a mesh resource" on page 170 for information on the #mesh type. See "Particles" on page 197 for information on the #particle type. See Model Resources for a full list of the methods and properties of the different model resource types.

To determine how many modelResources a 3D member currently contains, you can use the count property (Lingo) or method (JavaScript).

```
-- Lingo
put member("3D").modelResource.count
-- 1
// JavaScript
trace(member("3D").count("modelResource"));
// 1
```

To access a particular modelResource, you can use its name (Lingo only) or its index number.

```
-- Lingo
put member("3D").modelResource[1]
-- plane("DefaultModel")
put member("3D").modelResource(1)
-- plane("DefaultModel")
put member("3D").modelResource("model")
-- plane("DefaultModel")
// JavaScript
trace(member("3D").getPropRef("modelResource", 1));
// <plane("DefaultModel")>
```

If a modelResource with the given name or index number does not exist, Director returns VOID (Lingo) or undefined (JavaScript). A script error does not occur.

Accessing the resource of a model

You can get and set the resource property of a given model to any available modelResource:

```
-- Lingo syntax
vResource = member("3D").model(1).resource
put vResource
-- sphere("Sphere")
member("3D").model(1).resource = member("3D").modelResource(3)
put member("3D").model(1).resource
-- mesh("Mesh")
// JavaScript syntax
var vResource = member("3D").getPropRef("model", 1).resource;
<sphere("Sphere")>
vResource = member("3D").getPropRef("modelResource", 3);
<mesh("Mesh")>
member("3D").getPropRef("model", 1).resource = vResource;
<mesh("Mesh")>
```

Creating modelResources

The command to use to create a new modelResource depends on the type of resource:

- For the #sphere, #plane, #cylinder and #box types, use the member3D.newModelResource() function. See "Primitives" on page 165 for more details.
- For the #extruder type, use the function textMember.extrude3d(a3DMember). See "Creating an extruder resource" on page 166 for more details.
- For the #mesh type, you need to use member3D.newMesh(). See "Creating a mesh resource" on page 170for more details.

You cannot create a new modelResource of the type #fromFile from within Director. However, you can duplicate the modelResource of an existing model, using the node.cloneDeep() function. If you require, you can then modify the structure of the cloned modelResource by adding the #meshDeform modifier to the cloned model. For more details, see "Manipulating a mesh resource" on page 182.

While creating a new modelResource, provide a unique name for the modelResource. A unique name means that no other modelResource with the given name already exists.

Note: *If you try to create a new modelResource with the same name as an existing modelResource, a script error occurs: "Object with duplicate name already exists".*

See "3D namespace" on page 85 for more details. A model can, however, have the same name as a modelResource.

Deleting model resources

To delete a model, use the member3D.deleteModel() function. You can identify the model either by its name or its index number. When you delete a model, the index number of other models may change. If you attempt to delete a non-existent model, no error occurs.

```
-- Lingo
put member("3D").deleteModelResource("NoLongerRequired")
-- 1
put member("3D").deleteModelResource("NoLongerRequired")
-- 0
put member("3D").deleteModelResource(2)
-- 1
// JavaScript
trace(member("3D").deleteModelResource("NoLongerRequired"));
// 1
member("3D").deleteModelResource("NoLongerRequired");
0
member("3D").deleteModelResource(2);
1
```

Primitives

Director provides seven types of Primitive resources that can be generated on the fly at runtime in Director.

- #sphere: Sphere properties
- #plane: Plane properties
- #cylinder: Cylinder properties
- #box: Box properties
- #extruder: "Creating an extruder resource" on page 166
- #mesh: "Creating a mesh resource" on page 170
- #particle: "Particles" on page 197

Extruder, mesh, and particle resources are more complex than #sphere, #plane, #cylinder, and #box resources. This section explains the first four geometrical types.

Creating a regular primitive resource

To create a new modelResource of the type #sphere, #plane, #cylinder or #box, use the member3D.newModelResource() function.

Note: *Ensure that the name you provide for the new resource and the modelResource type is unique.*

The following code creates a new modelResource of the type #sphere in the member "3D", and then creates a model from the resource:

```
-- Lingo syntax
v3DMember = member("3D")
vName = "Sphere"
vResource = v3DMember.newModelResource(vName, #sphere)
vModel  = v3DMember.newModel(vName, vResource)
// JavaScript syntax
<(member 2 of castLib 1)>
vName = "Sphere";
vResource = v3DMember.newModelResource(vName, symbol("sphere"));
vModel  = v3DMember.newModel(vName, vResource);
```

Geometry of a regular primitive resource

The origin point of a regular primitive resource is at its center. The default dimensions of the #sphere, #box, and #cylinder primitives give them a width, height, and length of 50 world units.

The default size of the #plane primitive gives it a width and a length of 1 unit. The #plane resource's length property actually determines its height. The #plane primitive is rotated so that it is visible from the default camera position. To create a horizontal plane, rotate the model -90° around the world's x-axis.

Modifying the mesh of a primitive resource

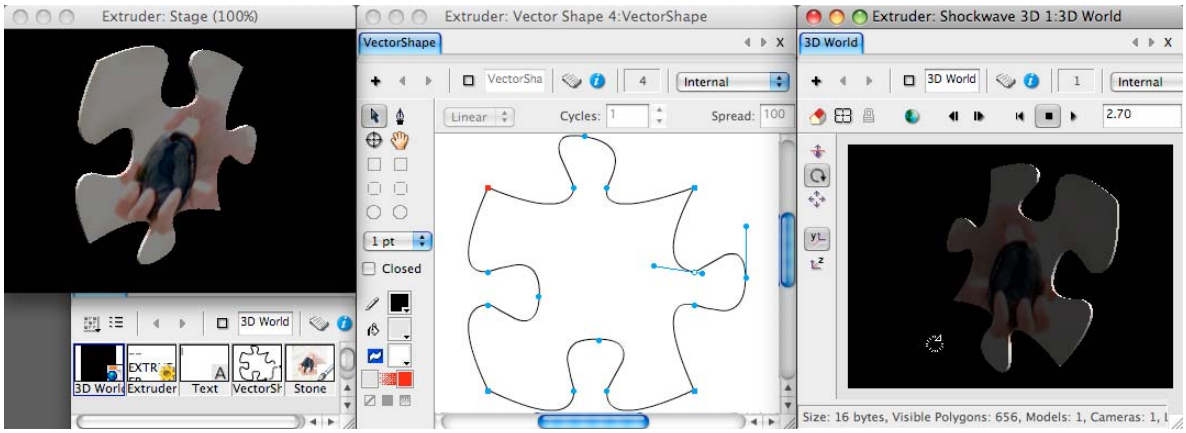
You can obtain a wide variety of shapes by modifying the various properties of the regular primitives. For example, you can create hemispheres, cones, tubes, rectangular fences, and other shapes. You cannot set the position of a particular vertex point in a primitive modelResource directly, but you can apply the #meshDeform modifier to a model created from a primitive modelResource, and use the meshDeform modifier to move vertex points around. See "MeshDeform modifier" on page 186 for details.

Creating an extruder resource

The extrude3d() function is designed to create an #extruder resource from a text cast member. However, you can change the vertexList property of an #extruder resource after it has been created.

Extruding text

To see an example of using an extruder to create three-dimensional text in a 3D cast member, download and launch the movie 3dExtruder.dir.



An #extruder resource creates 3D text using the contents of a text member

To creating a 3D model from extruded text, you require two lines of code:

```
vExtruder = member("Text to Extrude").extrude3D(member("3D"))
vModel = member("3D").newModel("Extrusion", vExtruder)
```

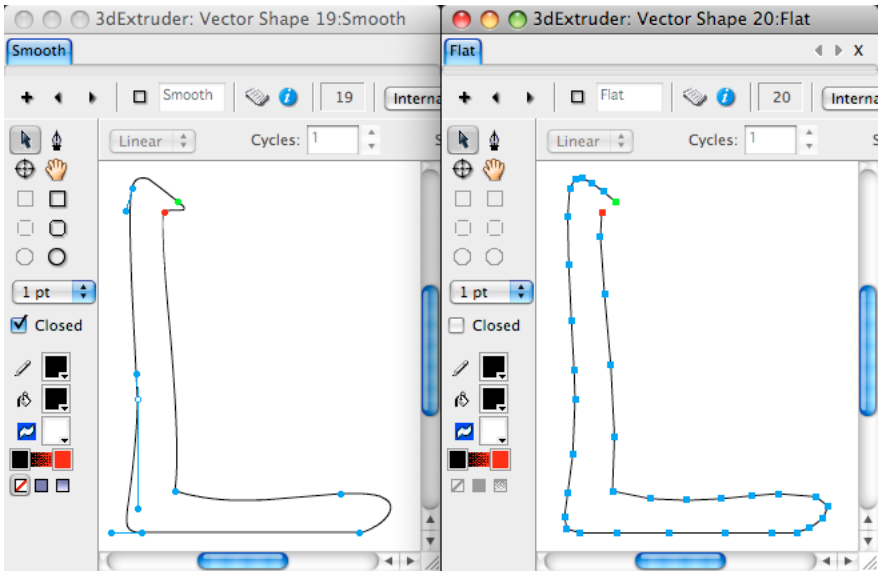
How it works

The information on the shape of each character in a font is stored as a vertexList, similar to the vertexList of a VectorShape member. A vertexList defines a curved line. It consists of a series of points that appear in known positions on the line and information about how the line curves between adjacent points.

A 3D modelResource can only store points. When a vertexList is converted from 2D to 3D, the information about the curvature of the surface between fixed points is lost. The points in the vertexList of a 3D modelResource are joined together by straight lines. To generate what looks like a smooth curve, the extrude3D() command generates a series of intermediate points along the curved line, and joins them up with straight lines.

The illustration below helps you visualize the process. The Smooth shape on the left uses curved lines. The Flat shape on the right creates an approximation to the Smooth shape using many more points joined together by straight lines. (You can find a Lingo handler that performs this conversion in the Create Straight Line Approximation script in the 3dExtruder movie).

The extrude3d() method then creates another group of points parallel to the first, and joins the points up to create a tunnel the shape of each letter.



A visualization of how the extrude3D() command approximates a curved line from a series of straight lines

Geometry

The origin point of the extruder resource is at the bottom left corner of the front face. The width and height of the resource, in world units, is approximately equal to the width and height of the image of the text, in pixels, after trimming away all the white space around the edges.

For example, if the text fits into a bounding box with a width of 100 pixels and a height of 64 pixels, the extruder resource fits into a bounding box that is approximately 100 units wide 64 units high. The length of the extruder resource depends on its tunnelDepth property (see below).

Here is a Lingo command that you can execute to check this. Launch the 3dExtruder.dir movie and then execute this command in the Message window:

```
-- Lingo syntax
do "vImage = member(3).image.trimWhiteSpace()"&RETURN&"put #trimmedImage, vImage.width,
vImage.height"&RETURN&"vBox = GetBoundingBox(member(1).model(1))"&RETURN&"put #boundingBox,
integer(vBox.maxX - vBox.minX), integer(vBox.maxY - vBox.minY)"
-- #trimmedImage 155 75
-- #boundingBox 157 75
```

The 3dExtruded.dir movie uses the Get Bounding Box script to determine the smallest possible box that fits around the extruder model. It then creates a group to act as the parent of the extruder model, and places this parent group at the center of the bounding box. If you rotate the parent group, the extruder model rotates around its center.

Extruder properties

Extruder resources have five properties that you can get and set:

- extruderResource.smoothness: An integer value between 1 and 10. This determines how many points are used to approximate the curved lines between the fixed points. A value of 1 means that only the fixed point is used. The default value is 5, meaning that 4 additional points are added along the curve.
- "Particles" on page 197tunnelDepth: A floating point value between 1.0 and 100.0. The default value is 50.0. This determines the distance in world units between the front and back faces of the extruded model.

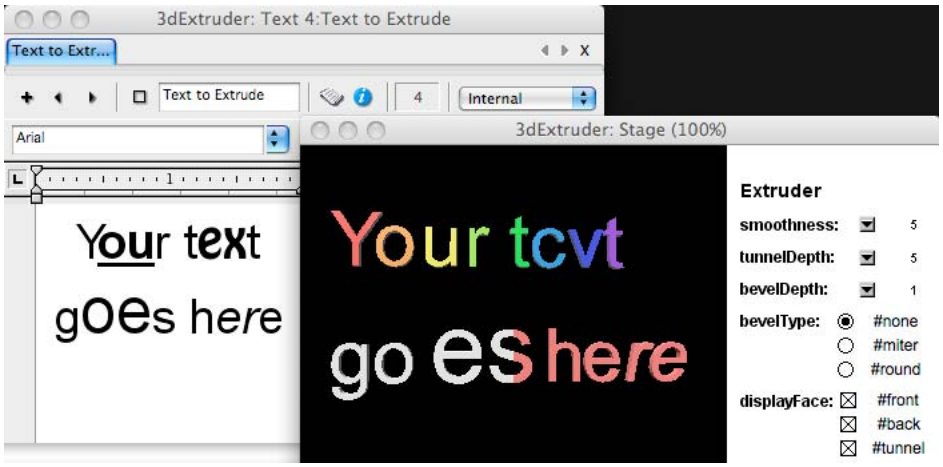
- extruderResource.bevelType: #none, #miter, or #round. The default value is #none. If #none is used then bevelDepth (see below) is ignored. This determines the shape of the bevel that may be added between the vertical faces and the horizontal tunnel.
- extruderResource.bevelDepth: A floating point value between 0.0 and 10.0. The default value is 1.0. This property does not have any effect if bevelType (see above) is #none. This determines how many world units the extruder resource is inflated to the top, bottom, and to the sides. If the bevelDepth is greater than half the tunnelDepth, the resulting mesh may look strange.

Multiple meshes

The extruder3d() command creates a separate mesh for each letter. This means that you can apply a different shader to each letter if you wish. See the shaderList section of "Models" on page 159 for more details.

Fonts and font properties

In Director 11.5, the extruder3d() command may yield unexpected results if some characters are in a different font, font size, or font style from others. The following screenshot illustrates some of the issues:



Changing font or font properties in the source text can lead to unexpected results

In the Text to Extrude text member, the alignment is set to #center. In addition, the middle two letters of each word are given different properties from the rest of the word. As you can see:

- textChunk.alignment is ignored.
- Bold and underlined text is treated as plain text.
- Changing the font leads to the wrong letters being displayed, and no change in font.
- If the fontSize of a given character is set that fontSize is applied to the next character.
- If a given character is set to appear in an italic typeface, the italic typeface is applied to the next character.

Note: *For best results, use the same font and font properties for the entire text of the source text member, or use a custom vertexList as explained below.*

Using a custom vertexList

After the extruder resource has been created, you can change the vertexList of the extruder on the fly. The shape of all models that use the extruder resource are updated to reflect the new geometry.

```
-- Lingo and JavaScript syntax
vExtruder = member("Text to Extrude").extrude3D(member("3D"))
vModel = member("3D").newModel("Extrusion", vExtruder)
vExtruder.vertexList = member("VectorShape").vertexList
```

In Director 11.5, extruderResource.vertexList is a write-only property. You cannot retrieve the vertexList from an extruder resource after it is set, as the following examples show:

```
-- Lingo syntax
put member("3D").model("Extruded").resource.vertexList
-- <Null>
// JavaScript syntax
trace(member("3D").getPropRef("model", 1).resource.vertexList);
// null
```

To see an example of this in action, download and launch the movie Extruder.dir.

Note: *The 2D co-ordinates of a vectorShape member are measured only rightwards and downwards, whereas the 3D co-ordinates of a model resource are measured rightwards, upwards, and forwards. As a result, the shape of an #extruder resource appears to be flipped vertically.*

Texture mapping

An #extruder resource has only one mesh. The same shader is applied to both front and back, as a mirror image, and it is smeared along the sides. If you want to create an modelResource that appears to be extruded, and which has separate shaders on the front, back and tunnel, use a #mesh resource. See "Creating a mesh resource" on page 170 for details.

Creating a mesh resource

The mesh generator is the most complex model resource. It allows you to create models with unique geometries at runtime. Some examples of typical uses are:

- Mathematically simple volumes
- Lathed shapes
- Walls for a building
- Low-polygon models used for collision detection
- Terrain meshes

Writing a script to create a custom mesh requires a good visualization of 3D space, a good grasp of a number of low-level 3D concepts, concentration, and plenty of patience. However, you can find scripts that generate a mesh for you, from information that you provide in some of the demo movies that accompany this manual.

Do-it-yourself mesh movies

- See "Mesh resources" on page 72 for a movie that allows you to create a 2D shape by setting points in a vectorShape member, and then transforms this into a 3D model by rotating the 2D shape around a vertical axis. The movie can generate scripts which recreate the models in your own movies.
- See "Not walking through objects" on page 222 for a script that generates a low-polygon ribbon mesh for use with collision detection. The script uses the vertexPoints of a VectorShape member to generate the shape of the ribbon.
- See "Following a pre-defined path" on page 220 for scripts that generate a tube from a curve defined by a vectorShape member.

- See "Hugging Terrain" on page 238 for a simple script that generates a rectangular terrain from data in a height-map Bitmap member. White pixels are high points, black pixels are low points, and gray pixels create a gradient in between. The Bitmap member's regPoint is used as the origin for the terrain mesh.
- See "Terrains" on page 319 for a more comprehensive script that generates a terrain from a height-map. This script also automatically creates a terrain object which uses the Dynamiks Physics Xtra extension to ensure that rigidBody objects do not pass through the terrain mesh. The mesh generator primitive's type property is #mesh and is created by the member's newMesh() method. The parameters included with that method describe how large the mesh will be.

You can use the mesh deform modifier to manipulate vertex positions at runtime for #mesh or any other type of model resource. You can also use the #mesh primitive to change mesh properties directly.

Ingredients

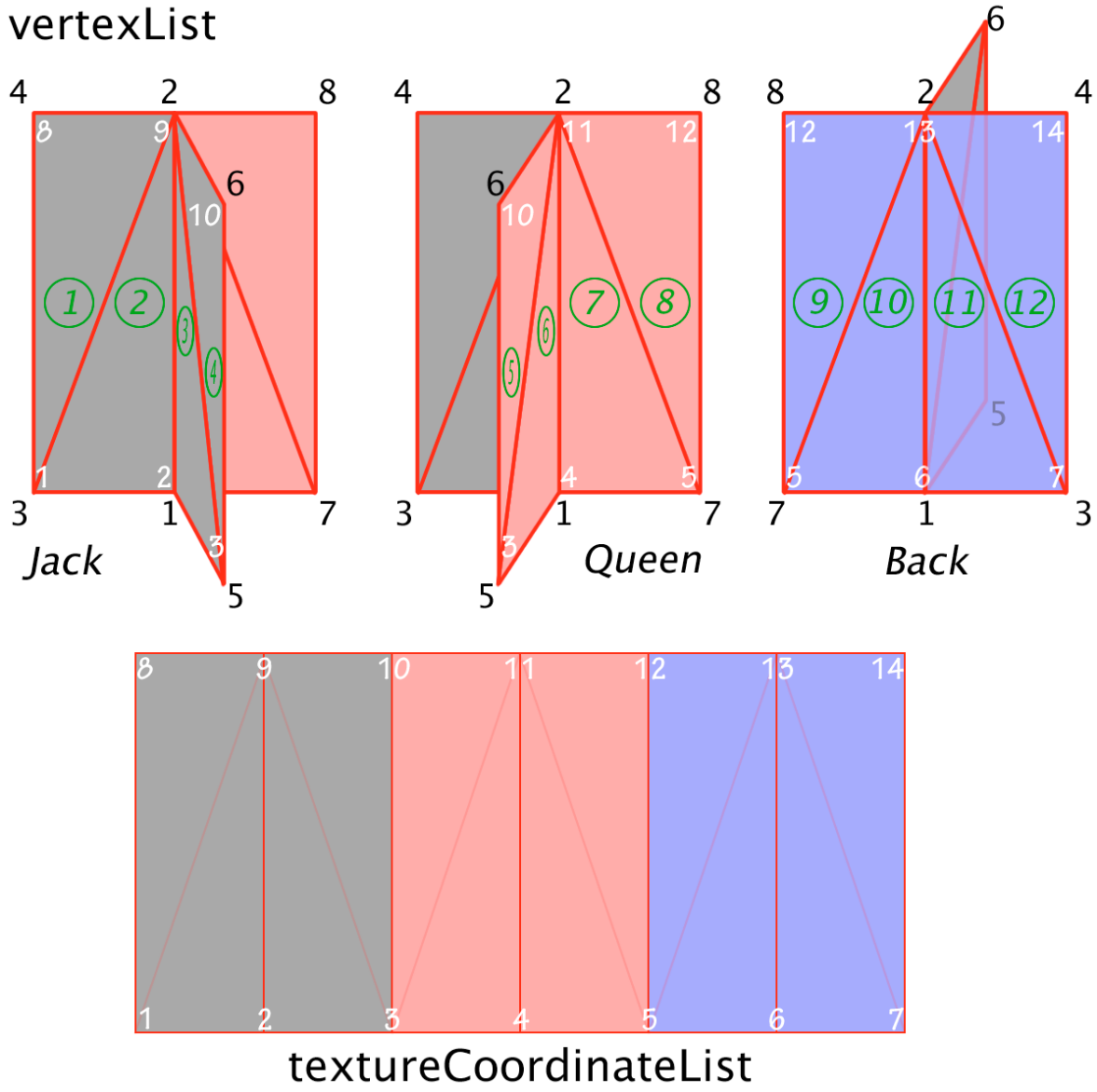
All types of model resource (except for the Particle resource) store five types of information:

- A list of vertex points
- A list of normal directions
- A list of colors
- A list of texture coordinates
- Information on each of the faces. This information includes:
  - Which vertex point is at each corner of the triangular face
  - The color of each vertex
  - The normal at each vertex
  - The shader for the face (this determines which mesh it belongs to)
  - The texture coordinates for the face
  - The faces that share a neighboring edge

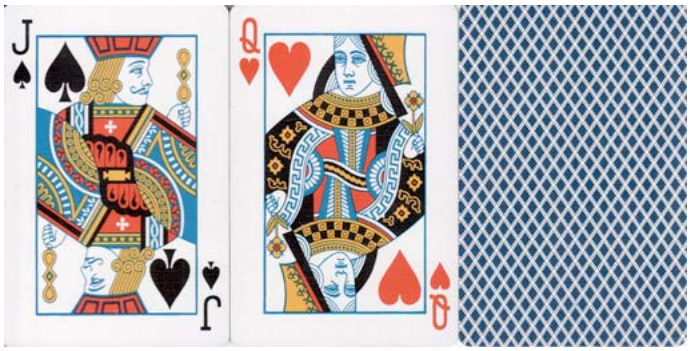
To create a model resource of the type #mesh, provide all this information, or rely on appropriate default values. Imagine that you want to create a trick playing card which can be used to display either a Jack of Spades or a Queen of Hearts. To help you visualize this, download and launch the movie TrickCard.dir.

You can simulate a trick playing card with 8 vertices and 12 triangular faces

Launch the TrickCard.dir movie, then use the left and right arrow keys to rotate the camera around the card, so that you can familiarize yourself with the shape of the card model. The diagram below shows the index numbers of the vertex points used to create the card in black and the index numbers of the texture coordinate points in white. It shows the index number of the faces in green, inside a circle. It shows three different views of the card in the top row, and the layout of the bitmap image for the texture in the bottom row.



The index positions of vectors in the vertexList (black) and points in the textureCoordinateList (white)



The 512x256 bitmap image used for the texture that is applied to the model

Defining a vertexList

In the Trick Card behavior in the movie, the mCreateMesh() handler starts by defining eight points for the vertexList. Compare the positions in this list with the index numbers shown in black in the illustration above.

```
vVertexList  = []
vVertexList.append(vector(  0,   0,  0)) --  1 center base
vVertexList.append(vector(  0, 256,  0)) --  2 center top
vVertexList.append(vector(-96,   0,  0)) --  3 left   base
vVertexList.append(vector(-96, 256,  0)) --  4 left   top
vVertexList.append(vector(  0,   0, 96)) --  5 front  base
vVertexList.append(vector(  0, 256, 96)) --  6 front  top
vVertexList.append(vector( 96,   0,  0)) --  7 right  base
vVertexList.append(vector( 96, 256,  0)) --  8 right  top
```

Defining the faces

The next step is to define the faces. Each face is defined by a list of three integers; each integer represents the index position of a vertex in the vertexList that was defined above. For example, face 1 is defined as [3, 2, 4]. This means:

- vVertexList[3] or vector(-96, 0, 0) at the bottom left
- vVertexList[2] or vector(0, 256, 0) at the top center
- vVertexList[4] or (vector(-96, 256, 0) at the top left

There are a total of 12 faces. Here is the complete definition of the list:

```
vFaceList = []
vFaceList.append([3, 2, 4]) --  1   left rectangle (front)
vFaceList.append([3, 1, 2]) --  2 *
vFaceList.append([1, 5, 2]) --  3 * flap (left)
vFaceList.append([5, 6, 2]) --  4
vFaceList.append([5, 2, 6]) --  5   flap (right)
vFaceList.append([5, 1, 2]) --  6 *
vFaceList.append([1, 7, 2]) --  7 * right rectangle (front)
vFaceList.append([7, 8, 2]) --  8
vFaceList.append([7, 2 ,8]) --  9  right rectangle (back)
vFaceList.append([7, 1, 2]) -- 10 *
vFaceList.append([1, 3, 2]) -- 11 * left rectangle (front)
vFaceList.append([3, 4, 2]) -- 12
```

Compare the items in vFaceList with the illustration above.

textureCoordinateList

Thirdly, the mCreateMesh() handler defines a series of textureCoordinates. These are measured as a proportion of the width and the height of the texture image. The point [0.0, 0.0] is in the bottom left corner of the texture image, and the point [1.0, 1.0] is at the top right corner. Compare these coordinates with the index numbers in white in the illustration above.

```
vTextureCoordinateList = []
-- Bottom of card
vTextureCoordinateList.append([0.0,    0.0]) --  1 vTextureCoordinateList.append([0.1667,
0.0]) --  2 Jack
vTextureCoordinateList.append([0.3333, 0.0]) --  3
vTextureCoordinateList.append([0.5,    0.0]) --  4 Queen
vTextureCoordinateList.append([0.6667, 0.0]) --  5
vTextureCoordinateList.append([0.8333, 0.0]) --  6 Back
vTextureCoordinateList.append([1.0,    0.0]) --  7
-- Top of card
vTextureCoordinateList.append([0.0,    1.0]) --  8
vTextureCoordinateList.append([0.1667, 1.0]) --  9 Jack
vTextureCoordinateList.append([0.3333, 1.0]) -- 10
vTextureCoordinateList.append([0.5,    1.0]) -- 11 Queen
vTextureCoordinateList.append([0.6667, 1.0]) -- 12| vTextureCoordinateList.append([0.8333,
1.0]) -- 13 Back
vTextureCoordinateList.append([1.0,    1.0]) -- 14
```

Mapping vertices to texture coordinates

Fourthly, the mCreateMesh() handler creates a list to map the vertex points to the textureCoordinate points. Note that the `vTextureFaceMap` below has 12 entries, and that each entry corresponds to one of the faces in vFaceList above. For example, face 1 will display the left-most triangle in the texture image, as defined by the texture co-ordinates at positions [1, 9, 8].

```
vTextureFaceMap = []
-- Jack
vTextureFaceMap.append([1,  9,  8])
vTextureFaceMap.append([1,  2,  9])
vTextureFaceMap.append([2,  3,  9])
vTextureFaceMap.append([3, 10,  9])
-- Queen
vTextureFaceMap.append([3, 11, 10])
vTextureFaceMap.append([3,  4, 11])
vTextureFaceMap.append([4,  5, 11])
vTextureFaceMap.append([5, 12, 11])
-- Back
vTextureFaceMap.append([5, 13, 12])
vTextureFaceMap.append([5,  6, 13])
vTextureFaceMap.append([6,  7, 13])
vTextureFaceMap.append([7, 14, 13])
```

Defining normals

A normal is a direction vector. Each vertex of each face will have a normal vector associated with it. When the Director Player is calculating how much light or shade to show for each face, it uses the information stored in the normals to determine which direction each vertex is facing in. In the same way that you define a textureCoordinateList and the mapping for each face, you can also define a list of normals and a list to map these normals to each vertex of each face. However, most cases, you can rely on Director to do it for you. Near the end of the mCreateMesh() handler, the meshResource.generateNormals() method will be used. For this reason, you must not define normals. You have to be explicit about this:

```
vNormalCount = 0
```

Defining a colorList

The Trick Card script applies a texture to the model. In other circumstances, you may want to give each vertex of each face a specific color. In this case, the best color to use for all vertices of all faces is white.

Note: *The default blendFunction of a standard shader is #blend. If you set the color of the faces of your mesh to something other than white, any texture you apply is blended with the mesh color. You can cancel out this blending by setting the blendFunction of the shader to #replace. See* *"Standard shaders" on page 131* *for more details.*

```
vColorList = [rgb(255, 255, 255)]
```

Creating the mesh

The mCreateMesh() handler now has all the information it needs to create a usable mesh modelResource. This needs to be done in five steps:

1. Use member3D.newMesh() to create a mesh resource.
2. Set the vertexList, colorList, and textureCoordinateList properties of the mesh resource. In your own projects, you may also want to set the normalList at this stage.
3. For each face in the mesh, set the vertices, colors, and textureCoordinates. In your own projects, you may also want to set the shader and normals properties for each face.
4. Use meshResource.generateNormals() to generate normals for each face automatically.
5. Use meshResource.build()to compile all the information that you have provided into a modelResource.

Note: *You can create a multi-mesh #mesh resource by defining which shader each face must use.*

Here is the code for each of the five steps.

```
 -- 1. Create a new mesh
vMesh = member("3D").newMesh( \
"Trick Card Mesh", \
vFaceList.count, \
vVertexList.count, \
vNormalCount, \
vColorList.count, \
vTextureCoordinateList.count \
)
-- 2. Apply the vertex, color, face and texture data
vMesh.vertexList = vVertexList
vMesh.colorList  = vColorList
vMesh.textureCoordinateList = vTextureCoordinateList
-- 3. Define each of the faces
vCount = vFaceList.count
repeat with ii = 1 to vCount
vMesh.face[ii].vertices           = vFaceList[ii]
vMesh.face[ii].colors             = [1, 1, 1]
vMesh.face[ii].textureCoordinates = vTextureFaceMap[ii]
end repeat
-- 4. Create a flat mesh
vMesh.generateNormals(#flat)
-- 5. Complete the process
vMesh.build()
```

Creating a model from a mesh resource

You can now treat your mesh resource like any other modelResource. The following code:

- Creates a model from the mesh resource
- Creates a texture from a bitmap member named “Trick Card”
- Creates a shader to use the texture
- Applies the shader to the model

```
vMember = member("3D")
vName = "Trick Card"
vModel = vMember.newModel(vName, vMesh)
vTexture = vMember.newTexture(vName)
vTexture.member = member(vName)
vShader = vMember.newShader(vName, #standard)
vShader.texture = vTexture
vModel.shader = vShader
```

Manipulating the vertexList and normalList

You have direct access at all times to the various lists stored by a #mesh resource. See “Manipulating a mesh resource” on page 182 to see how to fold this Trick Card model so that it appears to be either a Jack of Spades or a Queen of Hearts.

Creating a terrain mesh

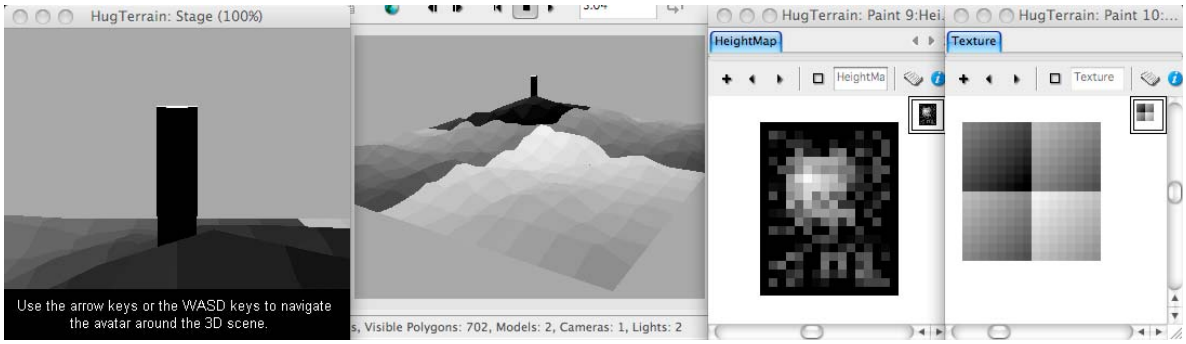
A terrain mesh resembles a 2D plane that has been deformed to create an interesting landscape. Normally, the end user only sees the top surface of a terrain mesh, and no objects falls through it. This means that you have to use some method of collision detection on the terrain.

You can use the following techniques for collision detection:

- Ray casting. See "Ray casting" on page 321.
- Using the Collision Modifier. See "Collision modifier" on page 280.
- Using the Dynamiks Physics engine. See "Terrains" on page 319.

Note: *You can use physics.createTerrain() to create an invisible Terrain object that interacts with rigidBodies. The createTerrain() method indirectly relies on a Matrix object. This imposes a particular structure on the terrain mesh. In particular, the origin point of the mesh resource used with a Terrain object needs to be in one corner; all x and z coordinates need to be positive. For more details on creating a terrain mesh for use with a Physics Terrain object, see "Terrains" on page 319.*

The technique described in this section allows you to create a terrain-shaped mesh with the origin point in any position. The process of generating the terrain mesh is similar to the one described at "Terrains" on page 319. The difference concerns the relative positions of the vertices and hence of the texture mappings. To see the technique in action, download and launch the movie HugTerrain.dir.



The height of the terrain is read in from a grayscale bitmap

Data required to create a terrain mesh

The HugTerrain.dir movie uses a movie script named Create Terrain Mesh to generate a terrain mesh automatically. In order to create a terrain mesh, you need to provide:

- A unique name for the model and resource to be created
- The dimensions of the mesh resource that you wish to create, in world units
- A mapping to connect x, z positions on the ground plane to y height values
- An origin point for the mesh

From this information the Create Terrain Mesh script can calculate:

- A vertex list
- A face list
- A textureCoordinate list

See "Creating a mesh resource" on page 170 for an example movie where these lists are created manually in preparation for generating a mesh resource.

Using a bitmap image as a height map

One simple way to define the height of each point in a terrain is to use the grayscale image of a bitmap member. In this example (HugTerrain.dir), white pixels are high and black pixels are low.

Using a Bitmap member also gives you a simple way to define the origin point for the mesh resource: you can use the member.regPoint of the Bitmap member. A Bitmap member has a width and a height, measured in pixels. A bitmap of `n x m` pixels generates a rectangular terrain of `(n - 1)` columns and `(m - 1)` rows. Each pixel can store one of 256 different values, from 0 to 255. To convert these numbers into world unit dimensions, you need to apply a different scale to each axis.

Here is a handler that you can use to find what scale to use for each axis. It returns a list containing three floating-point values, in the format [<xScale>, <yScale>, <zScale>].

```
-- Lingo syntax
on GetScale(aBitmap, xUnits, yUnits, zUnits)
    vColumns = float(aBitmap.width - 1)
    vRows    = float(aBitmap.height - 1)
    vScale  = []
    vScale.append(xUnits / vColumns)
    vScale.append(yUnits / 255.0)
    vScale.append(zUnits / vRows)
    return vScale
end GetScale
```

Example usage:

```
vScaleList = GetScale(member "HeightMap", 100, 39, 64)
put vScaleList
-- [6.6667, 0.1529, 3.3684]
```

You can now use the Bitmap's grayscale image data, its regPoint, and the scale list that you have just created to generate a series of vertex points.

```
on GetVertexListFromBitmap(aBitmap, aScale)
    vVertexList = []
    vScaleX = aScale[1]
    vScaleY = aScale[2]
    vScaleZ = aScale[3]
    vImage    = aBitmap.image
    vColumns  = vImage.width - 1
    vRows     = vImage.height - 1
    vRegPoint = aBitmap.regPoint
    vOffsetX  = vRegPoint.locH * vColumns / float(vColumns + 1)
    vOffsetZ  = vRegPoint.locV * vRows / float(vRows + 1)
    repeat with zz = 0 to vRows
        repeat with xx = 0 to vColumns
        vX      = (xx - vOffsetX) * vScaleX
        vPixel  = vImage.getPixel(xx, zz).red
        vY      = vPixel * vScaleY
        vZ      = (zz - vOffsetZ) * vScaleZ
        vVector = vector(vX, vY, vZ)
        vVertexList.append(vVector)
        end repeat
    end repeat
    return vVertexList
end GetVertexListFromBitmap
```

Below is the output generated by a Bitmap member named “3x3” that displays 8 black pixels in a square around a central white pixel. The member's regPoint is at point(3, 0), on the top right edge of the black square. The scale has been chosen to create a mesh resource that will be 20 units wide along the xAxis and 30 units along the zAxis. The maximum height of the terrain will be 20 units. The output has been edited and arranged neatly in columns, so that you can see that the height of the eight outer vertices is 0 and the height of the central vertex is 10.

```
put GetVertexListFromBitmap(member("3x3"), [10,0.0392,15])
-- [vector(-20,0, 0), vector(-10, 0, 0), vector(0,0, 0),
vector(-20,0,15), vector(-10,10,15), vector(0,0,15),
vector(-20,0,30), vector(-10, 0,30), vector(0,0,30)]
```

You may like to step through the lines of code in the GetVertexListFromBitmap()handler to determine how these values were calculated.

Generating the face list

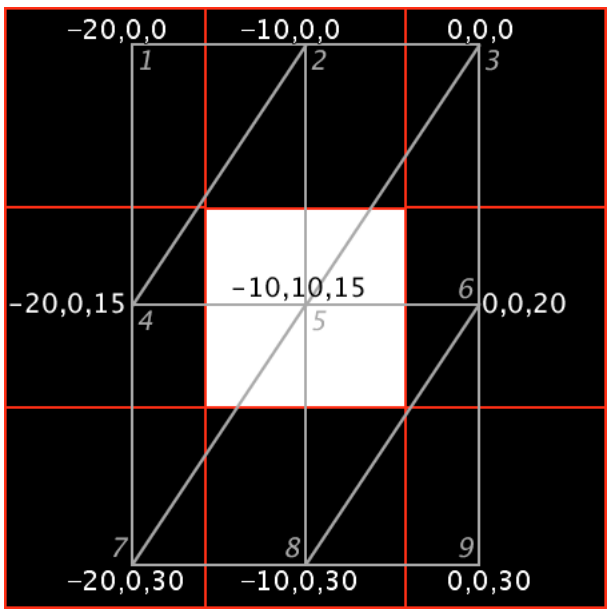
To generate the face list, there is no need to know the actual vertex points. All that is required is to know the number of vertices in each column and row.

```
on GetFaceList(aWidth, aLength)
    vFaceList    = []
    vColumns = aWidth - 1 -- n vertices per row gives (n - 1) faces
    vRows = aLength - 2 -- m vertices => (m - 1) faces, start at 0
    repeat with zz = 0 to vRows
    vRowAdjust = (zz) * aWidth
    vRowAdjust = [vRowAdjust, vRowAdjust, vRowAdjust]
        repeat with xx = 1 to vColumns
        vFace = [xx,  xx + aWidth, xx + 1] + vRowAdjust
        vFaceList.append(vFace)
        vFace = [xx + aWidth, xx + aWidth + 1, xx + 1] + vRowAdjust
        vFaceList.append(vFace)
        end repeat
    end repeat
    return vFaceList
end GetFaceList
```

To continue with the 3x3 terrain:

```
-- [[1, 4, 2], [4, 5, 2], [2, 5, 3], [5, 6, 3], [4, 7, 5], [7, 8, 5], [5, 8, 6], [8, 9, 6]]
```

The diagram below illustrates the positions of the vertex points and how the points are linked together to form faces.



3 x 3 pixels, overlaid with the vertexList and face map

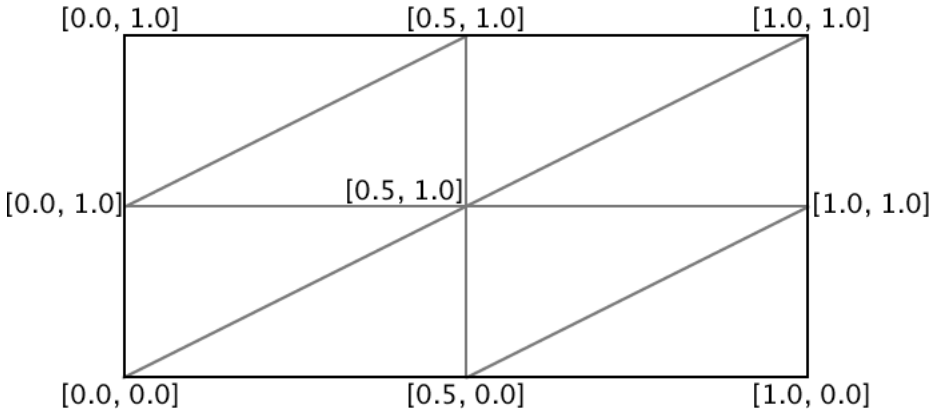
Generating the TextureCoordinate List

To generate the textureCoordinate list, as with the face list, all that is required is to know the number of vertices in each column and row. To create the textureCoordinates in the same order as the vertices, the GetTextureCoordinateList() needs to start in the top left-hand corner, or in the lowest-numbered column of the highest-numbered row. It will proceed left to right (from the first to the last column), and then top to bottom (from the last row to the first row).

```
on GetTextureCoordinateList(aWidth, aLength)
    vTextureCoordinateList  = []
    vColumns = float(aWidth - 1)
    vRows    = float(aLength - 1)
    zz = aLength
    repeat while zz
    zz = zz - 1
    vZ = zz / vRows
        repeat with xx = 0 to vColumns
        vX = xx / vColumns
        vCoordinate  = [vX, vZ]
        vTextureCoordinateList.append(vCoordinate)
        end repeat
    end repeat
    return vTextureCoordinateList
end GetTextureCoordinateList
```

Below is the output for the 3 x 3 terrain, together with an illustration of the positions inside an arbitrary texture image.

```
put GetTextureCoordinateList(3, 3)
-- [[0.0, 1.0], [0.5, 1.0], [1.0, 1.0],
[0.0, 0.5], [0.5, 0.5], [1.0, 0.5],
[0.0, 0.0], [0.5, 0.0], [1.0, 0.0]]
```



The relative positions of the texture coordinates within a texture image

Texture mappings for each vertex of each face

You will recall that the face list that was generated automatically looks like this:

```
-- [[1, 4, 2], [4, 5, 2], [2, 5, 3], [5, 6, 3], [4, 7, 5], [7, 8, 5], [5, 8, 6], [8, 9, 6]]
```

The list of textureCoordinates was generated in the same order as the vertexList. This means that the data used to define the faces can also be used to define the texture mappings.

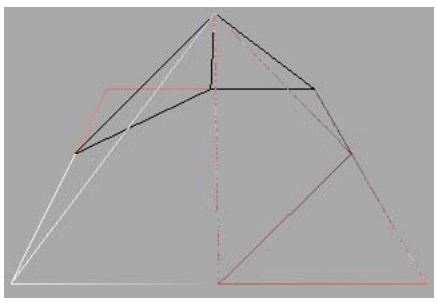
Creating the mesh resource

All the data needed to generate the mesh resource is now available. Here is a Lingo function that puts all this data together, and returns a mesh resource.

```
on GetMesh(a3DMember,aName,aFaceList,aVertexList,aTextureList)
    vFaceCount    = aFaceList.count
    vNormalCount = 0 -- normals to be created automatically
    vColorList    = [rgb("#FFFFFF")]
    vMesh = a3DMember.newMesh( \
    aName, \
    vFaceCount, \
    aVertexList.count, \
    vNormalCount, \
    vColorList.count, \
    aTextureList.count)
    -- Apply the vertex data
    vMesh.vertexList            = aVertexList
    vMesh.colorList             = vColorList
    vMesh.textureCoordinateList = aTextureList
    -- Map data to each face
    repeat with ii = 1 to vFaceCount
    vFaceData = aFaceList[ii]
    vMesh.face[ii].vertices = vFaceData
    vMesh.face[ii].textureCoordinates = vFaceData
    end repeat
    -- Create a smooth mesh
    vMesh.generateNormals(#smooth)
    vMesh.build()
    return vMesh
end GetMesh
```

Using the mesh

All this code allows you to provide just a unique name, a Bitmap image and the desired dimensions of your terrain mesh.



A wire frame view of the generated terrain

The 3 x 3 example allowed you to visualize each step in the calculation of a very simple terrain. However, the Create Terrain Mesh script can handle much larger height-map images with ease. It can even use images that are created from scratch at runtime. Terrains that are created by this script are simple models. They do not provide any automatic collision detection.

For details on how to use ray casting for collision detection, see "Ray casting" on page 321. For details on how to create a terrain model that is integrated with a Physics Terrain object, see "Terrains" on page 319.

Mesh resources with multiple shaders

You can create a mesh resource with multiple shaders. This allows you to use different materials in the same model. For example, you can create a cartoon television set with a screen that shows a realistic image. To see an example of a single mesh with multiple shaders, download and launch the movie Frame.dir.

The Frame.dir movie creates a mesh with two large faces in the center (faces 1 and 2), which form a rectangular canvas area, and 26 other faces to create a frame around the canvas (faces 3 to 28). Below is a handler that accepts five parameters:

- a3DMember: a 3D cast member
- aName: a unique name, not yet used for a model resource
- aVertexList: a list of vector positions
- aFaceList: a list of three-item lists, used to define the faces of the mesh
- aShaders: a list of two shaders, one for the canvas, one for the frame

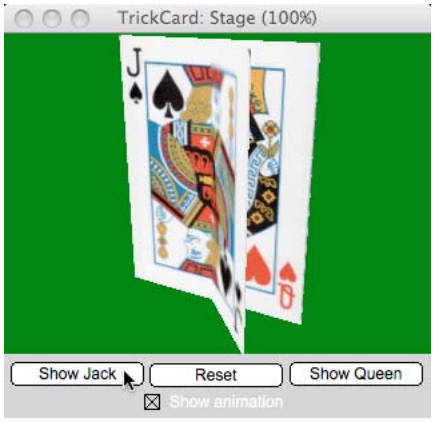
The CreatePicture() creates a new model resource, and applies the first shader to the two central faces. It applies the second shader to all the other faces.

```
on CreatePicture(a3DMember,aName,aVertexList,aFaceList,aShaders)
    vFaceCount = aFaceList.count
    vMesh = member("3D").newMesh( \
    aName, \
    vFaceCount, \
    aVertexList.count \
    )
    vMesh.vertexList = aVertexList
    vMesh.colorList  = []
    repeat with ii = 1 to vFaceCount
    vMesh.face[ii].vertices = aFaceList[ii]
    if ii < 3 then
        vMesh.face[ii].shader = aShaders[1]
    else
        vMesh.face[ii].shader = aShaders[2]
    end if
    end repeat
    vMesh.generateNormals(#flat)
    vMesh.build()
    return vMesh
end CreatePicture
```

Note: *In Director 11.5, if you add a standard shader to the faces of a mesh resource, the diffuse color of the shader is not correctly applied unless the shader's useDiffuseWithTexture property is TRUE, even if the shader has no texture. If the mesh resource has textureCoordinates applied to its faces, the diffuse color of the shader may fail to work altogether, even if the shader's useDiffuseWithTexture property is set to TRUE.*

Manipulating a mesh resource

In "Creating a mesh resource" on page 170, you can see step-by-step how to create a trick playing card which can be used to display either a Jack of Spades or a Queen of Hearts. To try this for yourself, download and launch the movie TrickCard.dir.



Check Show Animation then click on Show Jack to fold the trick card

If you click the Show Jack button, the flap showing the Queen is folded back on itself. If you click the Show Queen button, the same flap is folded the other way, to hide the Jack. If you click on Reset, the card is restored to its initial position. If you check the Show Animation check box, then the folding process is animated.

Modifying the vertexList

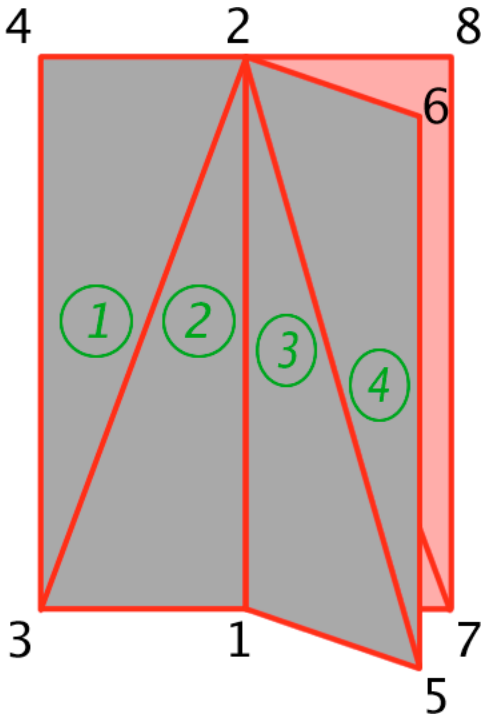
To move the flap at the front of the card, you need to change the entries in the vertexList of the mesh resource. Here is what the vertexList looks like initially. Note the values for the vertices at index numbers 5 and 6:

```
put member("3D").model(1).resource.vertexList
-- [vector( 0.0000, 0.0000, 0.0000 ),
vector( 0.0000, 256.0000, 0.0000 ),
vector( -96.0000, 0.0000, 0.0000 ),
vector( -96.0000, 256.0000, 0.0000 ),
vector( 0.0000, 0.0000, 96.0000 ),
vector( 0.0000, 256.0000, 96.0000 ),
vector( 96.0000, 0.0000, 0.0000 ),
vector( 96.0000, 256.0000, 0.0000 )]
```

Here is how to set the vertexList so that the card model folds back to show the Jack:

```
member("3D").model(1).resource.vertexList[5] = vector(96, 0, 1)
member("3D").model(1).resource.vertexList[6] = vector(96, 256, 1)
```

The vertex points in at positions 5 and 6 in the vertexList are now practically aligned with the vertex points in at positions 7 and 8. The illustration below helps you to visualize this.

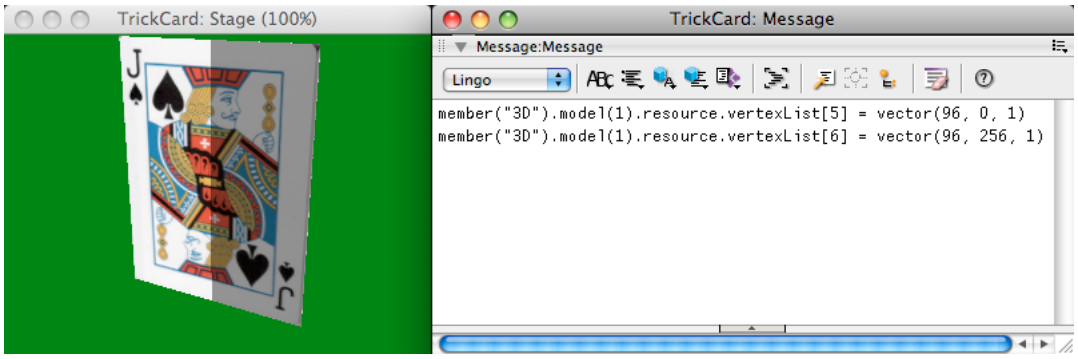


The index numbers of faces (green) the vectors in the mesh's vertexList (black)

Note: *There is a slight difference between the positions of the vertex points 5 and 7 and between the positions of the vertex points 6 and 8. If the positions were identical, the face of the queen may appear through either the Jack or the back of the card. Keeping a slight gap between the faces enables Director to know which face must appear on top.*

Modifying the normalList

If you execute the commands given above in the Message window, you will see a problem with the lighting of the faces whose vertex points have changed.



Modifying the vertexList does not automatically update the lighting normals

The faces that have been rotated are still lit as if they were in their original position. It is easy to correct that. First you need to discover which entries in the mesh resource's normalList need to be updated for face 3 and face 4:

```
put member("3D").model(1).resource.normalList
-- [vector( 0.0000, 0.0000, 1.0000 ),
vector( 0.0000, 0.0000, 1.0000 ),
vector( -1.0000, 0.0000, 0.0000 ),
vector( -1.0000, 0.0000, 0.0000 ),
vector( 1.0000, 0.0000, 0.0000 ),
vector( 1.0000, 0.0000, 0.0000 ),
vector( 0.0000, 0.0000, 1.0000 ),
vector( 0.0000, 0.0000, 1.0000 ),
vector( 0.0000, 0.0000, -1.0000 ),
vector( 0.0000, 0.0000, -1.0000 ),
vector( 0.0000, 0.0000, -1.0000 ),
vector( 0.0000, 0.0000, -1.0000 )]
```

This indicates that the mesh resource's normalList still thinks that faces 3 and 4 are looking towards the left in the direction defined by vector(-1, 0, 0). To correct this, execute the two lines of code below:

```
member("3D").model(1).resource.normalList[3] = vector(0, 0, 1)
member("3D").model(1).resource.normalList[4] = vector(0, 0, 1)
```

The faces are now turned to face the front, along the world's zAxis, and the lighting now looks correct.

Showing the flap at any angle

You need to do a little trigonometry in order to be able to fold the flap to any angle. The following handler accepts an angle in degrees.

```
on ShowFacesAt(aAngle)
    vCard = member("3D").model(1).resource
    vSine        = sin(aAngle * pi / 180)
    vCosine      = cos(aAngle * pi / 180)
    vQueenNormal = vector( vCosine, 0, -vSine)
    vJackNormal  = vector(-vCosine, 0, vSine)
    vX      = vSine * 96.0
    vZ      = max(vCosine * 96.0, 1)
    vCard.vertexList[5] = vector(vX,   0,  vZ)
    vCard.vertexList[6] = vector(vX, 256,  vZ)
    vCard.normalList[3] = vJackNormal
    vCard.normalList[4] = vJackNormal
    vCard.normalList[5] = vQueenNormal
    vCard.normalList[6] = vQueenNormal
end ShowFacesAt
```

To show the Jack, the angle must be positive, to show the Queen it must be negative. The angle must be in the range -90° to +90°. Note that the positions for only 2 vertices are reset, but this moves 4 separate faces, so 4 sets of normals need to be updated.

Modifying other properties

You can modify any property of a #mesh resource, including:

- meshResource.vertexList
- meshResource.textureCoordinateList
- meshResource.normalList
- meshResource.colorList

You can also modify the properties of individual faces. However, these changes do not take effect until you issue a meshResource.build() command. The build() command also resets all shaders attached to all models that use the mesh resource to the default value. Using the build() command can take a significant number of milliseconds, so it is not something that you must plan to do on every frame. You can access the following properties of each face:

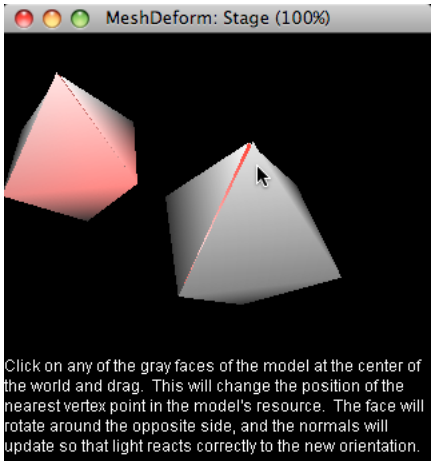
- meshResource.face[aIndex].vertices
- meshResource.face[aIndex].colors
- meshResource.face[aIndex].normals
- meshResource.face[aIndex].shader
- meshResource.face[aIndex].textureCoordinates

MeshDeform modifier

Adding the #meshDeform modifier to a model give you access the structure of the model's modelResource. If you modify the geometry of the modelResource of a model, all models that use the given modelResource will be affected.

Note: *You do not need to use the #meshDeform modifier with models whose resource is of the type #mesh: you can access the properties of a #mesh resource directly.*

To see the #meshDeform modifier in action, download and launch the movie MeshDeform.dir.



The MeshDeform.dir movie allows you to drag individual vertices of a primitive model

The MeshDeform.dir movie shows two models created with the same primitive modelResource. When you deform one of the models, the other is deformed in an identical way. This article starts with a description of the data to which the meshDeform modifier gives you access, and then describes how this data is used in the MeshDeform.dir movie. The diamond-shaped models in the MeshDeform movie provide the example data shown below. The MeshDeform.dir movie uses a number of operations in 3D mathematics to calculate the new positions of the vertices and the new direction of the vertex normals. See "3D mathematics" on page 361 for more details on the operations themselves.

Adding the meshDeform modifier to a model

The following command adds the #meshDeform modifier to model 2, named "DeformMe", of the member "3D".

```
-- Lingo syntax
member("3D").model("DeformMe").addModifier(#meshDeform)
put member("3D").model("DeformMe").modifier
-- [#meshDeform]
// JavaScript syntax
member("3D").getPropRef("model", 2).addModifier(symbol("meshDeform"));
1
trace(member("3D").getPropRef("model", 2).modifier)
// <[#meshDeform]>
```

Note: *If you add the #meshDeform modifier to one model with a given modelResource, all other models that use the same modelResource will have access to the modifier too. Adding the #meshDeform modifier to a model in fact applies the meshDeform to the model's resource, which may be shared with other models.*

Removing the modifier

You can use the command aModel.removeModifier() to remove the #meshDeform modifier.

```
member("3D").model("DeformMe").removeModifier(#meshDeform)
// JavaScript syntax
member("3D").getPropRef("model", 2).removeModifier(symbol("meshDeform"));
```

Meshes and faces

The #meshDeform modifier groups the faces of a given model in meshes. Every mesh in a modelResource is associated with one of the shaders in the shaderList of the model. Use aModel.meshDeform.mesh.count to determine how many meshes a model has. The result will always be identical to aModel.shaderList.count. You can use aModel.meshDeform.face.count to determine the total number of faces in a mesh. However, to access any of the face properties, you must pass through the appropriate mesh.

Mesh properties

Each face of a mesh has three vertices. Each of these vertices has five properties associated with it:

- vertex position
- normal
- textureCoordinate
- color
- neighbor

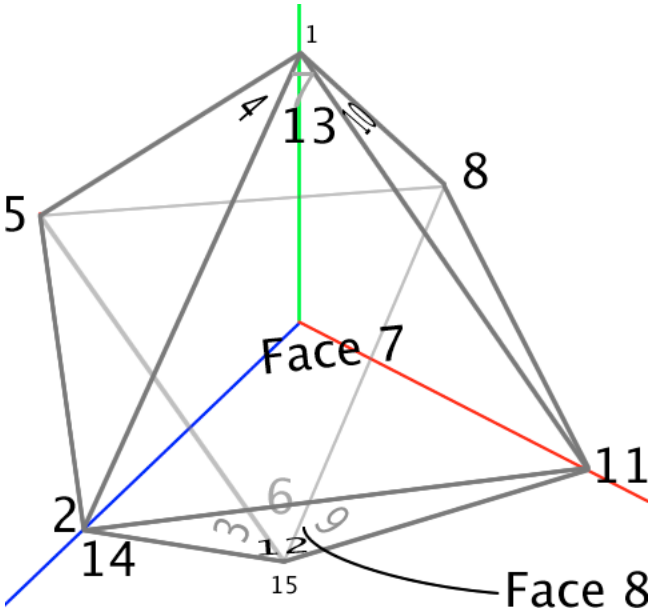
The values of all these properties, except for 'neighbor', is stored in a separate list accessible through the mesh. The lists are all ordered in the same way. The nth item in any list gives the value for one property at the nth vertex point.

- aModel.meshDeform.mesh[meshIndex].vertexList stores the list of all the vertex points used by the modelResource. The same vector position may appear multiple times in the vertexList. Two faces can share a vertex position, but actually use different vertex points to express that position. If this is the case, moving one of the vertex points will separate the faces at that point. Other vertices may be shared by many faces. In this case, moving the vertex point will deform all the faces.
- aModel.meshDeform.mesh[meshIndex].normalList stores a list of the normals to the surface of the mesh at each vertex. If two faces share the same vertex they will also share the same normal at that point. If the two faces have two different vertex points at an identical position, then the values for the normal at that position may be different for the two faces.
- aModel.meshDeform.mesh[meshIndex].textureCoordinateList stores the list of the texture coordinates associated with each vertex point. If two faces share the same vertex the will share the same textureCoordinateList value.

- aModel.meshDeform.mesh[meshIndex].colorList may be a list of the individual colors at each vertex point. However, this list may have no entries if no colors are defined. Nonetheless, accessing `aModel.meshDeform.mesh[meshIndex].colorList[colorIndex]` does not cause an error, even if `colorIndex` is an out-of-range integer (see Accessing face values below).

Example values

In the MeshDeform.dir movie, a model resource is created using the #sphere primitive (see sphere properties). This leads to a shape where all the normals face outwards from the center of the model. The illustration below shows the index numbers for the various vertices. You will notice that the positions vector(0, 25, 0) and vector(0, -25, 0) are defined multiple times. Vertices 1 and 15 are not used to create any faces.



The positions of the various vertex points in the mesh resource

The table below shows the values at each index point in the vertexList, normalList, textureCoordinateList, and colorList for the first (and only) mesh in the resource.

index	vertexList	normalList	textureCoordinateList	colorList
1	vector(0, 25, 0)	vector(0, 1, 0)	[0.9999, 0.9999]	
2	vector(0, 0, 25)	vector(0, 0, 1)	[0.9999, 0.4999]	
3	vector(0,-25, 0)	vector(0,-1, 0)	[0.9999, 0.0001]	
4	vector(0, 25, 0)	vector(0, 1, 0)	[0.7499, 0.9999]	
5	vector(25, 0, 0)	vector(1, 0, 0)	[0.7499, 0.4999]	
6	vector(0,-25, 0)	vector(0,-1, 0)	[0.7499, 0.0001]	
7	vector(0, 25, 0)	vector(0, 1, 0)	[0.4999, 0.9999]	
8	vector(0, 0,-25)	vector(0, 0,-1)	[0.4999, 0.4999]	
9	vector(0 -25, 0)	vector(0 -1, 0)	[0.4999, 0.0001]	
10	vector(0, 25, 0)	vector(0, 1, 0)	[0.2499, 0.9999]	

index	vertexList	normalList	textureCoordinateList	colorList
11	vector(25, 0, 0)	vector(1, 0, 0)	[0.2499, 0.4999]	
12	vector(0,-25, 0)	vector(0,-1, 0)	[0.2499, 0.0001]	
13	vector(0, 25, 0)	vector(0, 1, 0)	[0.0001, 0.9999]	
14	vector(0, 0, 25)	vector(0, 0, 1)	[0.0001, 0.4999]	
15	vector(0,-25, 0)	vector(0,-1, 0)	[0.0001, 0.0001]	

Face values

To determine the value of each of these properties for a given face, first retrieve the value of `aModel.meshDeform.mesh[meshIndex].face[faceIndex]`. This returns a list containing three integers. These integers give the index numbers of the vertices of the given face.

```
-- Lingo syntax
member("3D").model(1).meshDeform.mesh[1].face[7]
-- [13, 14, 11]
```

Accessing face values

To determine the value of a particular property at a particular vertex, you can use expressions similar to the following:

```
-- Lingo syntax
put member("3D").model(1).meshDeform.mesh[1].vertexList[13]
-- vector( 0.0000, 25.0000, 0.0000 )
put member("3D").model(1).meshDeform.mesh[1].normalList[13]
-- vector( 0.0000, 1.0000, 0.0000 )
put member("3D").model(1).meshDeform.mesh[1].textureCoordinateList[13]
-- [0.0001, 0.9999]
put member("3D").model(1).meshDeform.mesh[1].colorList[13]
-- color( 0 )
// JavaScript syntax
trace(member("3D").getPropRef("model", 1).getPropRef("meshDeform", 1).getPropRef("mesh",
1).vertexList[13]);
// <vector( 0.0000, 25.0000, 0.0000 )>
trace(member("3D").getPropRef("model", 1).getPropRef("meshDeform", 1).getPropRef("mesh",
1).normalList[13]);
// <vector( 0.0000, 1.0000, 0.0000 )>
trace(member("3D").getPropRef("model", 1).getPropRef("meshDeform", 1).getPropRef("mesh",
1).textureCoordinateList[13]);
// <[0.0001, 0.9999]>
```

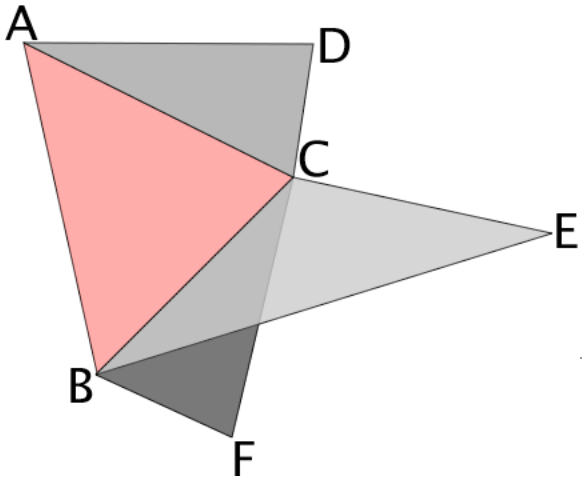
As the example above shows, the colorList for the sphere resource's mesh is empty. However, using Lingo to access a non-existent entry does not result in an error. A default value is returned.

```
 -- color( 0 )
Other invalid Lingo calls will also fail more or less gracefully.
put member("3D").model(1).meshDeform.mesh[1].colorList[123]
-- color( 0 )
put member("3D").model(1).meshDeform.mesh[1].colorList[-1]
-- <Null>
put member("3D").model(1).meshDeform.mesh[1].face[-1]
-- [0, 0, 0]
```

JavaScript, however, applies the rules more strictly. If you attempt to access a non-existent entry, a script error occurs.

Neighbor data

Any given face may have 0 or more neighbors along a given edge. In the illustration below, the face ABC has two neighbors on the edge opposite vertex A, one neighbor on the edge opposite vertex B and no neighbors on the edge opposite vertex C.



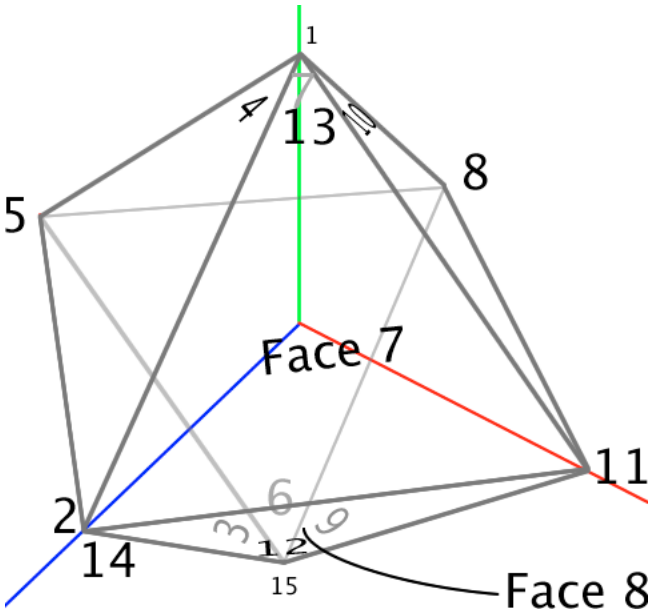
A manifold mesh may have three or more faces meeting at one edge

You can obtain information about the neighboring faces of a given face using the neighbor property. You can use the following Lingo code to find out which face is the neighbor to face seven, opposite vertex 13. First, find where vertex 13 appears in the list of vertices for face 7.

```
put member("3D").model(1).meshDeform.mesh[1].face[7]
-- [13, 14, 11]
```

Now you know that 13 is vertex one in the face list for face seven, you can ask for the neighbor of vertex one:

```
put member("3D").model(1).meshDeform.mesh[1].face[7].neighbor[1]
-- [[1, 8, 2, 1]]
```



The neighbor for face 7 opposite vertex 13 is face 8

The output is in the form of a list of lists. A face may have 0, 1, or many neighbor faces on a given edge. In this case there is only one face, so only one sub-list. The 4 integers in the sublist indicate:

- The meshID
- The faceID
- The position index of the vertex on the far side of the neighbor face
- A Boolean to indicate if the normal of the neighbor face is flipped

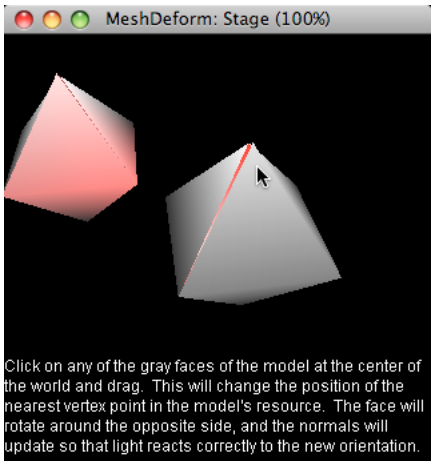
In this case, the output tells you that the neighbor is the face indicated in this expression:

```
put member("3D").model(1).meshDeform.mesh[1].face[8]
-- [14, 12, 11]
```

Note that vertex 12, opposite to vertex 13 is in the 2nd position in the face definition list.

Manipulating the model resource using the meshDeform modifier

The easiest way to understand how all this data is organized is with an example. Launch the movie MeshDeform.dir, and click at the top of the closest face, as shown in the screenshot below.



Click and drag to rotate a face round the opposite edge

On mouseDown, the Mesh Deform behavior in the movie uses modelsUnderLoc() to detect where you clicked. The list returned by call contains the following data:

```
[[#model:model("Deformable octahedron"),
#distance:      109.1726,
#isectPosition: vector( 1.0218, 23.5829, 0.3953 ),
#isectNormal:   vector( 0.5774, 0.5774, 0.5774 ),
#meshID:        1,
#faceID:        7,
#vertices:     [vector( 0.0000, 25.0000, 0.0000 ),
vector( 0.0000, 0.0000, 25.0000 ),
vector( 25.0000, 0.0000, 0.0000 )],
#uvCoord:      [#u: 0.0158, #V: 0.0409]]]
```

The information shown in bold above is used to work out which is the nearest vertex. The handlers executed during the #mouseDown event access the data stored in the meshDeform object. They use this to create a property list containing all the information required to rotate the vertex point around the opposite edge of its face.

Updating normals

It is not enough just to update the position of the vertex point. If the normals of each vertex are not updated, then the face continues to react to light as if it were in its original orientation. In the Mesh Deform behavior, after moving the vertex point, the mMoveVertex() handler calls mUpdateNormals(). This sets the normal of the dragged vertex at right angles to the face, and sets the normals of the two other vertices of that face to the average of the normal to that face and its neighbor.

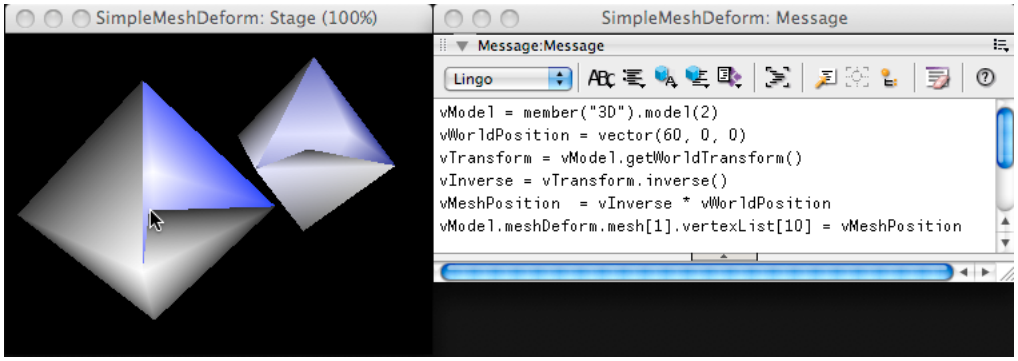
Note: *Wherever vertices are shared between faces, the normals at those vertex points are also shared. If you use the #meshDeform modifieron models that are custom-built for the project, you may need to make sure that the model is specifically constructed to allow the deformations that you want to apply.*

Multiple texture layers

With the #meshDeform modifier, you can add extra texture layers to specific faces. See add (3D texture) and textureLayer.

Dragging in world space and mesh space

There are many possible techniques for dragging the vertex point of modelResource around, using the #meshDeform modifier. To explore a different technique, download and launch the movie SimpleMeshDeform.dir.



In SimpleMeshDeform.dir, you can drag the vertex point in the same plane as the sprite

The technique for dragging a point in the plane parallel to the sprite is described in more detail in "Dragging" on page 250. However, the point being dragged here is defined in terms of the resource space, not the world space. To convert from world space to resource space, the `mMoveVertex()` handler of the Simple Mesh Deform behavior performs a multiplication using the inverse of the transform of the model. The following is the Lingo code that places the vertex 10 of the modelResource of model 2 at the point vector(60, 0, 0) in world coordinates.

```
vModel = member("3D").model(2)
vWorldPosition = vector(60, 0, 0)
vTransform = vModel.getWorldTransform()|
Inverse = vTransform.inverse()
vMeshPosition  = vInverse * vWorldPosition
vModel.meshDeform.mesh[1].vertexList[10] = vMeshPosition
```

For more details about this, see "Transforms" on page 370.

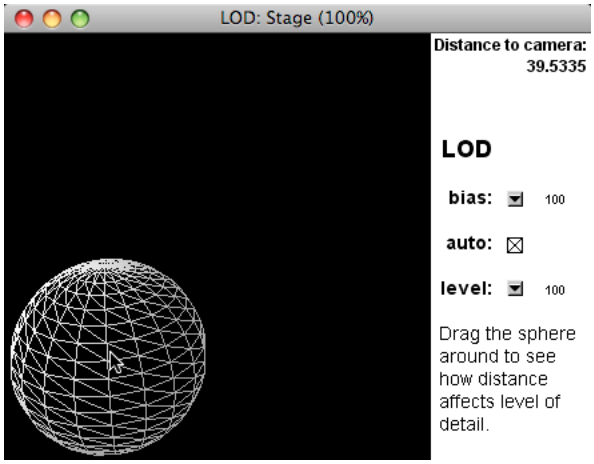
Level of detail (LOD) modifier

In the real world, as objects get further away, the less detail is visible. In a virtual 3D world, faces that are far from the camera appear tiny. Nevertheless, the computer processor must spend as much time calculating the color of the pixels for a distant face as for a face close to the camera. The solution is to simplify the geometry of distant models, so that they have fewer faces. All model resources created in third-party 3D design software contain data that allows the Shockwave 3D engine to reduce the level of detail on models far from the camera. You can set the `lod` properties of the imported model resource directly.

The modelResource of a model created in third-party 3D design software contains the following properties that modelResources created from Director primitives do not:

- modelResource.lod.auto
- modelResource.lod.bias
- modelResource.lod.level

These properties apply to all models to which the LOD modifier has not been added. The LOD modifier allows you to customize the level of detail shown on model on an individual basis. Even when you change the settings at the resource level, the LOD feature acts on the model, not on the resource. Two models that share the same resource can display different numbers of faces, depending on how close they are to the camera. The #lod modifier lets you give individual models their own lod settings, independently of the lod settings for the shared model resource. To see the #lod modifier in action, download the movie LOD.dir, and launch it.



The #lod modifier lets you set how aggressively to simplify model geometry as distance from the camera increases

The LOD.dir movie uses a wire frame shader. To make the effect obvious, select a low value for bias, and ensure auto is checked. Now drag the sphere around. You will see the value of level change as sphere moves closer to away from the camera.

Note: *Models created in Director from 3D primitives do not contain LOD data. There is no advantage to be gained from adding the LOD modifier to such models.*

Adding the LOD modifier to a model

The following command adds the #lod modifier to model 2, named "LoseDetail", of the member "3D".

```
-- Lingo syntax
member("3D").model("LoseDetail").addModifier(#lod)
put member("3D").model("LoseDetail").modifier
-- [#lod]
// JavaScript syntax
member("3D").getPropRef("model", 2).addModifier(symbol("lod"));
1
trace(member("3D").getPropRef("model", 2).modifier)
// <[#lod]>
```

Removing the modifier

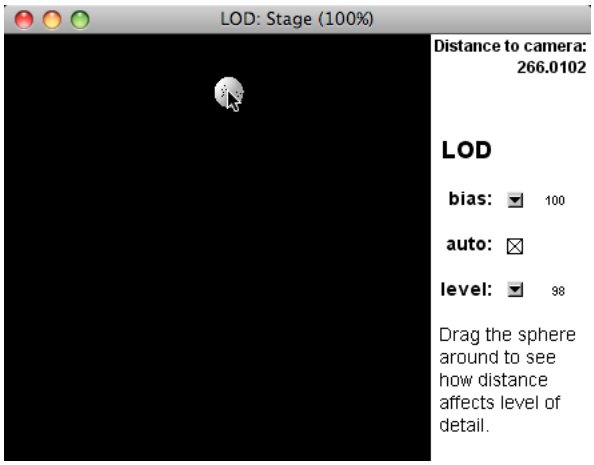
You can use the command aModel.removeModifier() to remove the #lod modifier.

```
-- Lingo syntax
member("3D").model("LoseDetail").removeModifier(#lod)
// JavaScript syntax
member("3D").getPropRef("model", 2).removeModifier(symbol("lod"));
1
```

LOD properties

The #lod modifier adds three properties to a model:

- model.lod.auto: If TRUE, the model obeys the `lod.bias` setting. If FALSE, it obeys the `lod.level` setting.
- model.lod.bias: Floating point value from 0.0 to +100.0. If set to 0.0, the LOD modifier removes all polygons, even when the model is immediately in front of the camera. If it is set to 100.0, it does not start to remove polygons until the model appears very small in the distance. The precise distance at which the bias setting starts to take effect depends on the camera's projection angle. The default setting is 100.0.
- model.lod.level indicates the amount of detail removed by the modifier when its auto property is set to FALSE. The range of this property is 0.0 to 100.00. When the modifier's auto property is set to TRUE, the value of the level property is dynamically updated. If you set the value manually, it will not have any effect.



With lod.bias at 100.0, the auto setting starts to take effect when the model appears small in the distance

See also targetFrameRate.

Subdivision surfaces (SDS) modifier

The #sds modifier adds geometric detail to models and synthesizes additional details to smooth out curves as the model moves closer to the camera. The #sds modifier directly affects the model resource. All models that use the same resource will be affected. To see the effect of the #sds modifier, download the movie SDS.dir, and launch it.

Note: *The #sds modifier and the #lod modifier behave in contrasting ways. Before adding the sds modifier, it is recommended that you set the* `lod.auto` *modifier property to FALSE and set the* `lod.level` *modifier property to the required resolution, as follows:*

```
vModel = member("3D").model("myModel")
vModel.lod.auto = 0
vModel.lod.level = 100
vModel.addmodifier(#sds)
```

The #sds modifier is best used on models that were created in a third-party 3D design application. If used with models created from Shockwave 3D primitives, the modifier affects the model resource, and all the models that share that resource.

Adding the SDS modifier to a model

The following command adds the #sds modifier to model 2, named "MakeMeSmooth" of the member "3D".

```
-- Lingo syntax
member("3D").model("MakeMeSmooth").addModifier(#sds)
put member("3D").model("MakeMeSmooth").modifier
-- [#sds]
// JavaScript syntax
member("3D").getPropRef("model", 2).addModifier(symbol("sds"));
trace(member("3D").getPropRef("model", 2).modifier)
// <[#sds]>
```

Note: *If you add the sds modifier to one model with a given modelResource, all other models that use the same modelResource will have access to the modifier too. Adding the sds modifier to a model in fact applies the sds feature to the model's resource, which may be shared with other models.*

Removing the modifier

You can use the command aModel.removeModifier() to remove the #sds modifier.

```
-- Lingo syntax
member("3D").model("MakeMeSmooth").removeModifier(#sds)
// JavaScript syntax
member("3D").getPropRef("model", 2).removeModifier(symbol("sds"));
1
```

SDS properties

The #sds modifier adds the following properties to all models that share the same modelResource:

- model.sds.enabled indicates whether subdivision surfaces functionality is enabled (TRUE) or disabled (FALSE). The default setting for this property is TRUE.
- model.sds.depth specifies the maximum number of levels of resolution that the model can display when using the #sds modifier.
- model.sds.error indicates the level of error tolerance for the subdivision surfaces functionality. This property applies only when the sds.subdivision property is set to #adaptive.
- model.sds.subdivision indicates the mode of operation of the subdivision surfaces modifier. Possible values are as follows:

  #uniform Specifies that the mesh is uniformly scaled up in detail, with each face subdivided the same number of times.

  #adaptive specifies that additional detail is added only when there are major face orientation changes and only to those areas of the mesh that are currently visible.

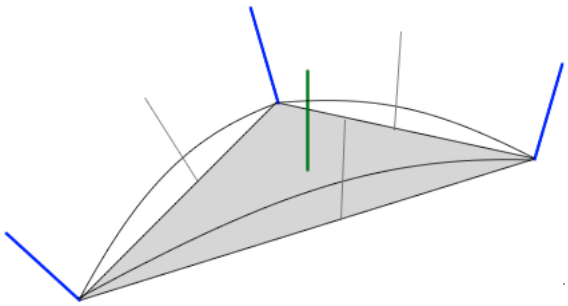
SDS and spot lights

If you are using spot lights with a narrow beam, then smaller faces give a better result. A face is lit correctly only if all three vertex points are within the light beam. If the faces are large, compared to the light beam, the beam may "hit" the center of a face, but not its vertices. In this case, the face does not light up at all. See "How faces are lit" on page 127 for more details.

You can use the #sds modifier to increase the density of the vertices in a model, so that it responds better to spot lights with a narrow beam.

Flat shading and smooth shading

When you use meshResource.generateNormals(), you can choose between #flat and #smooth. In a shader, you can set the #flat property to TRUE or FALSE. These properties determine how normals are treated over the surface of each face. Each face has one normal vector associate with each vertex. These normals may be pointing in different directions (blue in the illustration below). With flat shading, the average of the three normals is applied to even point on the face (green in the illustration below). This gives the impression of a flat triangle for each face.



Comparison of flat shading (gray triangle) and smooth shading (curved outlines)

With smooth shading, for each point on the surface of the face, a unique value is calculated for the normal, based on an interpolation from all three normals at the vertex points. This requires more processor time. Each face still has a triangular shape, but the shading varies smoothly across the surface to give the illusion than the surface is curved. For more details, see these links:

- http://en.wikipedia.org/wiki/Flat_shading
- http://en.wikipedia.org/wiki/Gouraud_shading
- http://en.wikipedia.org/wiki/Phong_shading

Particles

Particle systems are unique among model resources in that they include animation by default. Instead of being shapes, they create cascades of moving particles.

Particle systems, whose type is #particle, can have an infinite variety of appearances, simulating fire, smoke, running water, and other streaming or bursting effects.

Defining a particle system

To generate a unique particle effect, you can set up to 22 different properties. These include start and end size, blend and color of the particles, speed, and direction of initial movement, the effects of gravity and wind, and so on. See Particle system properties for the complete list of properties that you can use with a Particle system.

Also, view the sample movie, Particle.dir.

You cannot import particle resources from a third party 3D design application, so you have to build them directly in Director, using the #particle primitive. This means that, at least once, you have to create the Particle system manually. This often means using trial and error.

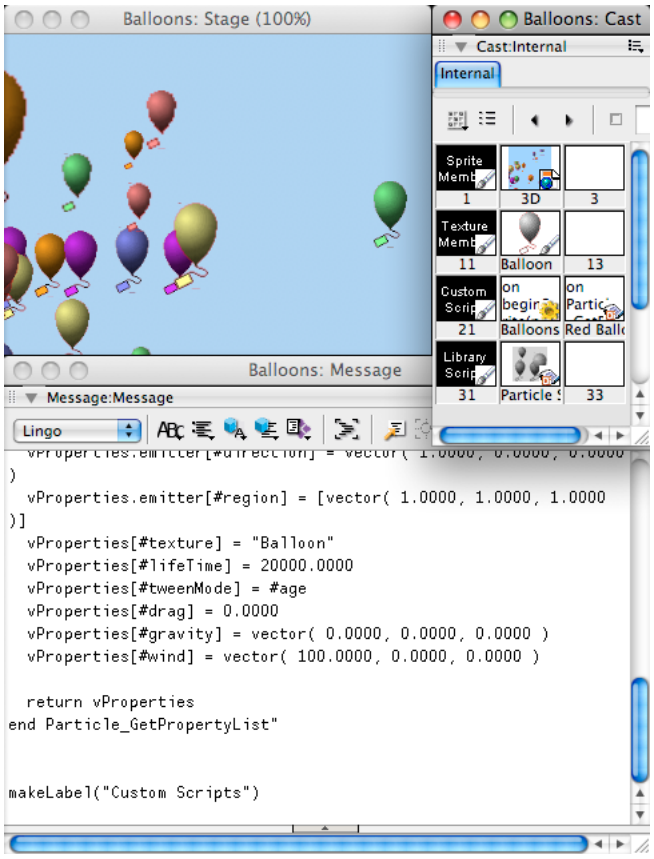
Saving a particle system

There are two ways you can save the Particle system that you have created:

- If you use the member3D.saveWorld() command, then save your movie, Director saves the current state of your world, including the state of any Particle system that you have created. See Saving the 3D world for more details.
- Saving the entire world may not be practical, especially if you want to compare slight variations in the Particle system that you are working on. The alternative is to save the current state of a Particle system as a property list, and then use that property list to regenerate the Particle system at a later time. The Particles.dir movie uses the second system to generate the different demonstration Particle systems. It includes a script named Particle Script, which has two main handlers:
  - Particle_GetScriptText(a3DMember, aParticleName)
  - Particle_SetProperties(a3DMember, aParticleName, aPropList). You can use the first to generate a string that you can paste into a Script member. This will create a new script. Calling the Particle_GetPropertyList() handler of this new script returns a property list that you can use with the Particle_SetProperties() handler of the Particle Script.

Using the Particle Script

To test this for yourself, you can follow the steps below.



Creating a new script that will regenerate a Particle modelResource

1 Make some changes to the properties of the Particle Test modelResource.

2 In the Message window, execute the following command:

```
new(#script).scriptText = Particle_GetScriptText(member("3D"), "Particle Test")
```

This will create a new Movie Script member in your movie.

3 Stop the movie and restart it. All the changes that you made to the Particle system will be lost.

4 In the Message window, execute the commands below:

```
vList = Particle_GetPropertyList() -- in your new
scriptParticle_SetProperties(member("3D"),"Particle Test",vList)
```

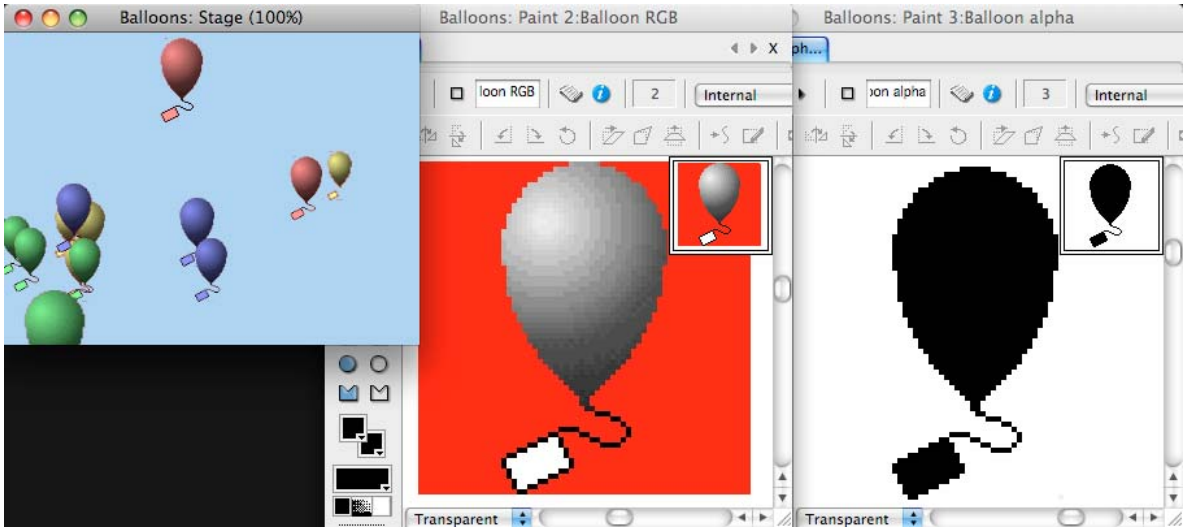
Initially, the new script will be created as a Movie Script. If you call Particle_GetPropertyList(), only the first movie script with that handler will respond. If you want to create a number of different Particle systems, you can give the scripts different names. The following are some commands to test in the Message window.

```
v3DMember = member("3D")
vName = "Explosion"
vPropList = script(vName).Particle_GetPropertyList()
vResource = Particle_SetProperties(v3DMember, vName, vPropList)
vModel = v3DMember.newModel(vName, vResource)
Using textured particles
```

These commands create a new particle modelResource, and a new model, using the data stored in the Explosion script.

In the Snow example in the Particles.dir movie, the snowflakes display a texture. Each particle is in fact a square plane. If you set the texture property of a #particle modelResource, that texture will be applied to the square plane. You can use a 32-bit image with transparency in the alpha to hide parts of the square. Since the snowflakes are round, a square texture mapped to a square plane works perfectly.

You can change the texture of a #particle modelResource on the fly. Immediately, all the particles will change their texture. To see this in action, show the Snow particles in the Particles.dir movie, then click on the 3D sprite. You will see the snowflakes spin. To see another example of textured particles, download and launch the movie Balloons.dir.



The balloon particles do not appear to be square, because of the alpha-channel transparency in the texture.

In this movie, too, the texture is mapped to a square plane. However, only a narrow part of the texture is opaque, so the balloon particles appear to be tall and thin. See "Images for textures" on page 144 for more details.

Multiple particle systems

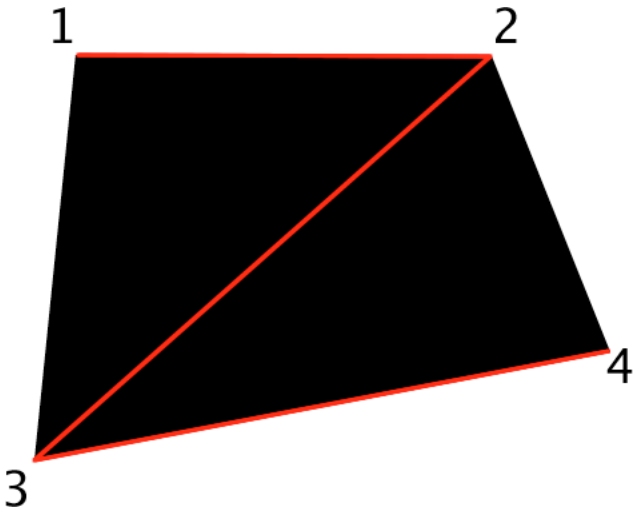
A single particle system gives you a single type of particles. In the Snow example, in the Particles.dir movie, all the snowflakes are the same shape. When you are mimicking a real-world phenomenon, such as fire, you will want to have a variety of colors and shapes in the particles. To do this, you can create multiple particle systems, each with slightly different properties. The Balloons.dir movie creates six particle emitters, all using the same texture, but with six different colors.

Emitter.region

By default, the particleResource.emitter.region property is set to a single vector position.This makes all particles appear at that point. You can set the emitter.region property to a line between two vector points, or to a quadrilateral between four vector points. For example, the emitter for the Snow particle can be set as follows:

```
vParticle = member("3D").modelResource("Particle Test")
vParticle.emitter.region = [vector(-75, -50, 0), vector(75, -50, 0), vector(-75, 50, 0),
vector(75, 50, 0)]
```

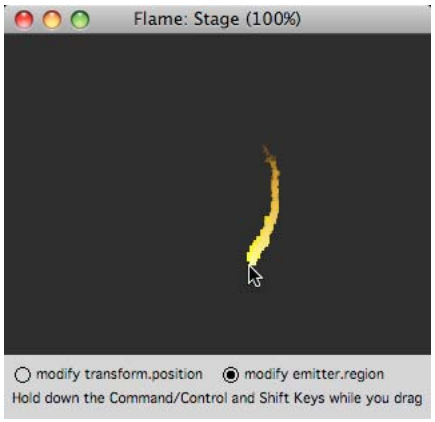
If you are used to working with 2D quads, you may expect the points in the emitter.region to follow each other in sequence around the four corners of the quadrilateral. In fact, they follow a Z pattern, as shown in the illustration below.



Defining the points of an emitter region

Moving the emitter.region

You may wish to move the Particle system around. For example, a trail of smoke from a car exhaust needs to follow the car around. The obvious solution is to attach the Particle emitter model to the car. However, if you do this, the particles that are emitted seem to be attached to the car in an unnatural fashion. A better solution is to place the model in a fixed location at the center of the world, and move its emitter.region around. To see an example of this, download and launch the movie Flame.dir.



Creating a realistic movement means moving the emitter.

Click on the 3D sprite and drag the flame around. Notice the difference between modifying the `transform.position` of the model and modifying the `emitter.region` of the resource. Pressing the Control/Command key updates the position of the emitter region inside a tight repeat loop. Pressing the Shift key will, in addition, adjust both the Particle emitters used by the flame to make the movement of the flame as realistic as possible.

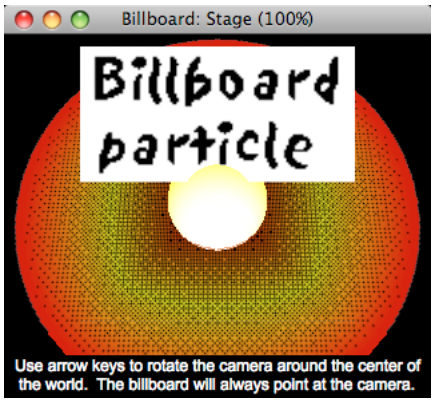
Scale

If you scale a model that displays a particle, the size of the particles themselves will not change. To scale the particles, you will need to modify the `sizeRange` properties, and also the various speed properties, such as `maxSpeed`, gravity, and wind. The `Particle_Scale()` handler in the Particle Script can do this for you. Note that scaling these properties affects every model that uses the Particle resource.

Billboards with a single particle

The planes that display the particles from a Particle system always turn to face the camera. You can use this to create billboards from a single static particle. You can use Particle systems like this to create signposts in an interplanetary adventure, or trees in the distance. To see this in action, download and launch the movie Billboard.dir.

Note: *If you used Particle system billboards for trees close to the camera, then the trees appear to lean towards the user as the camera approaches.*



Particles automatically turn to face the camera; a single static particle provides a 3D billboard

Chapter 4: 3D: Controlling action

For many of your projects, almost all the visible 3D assets will be created in third-party 3D design applications. Your job as a Director developer will often be to take those static assets and create an interactive experience from them.

The Lingo or JavaScript code that you write does more than simply control 3D objects. It brings a world to life. You are not just writing code, you are telling a story.

End-users do not care how hard it is to make a particular interaction appear simple. They want to forget about the mechanics of interacting with the 3D world, and immerse themselves in the experience. If an end-user becomes aware of your work as a programmer, it will probably be because you have done something wrong or unexpected.

This section provides you with the techniques that you will need to make your work invisible.

Arranging objects in a 3D world

In a 2D world, you can define an absolute position for each sprite, by measuring from the top left corner of the Stage. You can also rotate and scale sprites around their regPoint.

In a 3D world, the situation is similar. You can define the absolute position, rotation, and scale of a 3D node relative to the axes of the `group("World")`. Every node has a worldPosition property, which defines its position relative to the center of the `group("World")`. For each node, you can use getWorldTransform() to determine its position, rotation and scale relative to the `group("World")`.

One major difference between a 2D and a 3D world is that the viewing position is fixed in a 2D world. In a 3D world, everything that is drawn on the screen is shown from the viewpoint of the camera. The camera can move through 3D space, rotate around objects, and look at the world from any angle. In many cases, *absolute* information about a node is much less useful than *relative* information. Often you want to know the position, rotation, and scale of one node relative to another node.

Shockwave provides you with many ways to express positions, rotations, scales, and movements in 3D space. To work efficiently with these capabilities, you need to be familiar with the following ideas:

- Frames of reference. See “Frames of reference” on page 202.
- Transforms and their properties. See “Transforms” on page 204.
- How to set the position of a node. See “Setting a node's position” on page 205.
- How to move a node relative to a frame of reference. See “Translation” on page 206.
- How the scale and orientation of the frame of reference affects movements in a straight line. See “Translation” on page 206.
- How the origin point of the frame of reference affects rotations. See “Setting the rotation of a node” on page 208.

Frames of reference

In 3D, everything is drawn relative to the camera’s frame of reference. If the camera is behind an object, when the object moves to the left relative to the center of the world (the *world origin*), it appears to move toward the right of the screen.

Each piece of position and orientation information can be expressed relative to one or more *frames of reference*. A model's transform property, for instance, expresses its position and rotation relative to the model's parent.

When you use code to move a model, you must define the frame of reference for the movement. You define a frame of reference by referring to a node. You can use the symbols `#self`, `#parent` and `#world` as shortcuts for specific nodes (see the table below). You can also refer directly to a particular node.

`#self` or no reference	refers to the node that you are moving
`#parent`	refers to the parent of the node that you are moving
`#world`	refers to the `group("World")`

In other words, there are four frames of reference to consider:

- relative to the object (model, group, light, camera) itself (`#self`)
- relative to the object's parent (`#parent`)
- relative to the world (`#world`)
- relative to some other object (referred to explicitly).

Here are examples of movements made within each of these frames of reference.

Object-relative

When you create a model in a 3D-modeling program, you build it relative to its own frame of reference. For instance, when you create a model of a car, the front of the car may be pointed along its *z*-axis and the antenna may be pointed along its *y*-axis. To move such a car forward (along its *z*-axis) regardless of which direction it is pointing relative to the camera or the world, use `car.translate(0,0,10)`. To turn the car left, use `car.rotate(0,45,0)`.

The car model may have wheel models as children. To rotate the wheel of a car relative to itself, rather than relative to its parent (the car), use the following script:

```
wheel.rotate(0,10,0)
```

or

```
car.child[1].rotate(0,10,0, #self)
```

where the fourth parameter of the rotate method is the object, the rotation is relative to.

Parent-relative

Parent-relative: A model's transform property expresses its position and rotation relative to the model's parent. If you want the wheels of the car to move outward regardless of how the wheels are turned, use `car.child[1].translate(10,0,0,#parent)` or `car.child[1].transform.translate(10,0,0)`. If you want a planet model that is a child of the sun to orbit around the sun, use `planet.rotate(0,5,0, #parent)`.

World-relative

World-relative: If you want the car to move along the world's *x*-axis regardless of which way it is facing, use `model.translate(10,0,0,#world)`. If you want to rotate the car 20° around the world *y*-axis, with the rotation taking place at the world location vector (10, 10, 10), use `model.rotate(vector(10,10,10), vector(0,1,0), 20, #world)`.

Relative to another object

Relative to another object: If you want to move an object so that it goes toward the right edge of the screen, use `model.translate(vector(10,0,0), sprite(1).camera)`. If you want to rotate the object parallel to the camera and around the center of the screen, use `model.rotate(vector(0,0,0), vector(0,0,1), 20, sprite(1).camera)`.

Transforms

2D sprites have a number of properties that define where and how they appear on the Stage. For example, the `loc`, `rotation` and `skew` properties are all properties of the sprite itself.

In a 3D world, nodes have a worldPosition property, which is the 3D equivalent of the 2D `sprite.loc` property. For more details, see "Setting a node's position" on page 205.

Transform properties

All the other properties that determine the position and orientation of a node belong to the node's *transform*, and not to the node directly. You can find a complete table of transform properties at Transform Properties.

A transform has four properties that you can set:

Property name	Format
`position`	position vector
`scale`	vector
`rotation`	vector
`axisAngle`	[directionVector, angleOfRotationInDegrees]

Changing the values of any of the properties of the transform of a node will affect the node and all its children. If you change the `transform.position` property of a node, the node and all its children will move in 3D space. If you change the `transform.scale` property of a node, the size and relative spacing of the node and all its children will change proportionately. If you change either the `transform.rotation` or the `transform.axisAngle` property of a node, the node and all its children will rotate in 3D space. For more information on these two properties see "Setting the rotation of a node" on page 208.

The frame of reference for a transform

A transform defines the position, scale and rotation of a node in relative to the node's parent. Often the parent of a node will be the `group("World")`. If this is the case, then the `node.transform.position` will be identical to the `node.worldPosition`.

Get-only properties

A transform has three properties whose value you can get, but which you cannot set directly. These are:

- `xAxis`
- `yAxis`
- `zAxis`

These properties are all unit-length direction vectors, and they are all perpendicular to each other, according to the right-hand rule (see "3D space" on page 22). Their values will change when the `rotation` or `axisAngle` properties are changed.

Setting a node's position

You can change the position of a node in the following ways:

- Set the `node.worldPosition` property to a position vector representing an absolute position in world space
- Set the `node.transform.position` property to a position in the node's parent's frame of reference
- Use `node.translate()` to move the node a given distance in a given direction from its current location, in terms of a chosen frame of reference
- Use `node.transform.translate()` to move the node a given distance in a given direction from its current location, in terms the node's parent's frame of reference
- Modify the transform of one of the node's parents and the node will automatically move with its parent. If you change the parent's `.scale`, then the visible dimensions of the child node will also change.
- Change the node's parent while preserving the node's transform

You can find more details on using the `translate()` command for nodes and transforms at "Translation" on page 206. For details on parent-child relationships, see "Node hierarchy" on page 91.

This section explains how to set the `worldPosition` and `transform.position` properties directly.

WorldPosition

You can move a node to an absolute position in the 3D world space by setting its `.worldPosition` property. For example, this command moves the model "car" of 3D member "world" to the absolute position in the 3D world space:

```
member("world").model("car").worldPosition = vector(31, 287, -7)
```

GetWorldTransform()

You can use the command getWorldTransform() to obtain a transform whose properties describe the position, scale and rotation of the node in terms of the 3D world itself. If you alter any of the properties of the transform you obtain, you will not affect the node in any way: the transform returned by `getWorldTransform()` is not attached to any node.

The value of `node.getWorldTransform().position` will be identical to `node.worldPosition`. You can set the worldPosition property, using a position in world space, but there is no shortcut for setting `scale` or `rotation` in the frame of reference of world space.

Node.transform.position

Setting the value of `node.transform.position` will place the node in a position relative to its parent's frame of reference.

```
member("world").model("car").transform.position = vector(31, 287, -7)
```

Multiplying by a transform to convert between frames of reference

You can convert a `node.transform.position` to a `node.worldPosition` by multiplying it by the transform obtained using `node.parent.getWorldTransform()`.

In this code example, the `model("Blue")` is made the child of the `group("Frame of Reference")`. The model's `worldPosition` and its `transform.position` are different because the `group("Frame of Reference")` is not placed at the center of the world.

```
vGroup = member("3D").group("Frame of Reference")
vModel = member("3D").model("Blue")
put vModel.worldPosition
-- vector( -27.2652, -72.2760, -86.8480 )
vGroup.addChild(vModel, #preserveWorld)
put vModel.transform.position
-- vector( 44.7348, -79.2760, -25.8480 )
```

To convert the model's `transform.position` into a position within the frame of reference of the world, you can multiply it by the transform of its parent:

```
vTransform = vModel.parent.getWorldTransform()
put vTransform * vModel.transform.position
-- vector( -27.2652, -72.2760, -86.8480 )
```

To convert from world space to the frame of reference of the model's parent, you can use the `inverse()` of the transform returned by `getWorldTransform()`.

```
vInverse = vTransform.inverse()
put vInverse * vModel.worldPosition
-- vector( 44.7348, -79.2760, -25.8480 )
```

For more details on transform mathematics, see "Transforms methods" on page 372.

Translation

To move a node a certain distance in a particular direction, you can use the translate command. This accepts two types of information:

1 An indication of the direction and distance of the movement.

2 An indication of the frame of reference for the movement (defined by a node or a symbol representing a node).

You can express the direction and distance of the movement in one of two ways:

- As three separate floating point numbers, representing the distance to move along the x-, y- and z-axes.
- A vector.

The indication of the frame of reference is optional. If you omit it, the frame of reference is considered to be that of the node that you are moving.

```
node.translate(xIncrement, yIncrement, zIncrement {, relativeTo})
node.translate(translateVector {, relativeTo})
```

Note: *You can also apply the `translate()` command to the transform of a node. However, a transform has no parent, so the translation can only be applied within its own frame of reference. Providing a frame of reference parameter when using `translate()` with a transform will provoke a script error.*

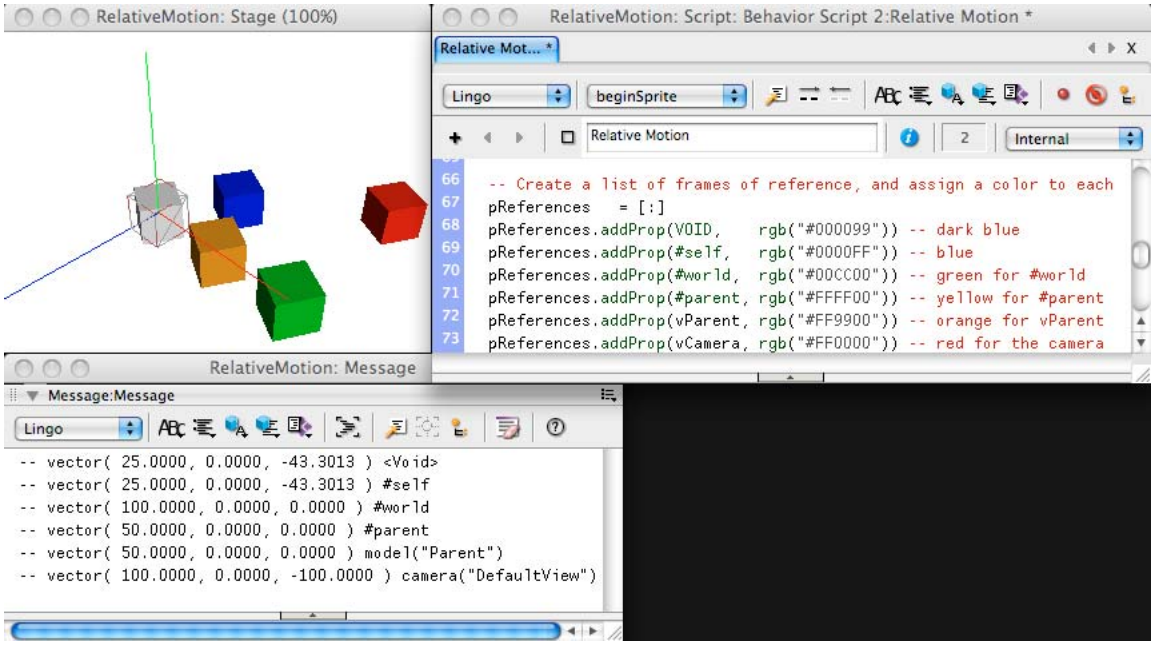
```
aTransform.translate(xIncrement, yIncrement, zIncrement)
aTransform.translate(translateVector)
```

Orientation and scale of the frame of reference

The distance and direction of a translation is affected by the orientation and scale of the node that acts as the frame of reference.

To see the effect of the orientation and scale of the frame of reference, download the RelativeMotion.dir and launch it. This movie creates two box models: a Parent model (which appears as a wire frame) and a Child model. It executes the same `translate()` action 6 times, using 6 different terms to define different frames of reference.

Each time the `translate()` action is executed, the color of the Child model is changed. The image below shows where the Child model ends up after each `translate()` action.



Changing the node that acts as the frame of reference affects the distance and direction of a translate() action.

Note the following points:

- The direction of the translation depends on the orientation of the reference node. In the example, the Child model's parent is aligned with the `group("World")`, but the camera is rotated 45° around the y-axis and Child model itself is rotated 60° around the y-axis.

  As a result, the translations illustrated in orange (relative to the parent) and green (relative to the world) both occur along the world's x-axis (the red line), but the other translations occur at different angles.

- The distance traveled depends on the scale of the node that acts as the frame of reference. The Parent model is scaled by a factor of 0.5, and its scale also affects the Child model. The `group("World")` is scaled by a factor of 1.0, and the camera is scaled by a factor of `sqrt(2.0)`.

  As a result, the translations shown in orange (relative to the parent) and blue (relative to the Child model itself) are half the length of the translation relative to the world (shown in green). The translation relative to the camera (shown in red) is about 1.4 times longer than the translation relative to the world.

- Using a symbol shortcut, such as `#self` or `#parent`, has exactly the same effect as using an explicit reference to the associated node. If you do not include any reference to a node, then the node that you are moving is chosen by default as the frame of reference.

  As you watch the movie, the positions of the dark blue and light blue boxes (using no reference and `#self`) are identical, and the positions of the yellow and orange boxes (using `#parent` and `model("Parent")`) are also identical.

Note: *This example uses a camera that has been scaled by a factor other than 1.0. This is for the purposes of this demonstration only. Generally, changing the scale of a camera will not be necessary.*

Setting the rotation of a node

A transform's `rotation` and `axisAngle` properties contain the same information in two different formats. If you change the value of one of these properties, the other will update automatically. For example:

```
vTransform = member("3D").model(1).transform
put vTransform.rotation
 -- vector( 164.0000, 29.0000, 71.0000 )
put vTransform.axisAngle
 -- [vector( -0.7860, -0.6049, 0.1278 ), -150.6098]
```

If you change the axisAngle, the rotation will update.

```
vTransform.axisAngle = [randomVector(), random(3600) / 10.0]
put vTransform.axisAngle
 -- [vector( -0.9109, -0.0399, -0.4107 ), -23.0000]
put vTransform.rotation
 -- vector( 20.9315, -0.8107, 9.4034 )
```

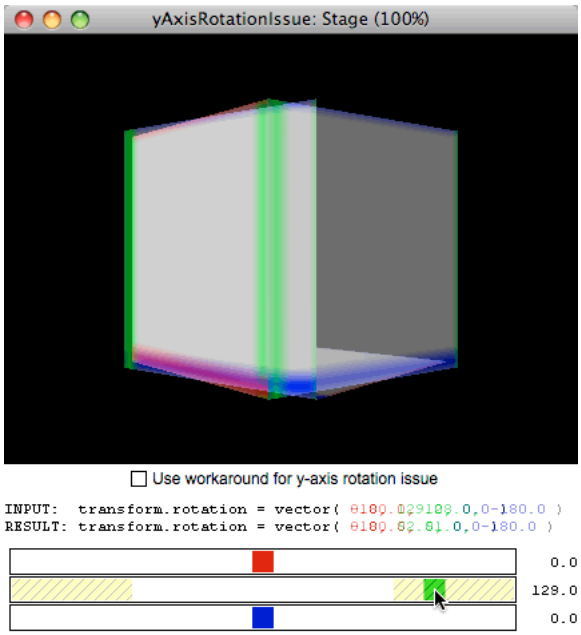
If you change the rotation, the axisAngle will update.

```
vTransform.rotation = vector(21, -1, 9)
put vTransform.axisAngle
 -- [vector( -0.9177, -0.0289, -0.3962 ), -22.9211]
```

Note: *In Director 11.5 and earlier, getting and then setting the* `.rotation` *property of a transform can lead to unexpected results. If the absolute value of the* `y` *component of the rotation is between 90.0 and 270.0, the transform may end up facing in a different direction from what you expect.*

The workaround is to store a duplicate of the `transform.rotation` *vector, and to use the duplicate rather than the current value of* `transform.rotation` *vector to set any new values.*

To see this issue and its resolution, download the yAxisRotationIssue.dir and launch it.



Unless you use a workaround, rotations where the y component is between 90.0° and 270.0° behave erratically

As you drag the sliders, the 3D model will rotate.

If you drag the green y-component slider inside the zone with diagonal yellow warning stripes, the model will oscillate unexpectedly. However, if you check the Use Workaround For Y-Axis Rotation Issue button, a code similar to the following will be used:

```
property pBox      -- pointer to the "Box" model
property pRotation -- duplicate of rotation vector
on beginSprite(me)
  v3DMember = sprite(me.spriteNum).member
  v3DMember.resetWorld()

  vResource = v3DMember.newModelResource("Box", #box)
  pBox      = v3DMember.newModel("Box", vResource)
  -- WORKAROUND: Use a duplicate rotation vector
  pRotation = pBox.transform.rotation
end beginSprite
on SetRotation(me, aAxis, aAngle)
  -- WORKAROUND: Alter pRotation and not pBox.transform.rotation
  case aAxis of
    #x:
      pRotation.x = aAngle
    #y:
      pRotation.y = aAngle
    #z:
      pRotation.z = aAngle
  end case

  pBox.transform.rotation = pRotation
end SetRotation
```

This workaround will continue to work correctly in releases of Director subsequent to 11.5, where this issue has been dealt with.

Note: *In a* `transform.axisAngle` *where the angle is set to zero, the direction of the axis becomes arbitrary. If there is no rotation, then it doesn't matter around which axis the non-rotation occurs.*

If you set the angle of a `transform.axisAngle` *to zero, then Director will reset the axis to* `vector( 1.0, 0.0, 0.0 )` *by default.*

```
vTransform.axisAngle = [randomVector(), random(3600) / 10.0]
put vTransform.axisAngle
 -- [vector( 0.0282, -0.6930, 0.7204 ), -163.1000]
vTransform.axisAngle[2] = 0
put vTransform.axisAngle
 -- [vector( 1.0000, 0.0000, 0.0000 ), 0.0000]
```

Rotate()

To rotate an object, you can use the `rotate()` command. Provide the following information with this command:

- A center of rotation. The object will maintain the same distance from this point as it rotates.
- An axis of rotation. The object will move in a plane perpendicular to this axis.
- An angle through which the object will turn.
- A frame of reference. (Optional) If this is omitted, the frame of reference is considered to be that of the node that you are moving.

Imagine that you want to tip a chair backwards. The center of rotation can be any point on the line between its two back feet. The axis of rotation would be parallel to the line between its two back feet, and the angle can be anything up to about 90°, after which its back will hit the floor. The frame of reference would be the chair itself, so you can use `#self`, or leave the frame of reference blank.

Imagine that you want to swing the chair around you in a circle, in preparation for throwing it through a window. In this case, the centre of rotation would be your center of gravity, the axis of rotation would be vertical, and the angle would be something like 360°. You can consider yourself to be the temporary "parent" of the chair, as its movement is defined by yours, so you can use `#parent` or `model("me")` as the frame of reference.

You can provide these four items of information in a number of different ways.

1. `aNode.rotate(position, axis, angle {, relativeTo})`
2. `aNode.rotate(xAngle, yAngle, zAngle {, relativeTo})`
3. `aNode.rotate(rotationVector {, relativeTo})`

Note: *You can also apply the `rotate()` command to the transform of a node. However, a transform has no parent, so the rotation can only be applied within its own frame of reference. Providing a frame of reference parameter when using `rotate()` with a transform will provoke a script error.*

```
transform.rotate(position, axis, angle)
transform.rotate(xAngle, yAngle, zAngle)
transform.rotate(rotationVector)
```

Three rotation angles or only one rotation angle? Read the following sections to know more.

Three rotation angles

If you use a rotation vector or three separate rotation angles, then the center of rotation and axis of rotation are defined implicitly. Consider this example:

```
chairModel.rotate(17, 23, 41, irateEmployeeModel)
```

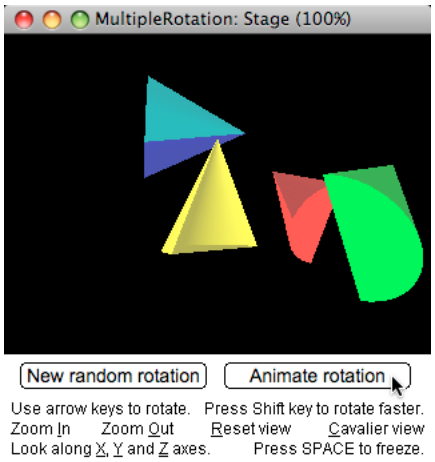
This means:

- Rotate `chairModel` 17° around the origin point of `irateEmployeeModel` in a plane perpendicular to `irateEmployeeModel`'s x-axis
- Then rotate `chairModel` 23° around the origin point of `irateEmployeeModel` in a plane perpendicular to `irateEmployeeModel`'s y-axis
- Finally rotate `chairModel` 41° around the origin point of `irateEmployeeModel` in a plane perpendicular to `irateEmployeeModel`'s z-axis

The end result of any sequence of rotations can be produced by one single rotation. The three separate rotations above are equivalent to a single rotation of about 47° around the origin point of the `irateEmployeeModel` on a plane perpendicular to an axis parallel to the `vector(-0.1681, -0.5940, -0.7867)` in the `irateEmployeeModel`'s frame of reference.

Tip: *You can use axisAngle to discover the orientation of the axis and the angle of rotation of a node, in its own frame of reference, after a rotation.*

To see a demonstration of this concept, download RandomRotation.dir and launch it.



The yellow half-cone shows the initial position, the blue half-cone shows the final position

The RandomRotation.dir movie creates a yellow half-cone model and places it at the center of the world. Next, it creates an invisible group named "Frame of Reference" and places it at a random position. Then it creates a random rotation in the form of a vector:

```
vector(randomXAngle, randomYAngle, randomZAngle)
```

The red, green and blue half-cones show the positions of the yellow model after its rotation around the x-axis, the y-axis and z-axis respectively. These models illustrate the intermediate steps that would be performed by the command:

```
yellowModel.rotate(xAngle, yAngle, zAngle, frameOfReference)
```

The blue model shows the final position after the rotation is complete.

Only one rotation angle

You can move the yellow model to occupy the same position as the blue model using a single rotation around a single axis. You can use `blueModel.transform.axisAngle` to find out which axis and angle to use. For example:

```
vAxisAngle = blueModel.transform.axisAngle
put vAxisAngle
-- [vector( -0.1681, -0.5940, -0.7867 ), -46.6984]
vAxis = vAxisAngle[1]
vAngle = vAxisAngle[2]
```

The cyan cone half-cone represents the position of the missing half of the yellow cone, after it has been rotated using the following command:

```
cyanModel.rotate(vector(0,0,0), vAxis, vAngle, frameOfReference)
```

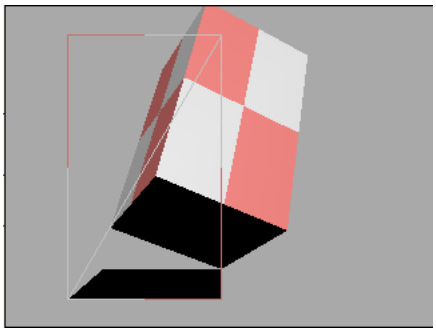
The cyan model and the blue model together now form a single cone. This demonstration shows that the two rotation commands lead to identical results.

You can click on the Animate Rotation button to see each step of these two commands played out, one after the other.

Using a group to help define a rotation

Groups are invisible. You can attach a group to any point of a model, as a child of the model, and then take advantage of the group's transform properties to track how that point on the model moves through space.

To see an example of this in action, download the RotationDemo.dir and launch it.



The RotationDemo.dir movie demonstrates rotation around three different axes, using a group to help

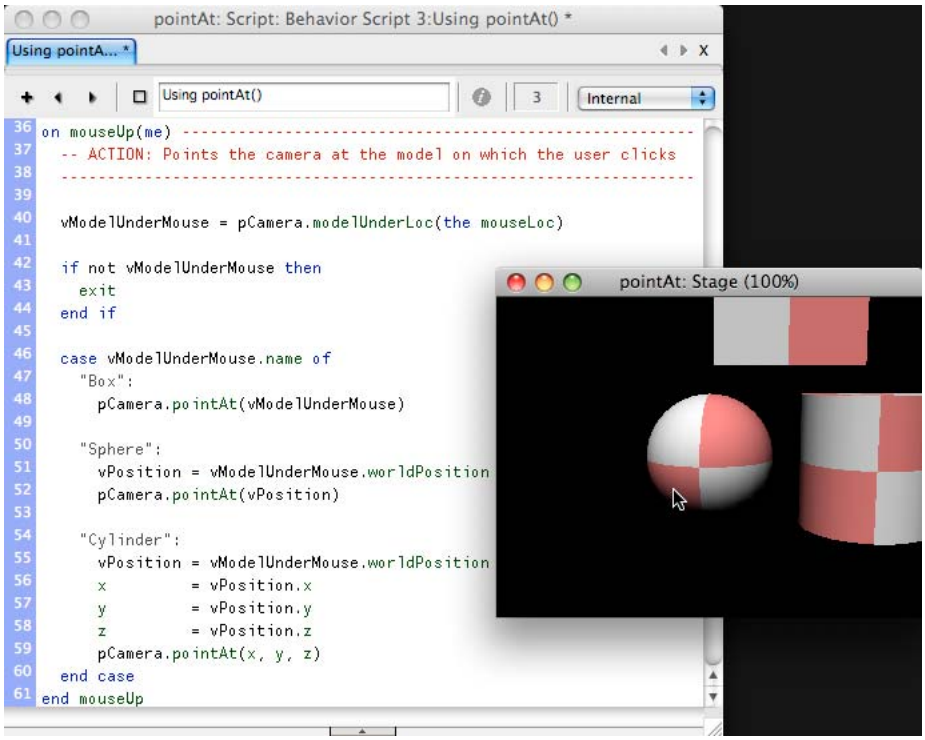
A group is attached to the rear right corner of the box. To tip the box backwards, it is rotated around the group's `transform.zAxis`. Because the group is a child of the box, its position relative to the box remains constant as the box rotates. The group's `transform.xAxis` is used next, as the axis for a new rotation. Finally, the box is spun around a vertical axis passing through the group's `worldPosition`.

Using pointAt() to rotate a node

To make a node rotate to face at a given point in space, you can use the pointAt command. Define a target using any of the following techniques:

- `node.pointAt(anotherNode)`
- `node.pointAt(worldPositionVector)`
- `node.pointAt(x, y, z)`

To see a demonstration of all three techniques, download and launch the movie pointAt.dir.



In pointAt.dir, a click on one of the models will turn the camera to point at that model

Note: *`pointAt()` is similar to `worldPosition` in that it functions in world space. Unlike `translate()` or `rotate()`, you cannot provide a frame of reference as a parameter to the `pointAt()` command.*

You can use `pointAt()` with a node (model, group, camera, light), but you cannot use it with a transform. You will encounter a script error if you attempt to it as a transform method.

Defining an upwards direction

Point at something with your index finger. Now stretch back your thumb so that it is at right-angles to your index finger. You can rotate your wrist so that your thumb is pointing in almost any direction, while your index finger continues to point at your target.

Imagine that your index finger is the `zAxis` of a node's transform, and that your thumb is the `yAxis`. By default, the `pointAt()` command will use the node's `zAxis` to point at the target position, and will place the `yAxis` in a plane that also passes through the world's y-axis. If you point vertically up or down, the `node.transform.yAxis` will be perpendicular to the `yAxis` of the world, so there is an infinite number of directions in which the node's `yAxis` can point and still be in the same plane as the world's `yAxis`. If this happens, Director will choose to align with the world's `xAxis` instead. When the node becomes absolutely vertical, it may suddenly twist through 180°.

Note: *This issue in known as Gimbal Lock. In the Apollo 11 space mission, which put the first men on the moon, the astronauts had to override the on-board computer when the Lunar Module was pointing directly upwards for the landing. Fortunately, the computer that you are using to run Director is much more powerful than theirs.*

You can avoid this issue by using an additional parameter for the `pointAt()` command:

- `node.pointAt(anotherNode, vectorUp)`
- `node.pointAt(worldPositionVector, vectorUp)`

- `node.pointAt(x, y, z, vectorUp)`

If this `vectorUp` parameter is not used, then the world's `yAxis`, `vector(0, 1, 0)`, is used by default.

Note: *You cannot get a node to point at itself, or to the worldPosition where it is currently located. If you do, an "Invalid Point" script error will occur. Always ensure that the target of a `pointAt()` command is a distinct point elsewhere in world space.*

You can also provoke this "Invalid Point" script error by using the `pointAt()` method with a `vectorUp` parameter that is parallel to the direction in which the node is to point.

The vectorUp.dir movie (see "Example" on page 214 below) provides solutions to both these issues.

Defining a node's pointAt direction

By default, a node will turn its `zAxis` to point in the direction of the chosen target. You can use pointAtOrientation to define a different direction. You can also use this property to define which direction must point upwards when `pointAt()` is used. Here is what the default value of `pointAtOrientation` looks like:

```
put member("3D").model(1).pointAtOrientation
-- [vector(0.0, 0.0, 1.0), vector(0.0, 1.0, 0.0)]
```

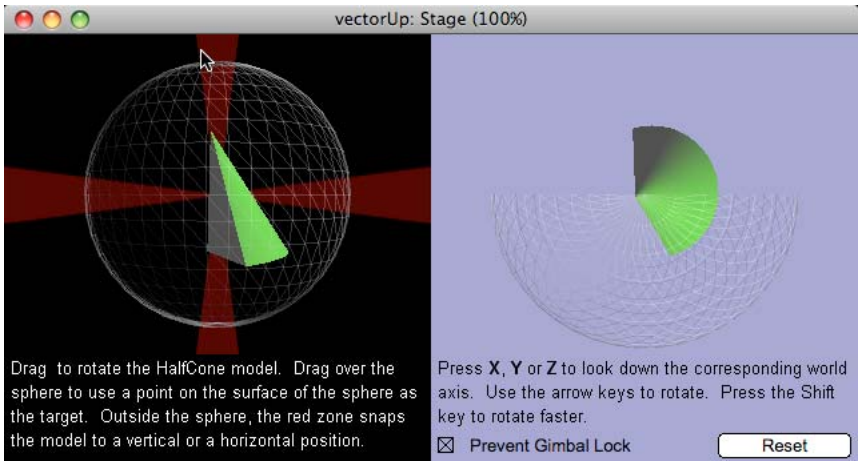
The first entry in the list indicates the direction that is used for pointing at the target (the node's `zAxis` by default). The second entry indicates which direction will be used to align with the `vectorUp` direction.

Suppose you create a half-cone using the Cone primitive, and that you want to point its tip at a target and keep the curved side facing upwards. The tip of the half-cone is along its `yAxis`, and the center of its curved surface may lie on the model's `xAxis`. Here is code that will make such a model, and which will set its `pointAtOrientation` appropriately:

```
v3Dmember = member("3D")
  vResource = v3DMember.newModelResource("HalfCone", #cylinder)
  vResource.topRadius  = 0
  vResource.startAngle = 180
  vModel = v3DMember.newModel("HalfCone", vResource)
  vList = []
  vList.append(vector(0, 1, 0)) -- yAxis
  vList.append(vector(1, 0, 0)) -- xAxis
  vModel.pointAtOrientation = vList
```

Example

Download the vectorUp.dir and launch it.



The vectorUp.dir movie demonstrates a solution to all the issues mentioned above

The vectorUp.dir movie uses a Movie Script called Point Node At as a wrapper for the `pointAt()` method. The global `PointNodeCarefullyAt()` handler ensures that no script errors occur, and that the rotation of the node does not suddenly switch when the node is asked to point vertically up or down.

However, one issue that requires a solution is Gimbal Lock. To see this issue, drag the mouse from left to right either above or below the sphere. The model will turn to point at the mouse. As the mouse passes through the vertical, the model will suddenly rotate 180°.

Note: *Note the red zone above and below the sphere. The demo is programmed to make the HalfCone model snap to the vertical when the mouse is in this area, if the Prevent Gimbal Lock button is not checked. This is to make it easier for you to make the model jump to a vertical orientation at any time. This "snap to vertical" behavior is not a feature of the Point Node At script.*

To avoid this unexpected, sudden rotation, check the Prevent Gimbal Lock button. When this button is checked, the behavior on the 3D sprite will prevent the HalfCone model from ever becoming absolutely vertical. Instead, it will make the pointAt target move in a small circle around the vertical axis.

This feature is not included in the Point Node At script, because your needs may vary from project to project and the Point Node At script is generic.

Here are the lines of code in the behavior on the 3D sprite that prevent the Gimbal Lock from occurring:

```
vLoc        = the mouseLoc - sprite(1).left, [sprite(1).top]
  vWorldSpace = sprite(1).camera.spriteSpaceToWorldSpace(vLoc)
  if pPreventGimbalLock then
    -- Rotate in a small circle around the absolute vertical
    vXSquared = vWorldSpace.x * vWorldSpace.x
    vMinimum  = 144.0 -- <HARD-CODED>
    if vXSquared < vMinimum then
      vWorldSpace.z = sqrt(vMinimum - vXSquared)
    end if
  end if
```

Note that this code takes a shortcut that relies on the knowledge that the HalfCone model is centered at `vector(0, 0, 0)`. In your own projects, you may need to make adjustments for the `x` and `z` components, depending on the `worldPosition` of the model that you are working with.

Moving the camera

A 3D world with no movement is just a static image. Movement brings a 3D world alive. As far as movement is concerned, there are four main elements in a 3D world:

- The sky or other background, which remains at a fixed distance from the camera
- The terrain and other objects that remain in fixed positions relative to the world
- Moveable objects, such as characters and vehicles
- The camera itself

The camera represents the user's eyes. A user who feels comfortable with the movements of your camera will become immersed in the world that you have created. If the camera moves in unexpected ways or is difficult to control, the user will feel alienated and may quickly lose interest in your creation.

This article helps you deal with the camera movements for your projects.

Camera control

Consider the following points while controlling the movement of the camera:

Choosing the appropriate movement for the 3D scenes

Each type of scene requires a different technique for controlling movement of the camera. Here are some examples:

- To present a simple object, you may want to have the camera rotate around the object so that it can be seen from any angle. See "Rotating around an object" on page 218.
- For a visit to a virtual art gallery, you may want to arrange a series of preset views of the key exhibits. See "Preset Views" on page 218.
- To present the architecture of a virtual building, you may want to provide a pre-defined path that the visitor can follow. See "Following a pre-defined path" on page 220.
- In a first-person action game, you may want the camera to be the eyes of the player. See "Steering with the mouse" on page 221.
- For a role-playing game, you may want the camera to follow the player's character. See "Third-person camera" on page 230 and "Making the camera move naturally" on page 236.

Giving user the control of the movement

Most users will have a mouse and a keyboard that they can use to control the movement of the camera. Some users may have a joystick or a steering-wheel input. Therefore, you have to choose the most appropriate input control system for the world that you have created.

A very simple camera movement may not provide any user control. The camera simply follows a pre-defined path through the space. In this case, the user witnesses the world as a passenger. At the other extreme, the user may play the pilot of a virtual helicopter or a stunt aircraft that is capable of moving in all three dimensions of space, and performing loops, yaws and barrel rolls.

Therefore, there is no one-size-fits-all solution for controlling camera movements. Each project may require you to design a different customized solution. Here are some examples:

Keyboard control of the camera

- Rotation around an object. See "Rotating around an object" on page 218.
- Forward movement and steering. See "Steering with the mouse" on page 221.
- Customization of camera control keys. See "Customizing control keys" on page 259.

Moving the viewpoint through space

- Following a path. See "Following a pre-defined path" on page 220.
- Warping to a new location. See "Preset Views" on page 218 and "MiniMap" on page 233.
- Moving in a straight line from the current location to a new location. See "Moving to a given location" on page 225
- Finding a path from the current position to a different position. See "Finding a path" on page 229.

Mouse control of the camera

- Select a view point by clicking on its name in a list. See "Preset Views" on page 218.
- Click on an object in the scene to jump to the best position for viewing that object. See "Preset Views" on page 218 and "Picking" on page 242.
- Steer by pointing the mouse in the direction you want to turn. See "Steering with the mouse" on page 221.
- Use the mouse to look around from the current camera viewpoint. See "Looking around" on page 222.
- Click on the scene to define where the camera is supposed to move to. See "Moving to a given location" on page 225.
- Click on a location in a mini-map to jump to that spot. See "MiniMap" on page 233

Looking around from the current viewpoint

- Use the mouse to look around from the current camera viewpoint. See "Looking around" on page 222.
- Rotate around the object so that it can be seen from any angle. See "Rotating around an object" on page 218.

Controlling interactions between the camera and the world

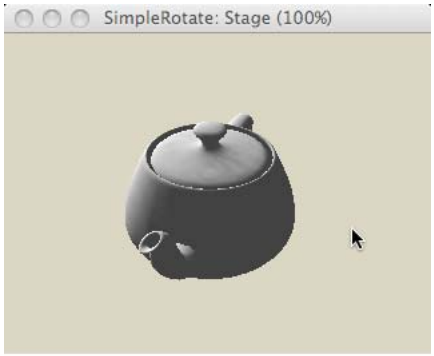
- Creating an elastic distance between the player's avatar and the camera following it, based on the avatar's speed. See "Making the camera move naturally" on page 236.
- Avoiding collisions between the camera and the scenery, and avoiding the edge of the world. See "Not walking through objects" on page 222.
- Sliding along a wall if the user tries to move forward when blocked by an obstacle. See "Sliding along a wall" on page 237.
- Keeping the same distance from the ground. See "Hugging Terrain" on page 238.
- Moving to a different part of the 3D environment. See "Moving to a new zone" on page 240.

Providing feedback on the camera's current location and orientation

- Displaying the current location on a minimap. See "MiniMap" on page 233.
- Displaying an overview. See "Preset Views" on page 218.

Rotating around an object

For a straightforward product presentation, you may simply want the user to be able to rotate the camera around an object. This type of movement can be done with a generic behavior. To see an example of how to use either the mouse or the keyboard to rotate the camera around an object, download and launch the movie SimpleRotate.dir.



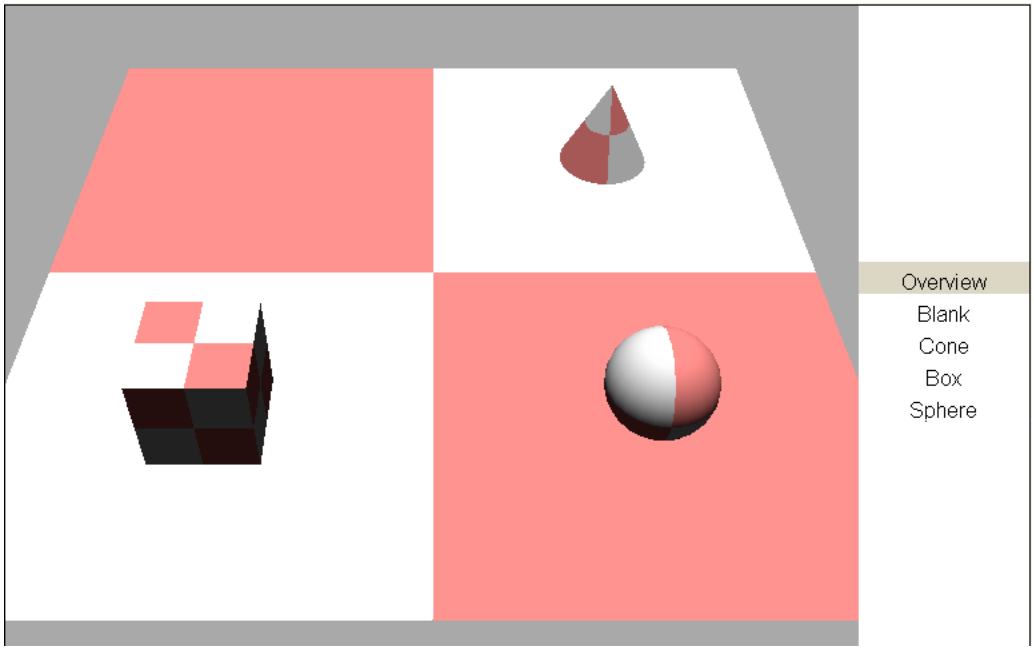
The further you drag the mouse from the point where you clicked, the faster the camera will rotate.

This movie uses two behaviors. The Drag to Rotate Camera behavior uses the movements of the mouse to control the rotation. The Move Camera Behavior uses the keyboard input from the arrow keys to control the rotation.

Preset Views

To demonstrate the architecture of a building, or as part of a guided tour of a virtual museum, you may want to move the camera to a preset viewing position.

To see a demonstration of this, download and launch the movie PresetViews.dir.



Click on the view names or on the colored squares on the floor plane to jump to a preset view.

In the PresetViews.dir movie, the data for the views is stored inside a property list. The property list contains information about the `position` and `rotation` of the camera transform, and about the camera's projectionAngle. Here's an extract that will create such a list as the `pViewList` property of a behavior:

```
property pCamera
property pViewList
on beginSprite(me)
  pCamera    = sprite(me.spriteNum).camera
  pViewList = [:]
  vViewData = [:]
  vViewData[#position] = vector( 270, 625, 820 )
  vViewData[#rotation] = vector( 22.5, 0, -90 )
  vViewData[#projectionAngle] = 35
  pViewList[#overview] = vViewData
  -- etc
end beginSprite
```

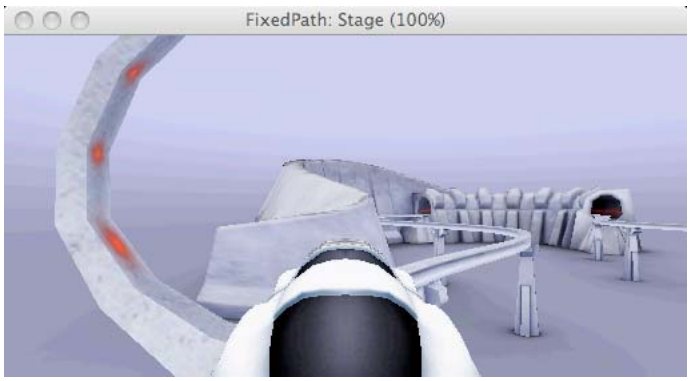
To apply a given preset view, the appropriate sub-list of pViewList is selected. The information contained in that sublist is then applied to the camera.

```
on SetView(me, aFloorName) -------------------------------------
  -- INPUT: <aFloorName> should be one of the symbols used as a
  --         property name in pViewList
  -- ACTION: Sets the transform and projectionAngle of the
  --         camera to recreate the preset view
  ---------------------------------------------------------------

  vViewData = pViewList[aFloorName]

  vPosition = vViewData.position
  vRotation = vViewData.rotation
  vProjectionAngle = vViewData.projectionAngle

  vTransform = transform()
  vTransform.position = vPosition
  vTransform.rotation = vRotation

  pCamera.transform = vTransform
  pCamera.projectionAngle = vProjectionAngle
end SetView
```

If you click inside one of the rooms which has a preset view associated with it, the SetView() handler will be called. For more details on how this is achieved, see "Picking" on page 242.

Following a pre-defined path

To create a fly-through of a scene, you may want to provide a pre-defined path that the visitor can follow. To see a demonstration, download and launch the movie FixedPath.dir.



The camera follows a track

The monorail track for the FixedPath.dir movie was created in 3ds Max. The track represents a spline which passes through a number of points in space. The designer exported the position of each of the points in the spline from 3ds Max as a text file. The beginning of the file looks like this:

```
200
0 0 0
0.23439370698124551 0 2.8052391247018487
0.79709400427770416 0 5.5735889387208921
1.4100892089568533 0 8.3189680738959133
2.2041555452853721 0 11.022120463300187
3.13483108877689 0 13.679583825865805
4.2009292738630357 0 16.286207871349418
5.38…
```

In other words, the file starts with the number of points, then a series of lines that begin with two TAB characters, then the `x` component, the `y` component (always 0), and the `z` component of a vector position in world space.

Tip: *In this movie, the camera is attached as a child to the Train model, and the train model follows the track. This helps you to visualize the spline in 3D space, and the movement of a node through space.*

If you are creating a pre-defined fly-through, you do not need to show either a visible track or a visible parent for the camera. You can simply move the camera between the points of an invisible spline.

The distances between adjacent points on the spline are not always equal, but the train needs to travel at a constant speed. The Train behavior uses the data from the text file to create four separate lists:

```
property plPoints   -- list of points on track
property plVectors  -- list of vectors that join the dots in
--                     plPoints
property plLengths  -- list of the lengths of the plVectors
property plTotals   -- sorted list of cumulated lengths in
--                     plLengths
```

On each `enterFrame`, the behavior determines how many milliseconds have elapsed since the train started, and calculates how far the train will have travelled in that time, as `vDistance`. The behavior then uses `plTotals.findPosNear(vDistance)` to determine which segment of the spline the train is currently on. It works out how far along that segment the train is, and what direction it is currently traveling in. This gives the current position and rotation for the Train model. Because it is a child of the Train model, the `worldPosition` and rotation of the camera is updated automatically to keep it in the same position relative to the train.

You can use the Train behavior as the starting point for your own fly-through projects.

You can find a demonstration of a similar technique at "Following a path" on page 267.

Steering with the mouse

There are several ways to control the camera with the mouse. This article explains a system based on the mouse position over the sprite. You can find a different solution at "Moving to a given location" on page 225.

Three in one

The Steer Camera With Mouse script deals with three different features of a walkthrough camera:

- Forward and backward movement, and steering.
- Looking around from a stationary position. For more information, see "Looking around" on page 222.
- Collision detection. For more details, see "Not walking through objects" on page 222.

Light on the wrist

In an action-packed game, the player will be moving around the world for much of the time. As the game designer, you will want to limit the amount of wrist movement that the player will need to make. The solution demonstrated here requires no pressure on the mouse button, and fluid movements of the hand.

In the center of the 3D sprite, both horizontally and vertically, there is a dead zone. If the mouse pointer is in near the center of the 3D sprite, the camera will not move. If the user moves the pointer more than 10 pixels to the left or right of center, the camera will start to turn. The further the pointer is from the center, on a horizontal axis, the faster the camera will turn.

If the user moves the pointer upwards, the camera will start to move forward. The further up, the faster the camera will move. The same will occur if the user moves the pointer downwards to make the camera move backwards.

Move later

A band 20 pixels wide in the center of the sprite (110 - 130 pixels from the top of the sprite) is a dead zone for forward and backward movement. Here is a handler similar to one that is used in the movie. It will move the camera forward or backwards along its own `zAxis`. The further the mouse pointer is from the center of the sprite, the faster the camera will move. This handler is called on every frame, after `mTurn()`.

```
on mMove(aCamera)
  vLocV = the mouseV

  vPixels = vLocV - 110
  if vPixels > 0 then
    vPixels = vLocV - 130
    if vPixels < 0 then
      -- The mouse is inside the dead zone
      exit
    end if
  end if

  -- Scale down movement to a reasonable speed
  vUnits = vPixels * 0.1

  aCamera.translate(0, 0, vUnits, #self)
end mMove
```

Looking around

In your 3D application, you may want the user to be able to stop at a chosen place and simply look around. You can use either the arrow keys or the mouse to provide a look-around feature.

Not walking through objects

You want the user to be able to walk through the world that you have created, but not to walk through objects. Moving a camera around a 3D space is not difficult. However, the illusion of the solidity of the 3D models will be shattered if the camera can move through walls.

Director provides a number of ways of detecting collisions with models. Depending on the circumstances, collision detection can require intense usage of the computer's processor. To prevent your end-user's computer overheating, or your movie playing back slowly, you will want to choose the least processor-intensive solution for each case.

modelsUnderRay()

The `modelsUnderRay()` command works fastest when it is used to detect only a small number of low-polygon models.

The concept

Imagine a hypothetical cylinder around the camera. Imagine that, each time the camera moves, the behavior needed to check for a possible collision between the cylinder and all solid objects in the scene. It would be time-consuming to check for an intersection between all the faces of the cylinder and all the faces of the models within range. Such a check would need to occur on every frame.

A simple solution is to create a barrier around each model. The barrier is a ribbon at a fixed distance from all the models. It represents the closest point that the center of this hypothetical cylinder can get to any model, before the cylinder starts to intersect with the model. All we now need to do is calculate whether the camera is about to touch the barrier. Each time the camera moves, it needs to make only one calculation: how far away is the barrier from the camera in the direction in which the camera is traveling?

To see how the barriers are created, see "2D barriers" on page 288.

One calculation per frame

The Steer Camera with Mouse behavior performs one `modelsUnderRay()` calculation on every frame, if the camera is moving. It sends out one ray in the direction in which it is moving. The `modelsUnderRay()` function returns a list with a number of properties, including `#distance`: the distance in world units from the origin point of the ray to where the ray intersects with a model.

Here is code similar to that used in the Steer Camera with Mouse behavior:

```
on mAttemptToMove(aCameraParent, aBarrierModelList, aDistance)
  vTransform = aCameraParent.getWorldTransform()
  vPosition  = vTransform.position
  vAxis      = -vTransform.zAxis -- camera moves down its zAxis
  -- Limit the scope of the modelsUnderRay() calculation
  vRayInfo   = [:]
  vRayInfo[#maxNumberOfModels] = 1
  vRayInfo[#levelOfDetail]     = #detailed
  vRayInfo[#modelList]         = aBarrierModelList

  -- Use a single modelsUnderRay() calculation
  vRayData = p3DMember.modelsUnderRay(vPosition, vAxis, pRayInfo)

  if vRayData.count then
    -- There is a barrier in the way
    vImpactData = vRayData[1]

    -- Do not let aCameraParent move right up against the
    -- barrier, otherwise rounding errors may put it just beyond
    -- the barrier, and  subsequent rays will fail to detect the
    -- barrier.
    vDistance   = vImpactData.distance - 0.1

    if aDistance > vDistance then
      -- Stop before the barrier
      aDistance = vDistance
    end if
  end if
  aCameraParent.translate(0, 0, aDistance, #self)
end mAttemptToMove
```

Controlling forward movement

This technique will work regardless of how the movement of the camera's parent node is controlled. See "Steering with the mouse" on page 221 for more details.

Advantages of using a parent node

This technique relies on the camera being attached as a child to a parent node. The parent node is moved around and the camera follows. Why not just move the camera around on its own?

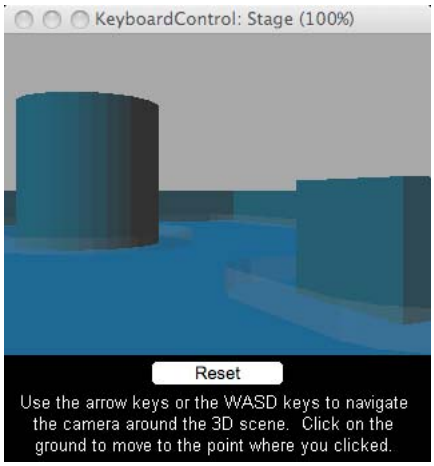
Here are the reasons:

- The `worldPosition` of the parent node determines the point of departure of the ray used by `modelsUnderRay()`. This needs to be at the same height as the barrier models. It is often simpler to place the barriers with their center at `y = 0`, even though the camera may need to be at a different height.
- The parent node's `zAxis` can always face forward, and remain perpendicular to the world's `yAxis`. You can tilt and swivel the camera and change its height independently of its movement over the terrain. See "Looking around" on page 222 for more details.

Steering with the keyboard

You may prefer to use the keyboard to move around inside the 3D world. Two common arrangements are to use the arrow keys with the right hand or to use the W, A, S and D keys with the left hand.

To experiment with using the keyboard to move around inside a 3D scene, download and launch the movie KeyboardControl.dir.



Use the arrow keys with your right hand or the WASD keys with your left hand to control the camera.

The KeyboardControl.dir movie contains a parent script called Steer With Keyboard. This script uses the keyPressed() function once per frame to determine whether the user is pressing certain keys. If so, it moves the parent node of the camera.

Turning

Here is a simple handler which checks whether any of the keys associated with turning have been pressed. If so, it rotates a node called `aCameraParent` around its own `yAxis`. For this to work correctly, the `yAxis` of `aCameraParent` must be aligned with the up axis of the world. You can find a similar handler in the Steer With Keyboard script in the KeyboardControl.dir movie.

```
on mTurn(aCameraParent)
  vLeft  = keyPressed("a") or keyPressed(123) -- left arrow
  vRight = keyPressed("d") or keyPressed(124) -- right arrow
  if vLeft = vRight then
    -- No keys pressed, or two keys cancel each other out
    exit
  end if
  if vLeft then
    aCameraParent.rotate(0, 1, 0, #self)
  else
    aCameraParent.rotate(0, -1, 0, #self)
  end if
end mTurn
```

Moving

Here is a simple handler which checks whether any of the keys associated with moving forward and backwards have been pressed. If so, it moves a node called `aCameraParent` along its own `zAxis`. You can find a similar handler in the Steer With Keyboard script in the KeyboardControl.dir movie.

```
on mMove(aCameraParent)
  vForwards  = keyPressed("w") or keyPressed(126) -- up arrow
  vBackwards = keyPressed("s") or keyPressed(125) -- down arrow
  if vForwards = vBackwards then
    -- No keys pressed, or two keys cancel each other out
    exit
  end if
if vForwards then
    aCameraParent.translate(0, 0, -1, #self)
  else
    aCameraParent.translate(0, 0, 1, #self)
  end if
end mMove
```

Collision Detection

The Steer With Keyboard script is designed to act as the ancestor for a behavior on a 3D sprite. The Key and Click Setup script creates the models used in the world, and prepares the data that will be required by the Steer With Keyboard. It uses a similar technique for collision detection as the one described in "Not walking through objects" on page 222.

Collision detection is performed once per frame, but only if the user is pressing one of the keys associated with forward or backward movements.

Click and go

The KeyboardControl.dir movie also demonstrates a technique where you click on the ground to move the camera to that point. For more details, see "Moving to a given location" on page 225.

Moving to a given location

You may want to allow the users of your 3D application to click anywhere in the scene, and have the first-person camera move automatically to that point. To experiment with this technique, download and launch the movie KeyboardControl.dir.

The click-to-go feature requires several different scripts to function:

- An instance of the Click To Go parent script is added to the userData list of the "Ground" plane, and it receives a `mouseUp()` call from the Pick Action behavior.
- The Pick Action behavior is attached to the 3D sprite. It detects when the user clicks on the 3D sprite, and forwards mouse events to the model that was clicked. For more details on how this works, see "Picking" on page 242.
- An instance of the Interpolate parent script is used to move the parent node of the camera. For more details, see "Interpolation" on page 264.

Click to go

The user can click the horizontal "Ground" plane model to define a target point for the camera to move to. In the KeyboardControl.dir movie, the parent node for the camera is not in the same plane as the ground. The Click To Go behavior does not move the parent node to the point that the user clicked on; it moves it to a point that is on a vertical line through the click point.

Here is a handler that sets two properties, `pStart` and `pEnd`, to the current position and the target position of the parent node of the camera:

```
property pStart -- worldPosition of the parent of the camera
--                 when the user clicks
property pEnd   -- worldPosition at the same height as the parent
--                 of the camera, above where the user clicked
on mouseUp(me)
  vSprite    = sprite(me.spriteNum)
  vCamera    = vSprite.camera
  vGround    = vSprite.member.model("Ground")
  vSpriteLoc = the mouseLoc - [vSprite.left, vSprite.top]

  -- Determine if the user clicked on the ground
  vRayInfo   = [:]
  vRayInfo[#maxNumberOfModels] = 1
  vRayInfo[#levelOfDetail]     = #detailed
  vRayInfo[#modelList]         = [vGround]

  vRayData = vCamera.modelsUnderLoc(vSpriteLoc, vRayInfo)
  if not vRayData.count() then
    -- The user did not click on the ground
    exit
  end if

  -- Get start position and put target position at same height
  pStart = vCamera.parent.worldPosition
  pEnd   = vRayData[1].isectPosition -- worldPosition of click
  pEnd.y = pStart.y

  me.mAttemptToMoveToClickPoint() -- (see explanation below)
end mouseUp
```

In the KeyboardControl.dir movie, the functionality of the handler above is treated in two different places. The creation of the `vRayData` list occurs in the Pick Action behavior, and the resulting list is sent to the `mouseUp()` handler of the Click To Go behavior, which executes the last four lines.

Jumping directly to the target position

In some cases, you may want to jump directly to the target position at `pEnd`. In such a case, use the following command instead of `me.mAttemptToMoveToClickPoint()`:

```
vCamera.parent.worldPosition = pEnd
```

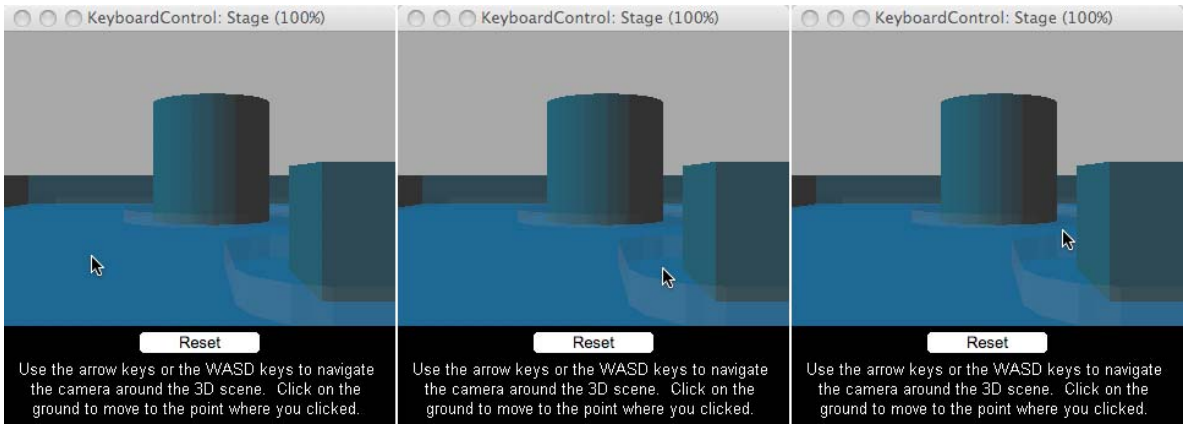
Attempting to move

In some cases, you will want the camera to move smoothly from the current position to the new position, without passing through any solid objects on the way. The first step, then, is to check if there are any barriers in the way.

The following are three possible cases:

- A barrier does not exist between the current position and the target position.
- The click is behind a barrier, in a position that the camera cannot reach.
- The click is on the far side of a solid object. The camera will have to move around the object in order to get to the target position.

These cases are illustrated in the following screenshots:



The target may be in open space (left), inaccessible (middle) or may require a detour around a solid object (right)

How does the Click To Go behavior distinguish between these three cases, and what does it do in each case?

Here is the `mAttemptToMoveToClickPoint()` method:

```
on mAttemptToMoveToClickPoint(me)
  -- Send a ray from pStart to pEnd looking for a barrier
  vImpact1 = me.mFindBarrier() -- 0 | [#isectPosition: <vector>]
  if not vImpact1 then
    -- There is no barrier in the way: move there directly
    return me.mStartMovingTowards(pEnd)
  end if

  -- If we get here, then the first ray hit a barrier. Send a
  -- second ray back from the target pEnd point towards pStart
  vImpact2 = me.mSendRayBack() -- 0 | [#isectPosition: <vector>]
  if not vImpact2 then
    -- The return ray did not hit anything, so the target
    -- position is inaccessible behind the barrier. Move as far
    -- as the barrier.
    return me.mStartMovingTowards(vImpact1.isectPosition)
  end if

  -- If we get here, then the return ray also hit a barrier. This
  -- suggests that there may be a path around the obstacle.
  -- Use a pathfinding technique.
  me.mFindWayAround(vImpact1, vImpact2)
end mAttemptToMoveToClickPoint
```

The `mFindBarrier()` and `mSendRayBack()` handlers work in similar ways. These handlers:

- Send a ray between the pStart and pEnd positions.
- Determine the distance to the nearest intersection point with a barrier.
- Compare this distance with the distance between the pStart and pEnd points.

They return either 0 (to indicate that there is no intervening barrier), or an intersection property list with the format:
`[#model: <model>, #position: <vector>, ...]`

They use a similar technique to the one used in the `mouseUp()` handler above.

Moving towards the target point

When the user releases the mouse, the camera starts to move towards the target position. If the user clicked in an inaccessible space behind a barrier, the movement will stop at the barrier.

The movement is launched by the `mStartMovingTowards()` handler. The movement takes some time. It is an asynchronous process. You can perform other actions while the movement is occurring. You can interrupt the movement by clicking a second time on the "Ground" model or by pressing on one of the camera control keys.

The movement is animated in two parts. First, the camera turns to face the target position, and then it moves towards the target position in a straight line. The two animations are controlled by two different instances of the same Interpolation parent script.

To use the Interpolation script, call its new handler, in the following way:

```
vInstance = script("Interpolation").new(aPropertyList)
```

Use the parameter `aPropertyList` as a property list with the following properties and values:

#object The 3D node to move, or the transform of the node to move.

#end Transform representing the end position, rotation and scale of the node.

#duration Duration of the movement in milliseconds.

#easing #none | #easeIn | #easeOut | #easeBoth

#callback Behavior or script instance to call back when the movement is complete.

#handler Symbol handler name to call when the movement is complete.

The `#easing` property in the parameter list decides whether the movement starts and ends abruptly, or whether it starts slowly, accelerate, and then slow down to a gentle stop. If a gradual start or finish is required, it uses a sine curve to ramp the speed.

The new instance of the Interpolation script places a pointer to itself on the `actorList`. It, therefore, receives a `#stepFrame` event once per frame, and uses this to control the movement of the camera's parent node.

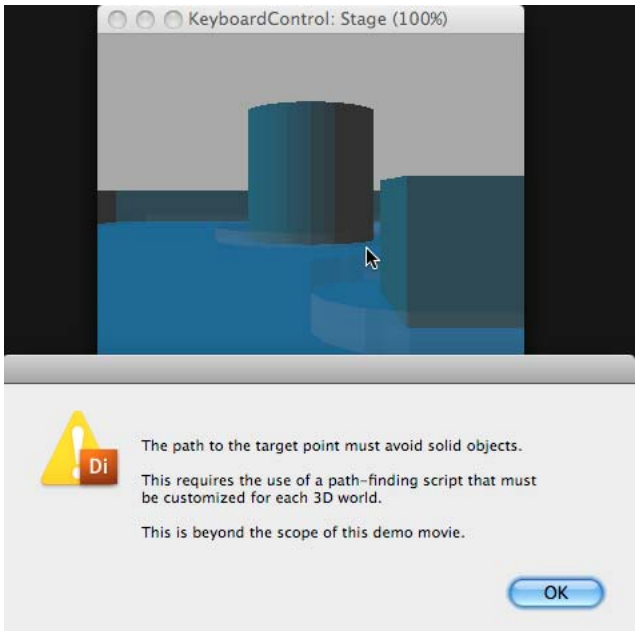
On every frame, it determines how many milliseconds have elapsed since the start of the movement, and uses this to calculate how far the parent node will have rotated or moved forward. The script instance stores the initial and final transform for the movement in the `pStartTransform` and `pEndTransform` properties. It stores a pointer to the transform of the node that it is moving as `pObject`. The key lines in the Interpolation script are:

```
vTransform = pStartTransform.interpolate(pEndTransform, vPercent)
pObject.position = vTransform.position
pObject.rotation = vTransform.rotation
pObject.scale    = vTransform.scale
```

For more details on interpolating between two transforms, see the interpolate() function.

Finding a path

In the KeyboardControl.dir movie, you can click in an empty space behind a solid object. The camera can move to the target position, but it cannot not travel in a straight line to get there. Each of your projects will use a different map, so you will need to use a customized path-finding technique. This movie does not demonstrate any path-finding techniques. Instead, it simply has a stub parent script called “Custom PathFinder”, which simply shows an alert.



The KeyboardControl.dir movie will not automatically find a path around an obstacle.

For more information on path-finding techniques to use in your projects, look up the A* algorithm by clicking the following links:

- Google search
- Wikipedia
- Lingo implementation

Third-person camera

In a 3D scene where the camera follows a user-controlled avatar, two levels of collision detection are required. Neither the avatar nor the camera must pass through any solid objects. If the camera moves through a wall, or even moves too close to it, the surface of the wall will appear to tear. The illusion of solidity will be lost.

To animate a third-person camera that follows an avatar, you will need to:

- Control the movement of the avatar in 3D space.
- Control the movement of the camera relative to the avatar.

In "Not walking through objects" on page 222, you learn how to simulate a cylindrical space around a 3D node, to prevent the node itself from colliding with walls. This concept is used here to create a zone around the avatar. If the camera is on the surface of the same cylinder then it will not pass through any walls either.

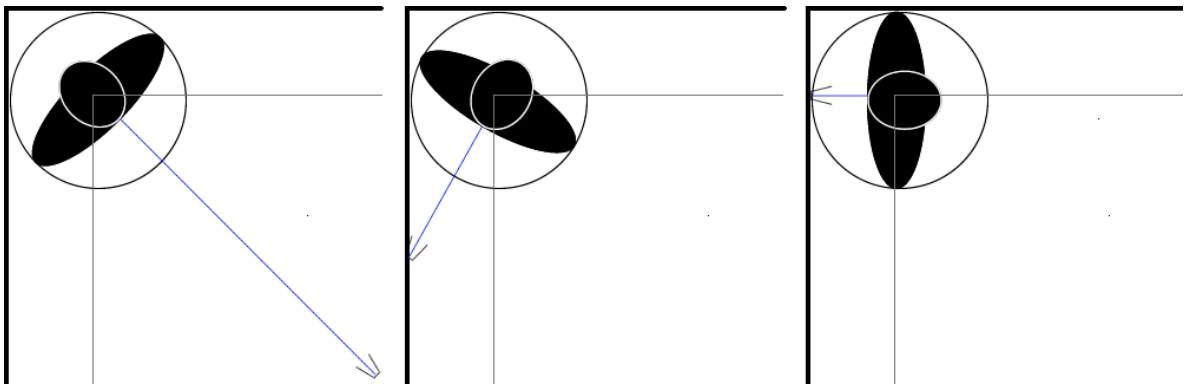
In "Steering with the keyboard" on page 224, you learn how to move a node around a 3D space. In this article, the same technique is used to move an object representing the user's avatar. The movement of the camera is treated as a second step.

The concept

A third-person camera normally follows an avatar from behind. If the avatar always moves straight forwards, the camera can simply move forward with the avatar. You can safely assume that the space between the avatar and the camera is clear of obstacles. The situation is not so simple if the avatar turns.

Imagine that the user steers the avatar into a corner. The avatar cannot move any further forward. To move elsewhere, the user first has to turn the avatar around to face away from the corner. To get behind the avatar if it turns, the camera must move into the space between the avatar and wall.

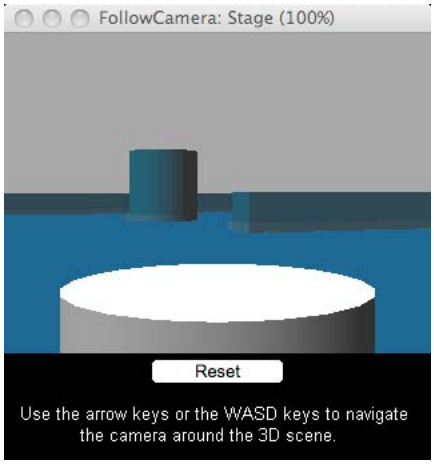
Imagine that the avatar is now standing with its back against the corner. There is no space between the imaginary cylinder that surround the avatar and the wall. As a result, the camera will have to move in towards the avatar until it is on the surface of the imaginary cylinder.



The distance that the camera lags behind the avatar depends on the relative position of walls

When the avatar moves away, the camera can stay in the same position until there is enough space between it and the avatar for it to start following the avatar again.

To experiment with this download the movie FollowCamera.dir and launch it.



Rotate the camera to see how it moves closer to the avatar if there is a wall behind it

Placing a script instance on a node's userData list

The code that determines the position of the camera relative to the avatar is contained in the Third-Person Camera script. An instance of the script is added to the camera's userData list, so that any script that has access to the camera object itself can call the Third-Person Camera instance.

Required information

The Third-Person Camera script requires the following information:

- The 3D member (or the sprite which contains the 3D member). This information is required in order to send `modelsUnderRay()` calls to detect barriers.
- A list of models used for collision detection.
- The radius of the imaginary collision cylinder around the avatar model.
- The maximum distance that the camera can lag behind the avatar.
- The camera to use to follow the avatar, if a 3D member or sprite has several cameras

The node that the camera is to follow is assumed to be the camera's parent.

Action

Each time the avatar model moves, the `Follow_Update()` handler in the Third-Person Camera script performs these actions:

- Send a ray backwards from the avatar's current position.
- Determine the distance along this ray to the nearest barrier model.
- Add the radius of the imaginary collision cylinder to this distance, to get the distance to the solid object behind the barrier.
- Decrease this distance, if necessary, to the maximum distance between the avatar and the camera.
- Move the camera the appropriate distance along the same line as the ray.

The camera is now behind the camera at the optimal distance, and will not collide with any walls.

Sample code

Here's a simplified version of the Third-Person Camera script:

```
-- pointer to the 3D member
property p3DMember
-- pointer to the camera object that is
-- controlled by this instance.  This
-- instance is added to pCamera.userData
property pCamera
-- radius of cylinder around the parent
-- node = minimum distance of pCamera from
-- parent
property pRadius
-- maximum distance of pCamera from parent
property pMaxDistance
-- [#maxNumberOfModels: 1,
-- #levelOfDetail:     #detailed
-- #modelList:         <list of models>]
property pRayInfo
-- pointer to the parent of pCamera = the
--                       node that pCamera will follow
property pParent
-- initial transform of pCamera,
--                       representing its vertical position at
--                       the center of the cylinder surrounding
--                       the parent node.
property pTransform
on new(me, a3DMember, aRadius, aMaxDistance, aBarrierList)
  p3Dmember    = a3DMember
  pRadius      = aRadius
  pMaxDistance = aMaxDistance

  pRayInfo = [:]
  pRayInfo[#maxNumberOfModels] = 1
  pRayInfo[#levelOfDetail]     = #detailed
  pRayInfo[#modelList]         = aBarrierList

  pCamera    = p3DMember.camera(1)
  pParent    = pCamera.parent
  pTransform = pCamera.transform.duplicate()

  (pCamera.userData)[#followCamera] = me
end new
```

```
on Follow_Update(me)
  vTransform = pParent.getWorldTransform()
  vPosition  = vTransform.position
  vZAxis     = vTransform.zAxis

  vRayData   = p3DMember.modelsUnderRay(vPosition, vZAxis, pRayInfo)

  if not vRayData.count then
    vDistance = pMaxDistance

  else
    vImpactData = vRayData[1]
    vDistance   = vImpactData.distance + pRadius
    vDistance   = min(vDistance, pMaxDistance)
  end if

  pCamera.transform = pTransform.duplicate()
  pCamera.translate(vZAxis * vDistance, #world)
end Follow_Update
```

Note: *You can use a third-person camera to follow any character, including characters that are not controlled by the user. The camera will follow whichever node you set it to follow. You will simply have to call the* `Follow_Update()` *handler each time the nod moves.*

MiniMap

You can use a secondary camera to create a minimap of your 3D world. If you place a camera directly overhead, facing downwards, it will automatically create a bird's view of the world. You can work with this view in several different ways. For example:

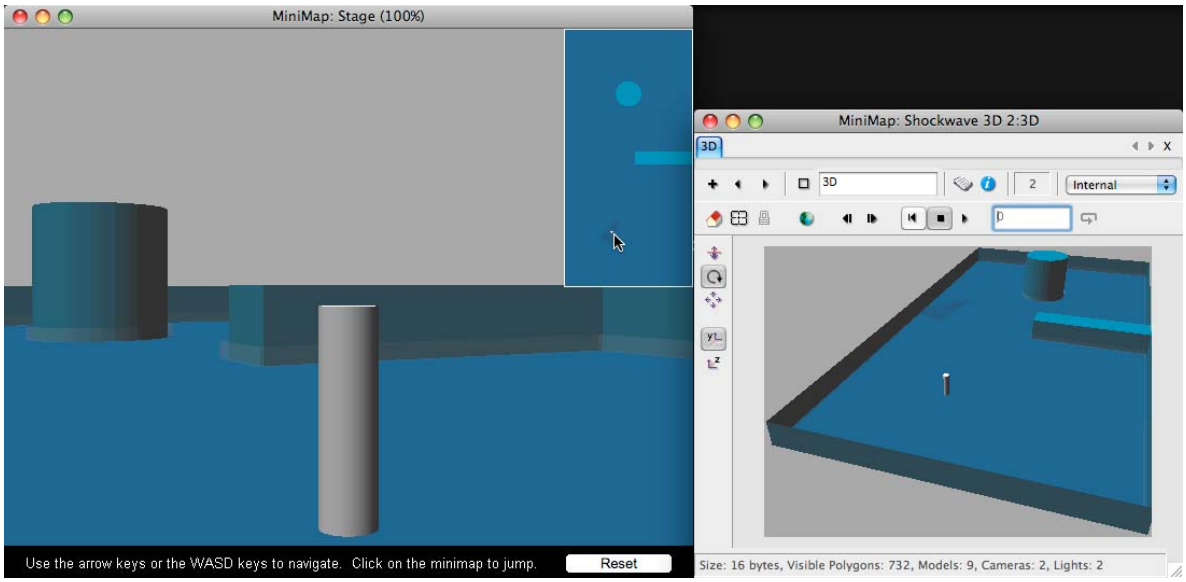
- As an insert in the main 3D sprite.
- As an alternative view in the main 3D sprite, that the user can switch to.
- As the main view in a second 3D sprite that displays the same 3D world.

Note: *Using a camera to display a minimap as an insert creates an opaque area within the main sprite. You may prefer to use a semi-transparent overlay to create your map. See* *"Overlays and backdrops" on page 47* *for more information.*

You can also choose between different ways of handling the movements of the minimap camera with respect to the user's avatar:

- The minimap camera can be a child of the avatar. In this case, the avatar will always appear in the center of the minimap, and the view in the minimap will always rotate so that the avatar's view looking towards the top of the map.
- The minimap camera may be in a fixed position. In this case, the avatar will move around the visible space. You may need an additional way of indicating the direction in which the avatar is facing.

To see an example of a minimap created as an insert with the camera in a fixed position, download the movie MiniMap.dir and launch it. This demo allows you to click inside the minimap to move the avatar to a new position.



A flattened cone (not visible in the main view) provides a direction arrow in the MiniMap view.

In the MiniMap.dir movie, the MiniMap behavior creates a second camera for the 3D sprite, and places it as in insert in the top right corner. The bird's-eye camera is given an `#orthographic` projection, so that there is no distortion of the view due to perspective. A cone-shaped primitive model serves to show the direction in which the avatar is looking. This model is attached as a child to the avatar group, and placed above the avatar so that it will not be visible in the main view.

Setting up the bird's-eye view camera

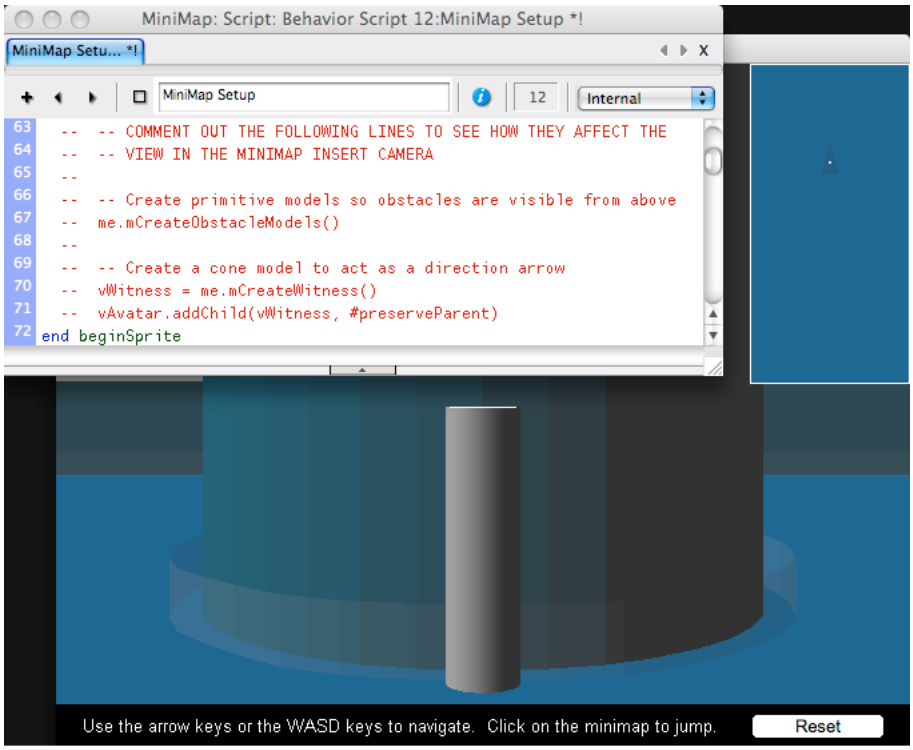
Here's a simplified version of the `mCreateCamera()` handler in the MiniMap behavior. It creates a new camera named "MiniMap", and sets its properties so that it looks down on the world, then adds it as an insert to the top right corner of the sprite.

```
on mCreateCamera(me, a3DSprite)
  vCamera = a3DSprite.member.newCamera("MiniMap")
  vCamera.projection = #orthogonal
  vCamera.orthoHeight = 1000
  vCamera.translate(0, 200, 0)
  vCamera.rotate(-90, 0, 0, #self)
  a3DSprite.addCamera(vCamera)
  vCamera.rect = rect(520, 0, 640, 240)
end mCreateCamera
```

Making objects visible from above

When you design the lighting for your 3D world, your main concern will be to make the scene look realistic from the avatar's point of view. In the MiniMap view from above, the lighting may not be perfect. Depending on your project, you can choose several solutions for this problem.

In the MiniMap.dir demo, two primitive models are used to create a “roof” for the two obstacle shapes. These models are placed with their top surfaces flush with the obstacle shapes. However, because the MiniMap camera view is orthographic, these models can be placed at any height, without affecting the view in the MiniMap camera. If you comment out the last 4 lines in the `beginSprite()` handler of the MiniMap Setup behavior, you will see how this affects the view in the MiniMap.



You will need to take care to make the world look good in the bird's-eye view

Jumping to a new position

Here's a `mouseUp()` handler that will move the group "Extraplanetary" to the position under the mouse in the MiniMap insert view, if the user clicks inside the MiniMap.

```
on mouseUp(me)
  vSprite  = sprite(me.spriteNum)
  vMember  = vSprite.member
  vCamera  = vMember.camera("MiniMap")
  vCamRect = vCamera.rect
  vSpriteLoc  = the mouseLoc - [vSprite.left, vSprite.top]
  if not (vSpriteLoc.inside(vCamRect)) then
    exit
  end if
  vCameraLoc = vSpriteLoc - [vCamRect.left, vCamRect.top]
  vPosition = vCamera.spriteSpaceToWorldSpace(vCameraLoc)
  vPosition.y = 0

  vAvatar  = vMember.group("Avatar_Parent")
  vAvatar.worldPosition = vPosition
end mouseUp
```

Note: *This demo does not check whether the user clicked on a solid object. In your projects, it is recommended that you add the appropriate code for this behavior.*

Rear-view mirror

You can use a very similar technique to create a rear-view mirror for a racing game. Simply create a secondary camera, add it to the sprite, and set it as a child of the user's vehicle, pointing backwards. To get a wide enough view for the rear-view mirror to be useful, you may wish to set the camera's projectionAngle property to a fairly large angle.

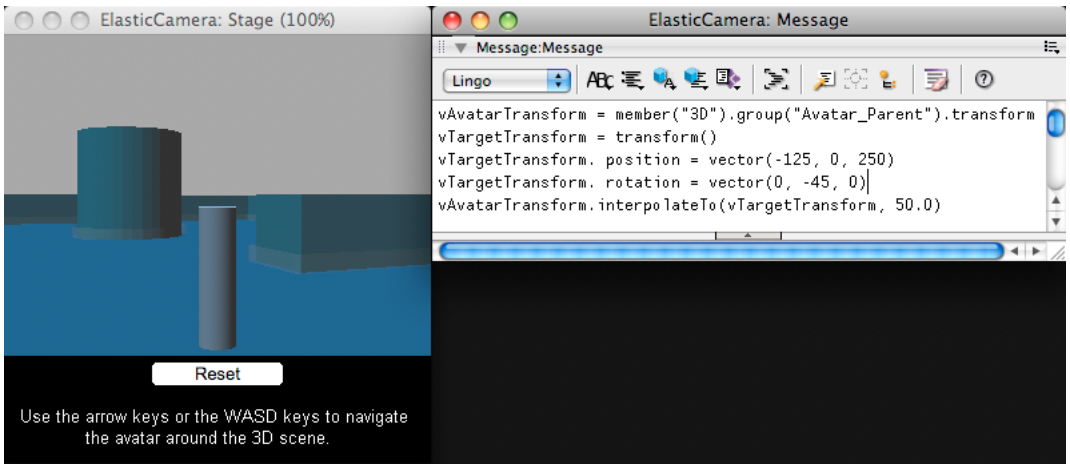
Making the camera move naturally

Your target audience is used to the camera movements used in the film industry. In the film industry, cameras are heavy and cannot be accelerated quickly. In a virtual world, a camera has no weight, so you can move it instantaneously from one point to another. However, your audience may see such movement as unnatural.

Both the avatar that you are following and the camera must start moving slowly, and increase in speed. If the avatar accelerates quickly, it looks more natural if the camera lags behind initially, and then catches up when the avatar has reached its top speed.

Instead of making the third-party camera follow at a fixed distance, you can vary the distance based on the changing speed of the avatar. One simple way to do this is to use the interpolate() and interpolateTo() methods.

To see an example of the use of `interpolateTo()` to make the camera move more smoothly, download and launch the movie ElasticCamera.dir.



Using interpolateTo() gives a natural lag between the movement of the avatar and the movement of the camera

You can find more details on this technique at "Interpolation" on page 264.

You can see an example of the `transform.interpolate()` method in the demo movie for "Moving to a given location" on page 225. In this example, a click on the Ground model will make the camera accelerate then decelerate as it arrives at its target position.

Other ideas

When navigating with the keyboard, you may prefer to ramp up to a maximum speed, and then continue to cruise at that speed, and finally ramp down to a halt when the user releases the mouse.

With a third-person view, you can associate each of these stages with a different motion for the avatar. The avatar can start with an "Idle" motion, then adopt an "Idle to walk" motion, then continue with a "Walk" motion. When the user releases the movement key, the avatar can adopt a "Walk to Idle" motion, and then return to "Idle". Each motion can be designed to move seamlessly into the next one.

See "Pre-defined animations" on page 270 for more details.

Pre-defined camera movements

Your 3D designer can create pre-defined motions that you may wish to apply to a camera. For example:

- When the player arrives at a portal in your 3D world, you may want the camera to shake as the avatar and the camera warp to a new location.
- When the player's avatar dies, you may want the camera face down at the avatar as it moves upwards like a spirit ascending.

You cannot apply motions directly to a camera object. You can only apply motions to models. Motions are controlled by the #keyFramePlayer or the #bonesPlayer modifiers. Models are the only type of node with which you can use the addModifier() method.

The solution is to attach the camera as a child to a model, and to apply the motion to the parent model.

You can create a model without any model resource, but you cannot add a modifier to a model that does not have any geometry. If you create the model with a model resource, and then set the resource of the model to `VOID`, the model's `modifier` list will be emptied.

As a result, you can only animate a camera with a motion if you attach it to a model that possesses geometry data, even if that geometry is never intended to be visible.The solution is to set the model.visibility for the model to #none. The plane("DefaultModel") resource is a good resource to use for this purpose.

```
v3DMember = member("3D")
vCamera     = v3DMember.camera(1)         -- built-in camera
vResource   = v3DMember.modelResource(1) -- built-in plane
vMotion     = v3DMember.motion(2)         -- custom motion
vMotionName = vMotion.name
vModel      = v3DMember.newModel("Camera_parent", vResource)
vModel.visibility = #none
vModel.addModifier(#keyFramePlayer)
vModel.addChild(vCamera)
vModel.keyFramePlayer.play(vMotionName)
```

The code above does the following:

- Creates a model.
- Makes the model invisible.
- Adds the #keyFramePlayer modifier to the invisible model.
- Adopts the camera of the 3D sprite as a child of the model.
- Animates the model and its child using a custom motion.

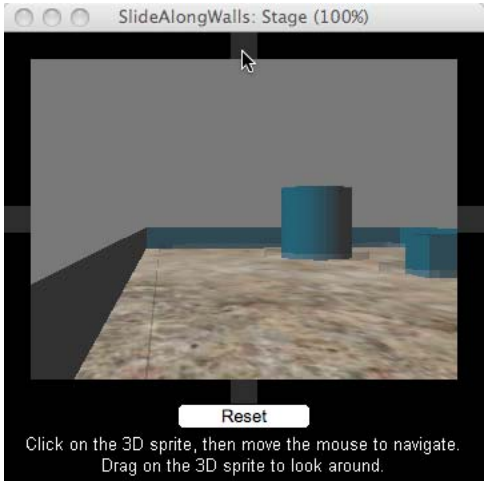
Sliding along a wall

What is to happen when the user navigates the first-person camera or an avatar up against an obstacle? In real life, people do not just walk into walls and remain stuck there; they start to move along the wall.

In a 3D world, you may want to choose between two different approaches to this issue:

- The camera or avatar remains facing the barrier, but moves sideways.
- The camera or avatar turns to face parallel to the wall, and then moves forwards.

To see a demonstration of the second solution, download and launch the movie SlideAlongWalls.dir. This movie uses a slight modification of the script used in the demo for "Steering with the mouse" on page 221.



In SlideAlongWall.dir, the camera turns to face along a barrier after coming close to it

Below you will find a handler similar to the one in the SlideAlongWall.dir movie. It accepts three parameters:

- `aNode`: a pointer to a node that may collide with a barrier
- `aDirection`: a vector pointing in the direction that aNode is attempting to travel
- `aNormal`: a vector pointing out from the barrier at right-angles to the point of collision

The handler will turn `aNode` by 2 degrees, or until it is no longer facing towards the barrier. This handler is called once per frame, so the node will turn automatically until it is facing along the barrier. The node can then continue to travel parallel to the barrier.

Note: *If the node were touching the barrier but facing at exactly 90° to the normal, then it may remain stuck. By using a value just over 90.0°, you can be sure that the node will be able to move after it has completed its turn.*

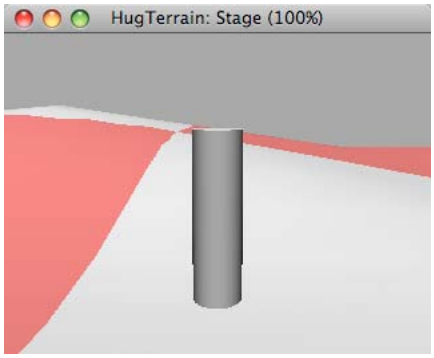
```
on TurnAlongBarrier(aNode, aDirection, aNormal)
  vAngle = aDirection.angleBetween(-aNormal)
  if not vAngle then
    -- Let the user choose which way to turn
    exit
  end if

  vPosition = aNode.worldPosition
  vAxis     = aDirection.cross(aNormal)
  vAngle    = min(2, (90.001 - vAngle))
  aNode.rotate(vPosition, vAxis, vAngle, #world)
end TurnAlongBarrier
```

Note: *If the user is controlling the rotation of the camera or avatar, then you will want to avoid applying an automatic rotation. The Steer and Slide Camera With Mouse behavior detects when the user is explicitly turning, and does not call the `TurnAlongBarrier()` handler when this occurs.*

Hugging Terrain

In your 3D world the ground may not be flat. As the camera or avatar moves around, you will want it to remain at the same position with respect to the terrain beneath it.

One easy way to achieve this is to send a ray downwards from above the camera or avatar, and detect where this ray intersects with the ground model beneath it. To see an example of this in action, download and launch the movie HugTerrain.dir.



The HugTerrain.dir movie move the avatar up and down as it moves over the terrain

Creating a terrain

The HugTerrain.dir movie uses a terrain mesh whose contours are defined by a height map stored in a grayscale image. For more information on this technique, see "Creating a terrain mesh" on page 176.

An alternative to the solution described here is to use the "Steering with the mouse" on page 221createTerrain() method of the Physics Engine to create a terrain which will detect collisions with objects. See "Terrains" on page 319 for more details.

Detecting the terrain under the avatar

The `HugTerrain()` handler below accepts three parameters:

- `aNode`: may be a first-person camera, an avatar, or the group which acts as the parent for the camera or the avatar
- `aTerrainModel`: the model that describes the collision geometry for the terrain
- `aDirection`: ±1, to indicate whether `aNode` is attempting to move forward or backwards.

The handler creates a temporary transform at the position where `aNode` can be after moving. It then raises the transform vertically by an amount greater than the greatest possible change in height on the terrain, and sends a ray downwards from that point. If the ray intersects with `aTerrainModel` then `aNode` can move to the position of the intersection between the ray and `aTerrainModel`.

```
on HugTerrain(me, aNode, aTerrainModel, aDirection)
  vTransform = aNode.transform.duplicate()
  vMoveToTry = vTransform.zAxis * aDirection
  vMoveToTry.y = 1000
  vTransform.translate(vMoveToTry)

  vPosition  = vTransform.position
  vAxis      = -vTransform.yAxis
  vRayInfo   = [:]
  vRayInfo[#maxNumberOfModels] = 1
  vRayInfo[#levelOfDetail]     = #detailed
  vRayInfo[#modelList]         = [aTerrainModel]
  vRayData = p3DMember.modelsUnderRay(vPosition, vAxis, vRayInfo)
  if not vRayData.count then
    return 0
  end if

  vImpactData = vRayData[1]
  vPosition = vImpactData.isectPosition
  vTransform.position = vPosition

  aNode.transform = vTransform
end HugTerrain
```

Note: *For simplicity, this handler assumes that* `aNode` *is a child of group("World"). If it has a different parent, then* `vTransform` *will need a further manipulation before being applied to* `aNode`*.*

Moving to a new zone

If your 3D world is complex, then it can be a good idea to break it up into sections. Each section can show a part of the world. When the user is visiting one section, only the models, shaders and textures for that part of the world need to be loaded into the computer's memory.

You can trigger a change of zone in many ways. For example:

- The player's character can move over a specific section of the ground (perhaps in a corridor). When a ray sent downwards from the character's position detects a specific model on the ground, the action moves to a new section.
- The player's character may move a given distance from the center of the current scene.
- The player may click a specific object.

User interaction

Users can interact with a 3D world by pressing keys on the keyboard by moving and clicking the mouse, or by tapping and dragging on a touch screen. This section looks at how you can detect and react to the user's input.

Mouse input

Director will natively react to mouse movement, and to left and right button clicks. You can find links to a number of free Xtra extensions here.

With a custom Xtra extension you can:

- Detect the use of the mouse's scroll wheel.
- Move the mouse pointer to any position on the screen without the user's intervention.

In a 3D world game where the user navigates with the mouse, it is important to reposition the mouse at the center of the screen continually. This means that the user can continue moving the mouse in one particular direction, without the pointer ever reaching the edge of the screen.

Touch screens

If your project is designed for a touch screen, then check whether the screen will react to:

- Both press and release (mouseDown and mouseUp)
- Dragging
- Dropping

Unless the user's finger is on the screen, your application will generate no `#mouseEnter`, `#mouseWithin`, or `#mouseLeave` events. Also take into account the size and lack of precision of fingers compared to the single-pixel accuracy of a mouse.

Director 11.5 does not natively detect multiple touches on a touch screen, so effects such as pinching and rotating with two fingers cannot be implemented without the use of a custom Xtra extension.

Clicking and dragging

With the mouse or a touch screen, the user can:

- Indicate which direction to move the camera or an avatar. See "Mouse control" on page 242 for more details.
- Select a model or a part of a model by clicking on it. See "Picking" on page 242 for more details.
- Determine the direction in which the user is pointing. See "Sprite space and world space" on page 247 for more details.
- Drag the mouse pointer to drag an object or draw on its surface. See "Dragging" on page 250 for more details.

Keyboard input

Director generates a `#keyDown` event when the user first presses any key, and repeats the event at regular intervals until the user releases all keys. You can use an `on keyDown()` handler to detect keyboard input.

In a 3D environment, you can use keyboard input in many ways. For example:

- To control the movement of the camera or an avatar. See "Steering with the keyboard" on page 224 for more details.
- To make an avatar perform specific actions.
- To cut to different camera views.
- To make menu selections.

The layout of keyboards varies. Different keyboards may not have the keys organized physically in the same locations. Your end users may be using keyboards that have been localized for their native language, so you cannot be sure that the characters that you are used to seeing together on your keyboard appear in the same place on your end-users' keyboards.

Director can react to multiple keys being pressed simultaneously. The number of simultaneous key presses that can be detected depends on the keys and on the end-user's operating system. See "Keyboard control" on page 254 for more details

One solution to this issue is to allow the end-user to customize which keys to use for which actions. See "Customizing control keys" on page 259 for more details.

Other input devices

You can find links to a number of third-party Xtra extension that allow you to control input from other devices here.

Mouse control

You can let the user control the movement of the camera or an avatar with the mouse in many ways. Here are some examples:

- With a fixed view camera and an avatar, you can make the avatar move towards the current position of the mouse pointer. To determine which point is currently under the mouse, see "Picking" on page 242 and "Sprite space and world space" on page 247.
- With any type of camera, you can move the character towards the point where the user last clicked. You can see an example of this in the demo movies for "Moving to a given location" on page 225 and "MiniMap" on page 233.
- With a first-person or third-person camera, you can use the horizontal movement of the mouse to indicate the direction in which the character has to turn, and the vertical movement to indicate the speed of its forward or backward motion. You can see an example of this in the demo movie for "Steering with the mouse" on page 221.
- With a first-person camera, you can use the mouse to control the direction in which the character is looking. You can see an example of this in the demo movie for "Looking around" on page 222.

If your project allows you to use a third-party Xtra extension, then you can also implement a system where the user turns the mouse scroll wheel to advance, or moves the mouse continuously in the direction of travel. See the tutorials here.

Picking

A 3D sprite simulates a 3D space on a 2D screen. On a 2D screen, the tip of the mouse pointer covers exactly one pixel. On top of a 3D sprite, the mouse pointer is like a finger pointing away from your eyes out towards the far end of the universe. The position of the pointer defines a line, not a point. The technical term for a line that starts at a point and continues to infinity is a *ray*.

Director provides two functions to let you discover what model or models appear under the mouse pointer: the simple modelUnderLoc and the more powerful modelsUnderLoc.

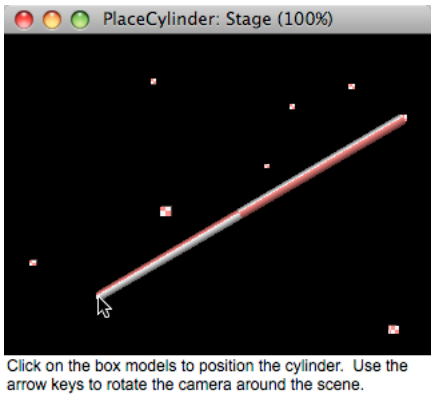
Tip: *You can also send out a ray from any point within a 3D world in any direction to see what model or models can be found in that direction. See modelsUnderRay for more details.*

To detect rigidBody and terrain objects, you can use these methods provided by the Dynamiks Xtra: rayCastClosest and rayCastAll.

modelUnderLoc()

This simple function returns `VOID` or a pointer to the model that the user clicked on. There may be several models at the position where the user clicked. The model that is returned will be the one whose clicked face is closest to the camera.

To see this in action, download and launch the movie PlaceCylinder.dir.



This movie uses modelUnderLoc() to determine which box model the user clicked

This movie creates a set of ten small box models and places them randomly in space. It also creates a cylinder model. When you click on the sprite, the behavior on the 3D sprite uses `modelUnderLoc()` to check if the click was on one of the box models. If so, it works out how to scale and position the cylinder so that it stretches to reach to the model the user clicked on.

Here's an extract of the mouseUp handler that discover which model the user clicked on. If the user clicked on the sprite, but did not click on any model, the handler exits.

```
on mouseUp(me)
  vSprite    = sprite(me.spriteNum)
  vCamera    = vSprite.camera
  vSpriteLoc = the mouseLoc - [vSprite.left, vSprite.top]

  vModel     = vCamera.modelUnderLoc(vSpriteLoc)
  if not vModel then
    exit
  end if
  -- More code to position the cylinder goes here
end mouseUp
```

modelsUnderLoc()

This function returns VOID or a list of models that appear at the given position in the 3D sprite. The method allows a number of different parameters. For more details, see modelsUnderLoc. The syntax discussed here is:

```
aRayDataList = camera.modelsUnderLoc(a2DPosition, aOptionsList)
```

The options list must be a property list that can have the following properties and values:

```
[#maxNumberOfModels: <integer>,
 #maxDistance:       <positive float>,
 #levelOfDetail:     <#simple | #detailed>,
 #modelList:         [<model>, ...]]
```

For example:

```
[#maxNumberOfModels: 1, #levelOfDetail: #detailed, #modelList:
[sprite(1).member.model("Target")]]
```

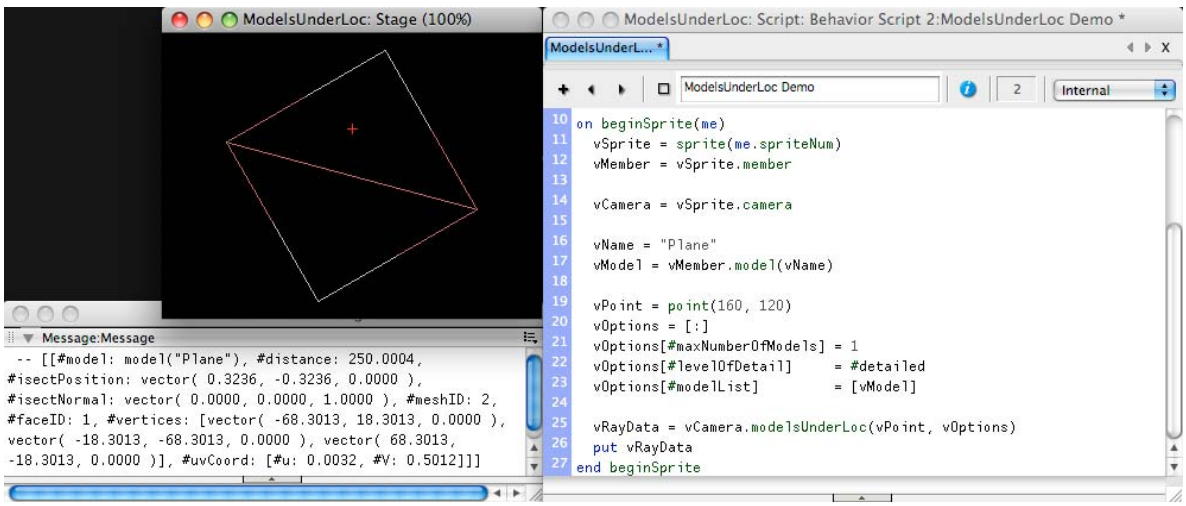
When you use this syntax and the value `#detailed` for the `#levelOfDetail` property, the output from the function is a list of property lists in the following format:

```
[[#model:          <model under the given loc>,
  #distance:       <float distance to intersection point>,
  #isectPosition: <vector worldPosition of intersection point>,
  #isectNormal:    <vector normal of the face at intersection>,
  #meshID:         <integer id of mesh to which the face belongs>,
  #faceID:         <integer id of intersected face>,
  #vertices:      [<vector>, <vector>, <vector>],
  #uvCoord:       [#u: <float>, #v: <float>]], ...]
```

The `#vertices` property is a list of the `worldPosition`s of the vertices at the corners of the face that was intersected by the ray.

The `#uvCoord` property gives you information about which pixel in the texture image was touched by the ray. See "Mapping a texture to a mesh resource" on page 150 for more details.

To see an example of `modelsUnderLoc()` used with the `#detailed` option, download and launch the movie ModelsUnderLoc.dir.



Using the #detailed option, modelsUnderLoc() provides a complete list of intersection data

The ModelsUnderLoc.dir movie simply prints the results of the `modelsUnderLoc()` function when it is applied to the point identified by the red cross.

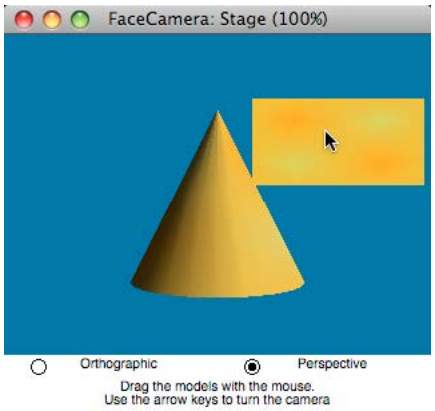
Pick Action behavior

On Director's 2D stage, you can drag and drop behaviors onto a sprite. See "3D behaviors" on page 75 for details of the built-in behaviors that you can use with 3D sprites.

In a 3D world, you cannot attach behaviors to specific models. However, every node has a userData property list, and you can add script instances to this list. This technique has many of the same characteristics as adding a behavior to a sprite.

You can add the Pick Action behavior to a 3D sprite to forward mouse events from the sprite to any script instances on the `userData` list of the model that is under the mouse.

To see this technique in action, download and launch the movie FaceCamera.dir.



The FaceCamera.dir movie adds instances of two different scripts onto the userData list of two different models

In the FaceCamera.dir movie, when you roll your mouse over the rectangular Plane model, the shader on the Plane model is swapped for a brighter shader. When the pointer rolls off the Plane, the original shader is swapped back. This feature is implemented by an instance of the Rollover Model script, attached to the Plane model's `userData` list.

If you click on the Plane model and drag the mouse, the Plane model will move with the mouse. This feature is implemented by an instance of the Drag Model script, attached to the Plane model's `userData` list. The same instance is also attached to the Cone model's `userData` list, so you can drag that around too. The Plane is a child of the Cone model, so it will move around with the Cone.

The Pick Action behavior reacts to mouse events such as `#mouseEnter` and `#mouseDown`, and forwards them to the `userData` list of the appropriate model.

Registration

You can use the Pick Action behavior's Behavior Parameters dialog to switch on model registration. If this option is selected, only models which have been registered through `Pick_RegisterModel()` will receive click events.

You can also choose to have registered models receive rollover events. If you select rollover detection, but not obligatory model registration, all models will receive click events but only registered models will receive rollover events (`#mouseEnter`, `#mouseWithin`, `#mouseLeave`).

Using model registration improves performance while using rollover detection degrades it.

Testing with a simple script

The FaceCamera.dir movie contains a very simple script called "Simple userData Example". The script's `new()` handler accepts two parameters:

- `a3DSprite`: the sprite containing the 3D world
- `aModel`: the 3D model to which the new instance has to be attached

When you call the `new()` handler, it:

- Adds a pointer to the new instance to the chosen model's `userData` list
- Registers the model with the Pick Action behavior, so that it will receive rollover mouse events.

```
on new(me, a3DSprite, aModel)
  vUserData = aModel.userData
  vUserData.addProp(#simpleDemo, me)
  sendSprite(a3DSprite, #Pick_RegisterModel, aModel)
  return me -- ... may not in fact be required
end new
on mouseEnter(me, a3DSprite, aRayData)
  vMember  = a3DSprite.member
  vColor = rgb(random(5)-1, random(5)-1, random(5)-1) * 51
  vMember.bgColor = vColor
end mouseEnter
```

To test this script, launch the movie, then execute the following line in the Message window:

```
script("Simple userData Example").new(sprite(1), sprite(1).member.model("Cone"))
```

If you now roll the mouse over the Cone model, the background color of the 3D member will change.

Parameters sent with mouse event handlers

The Pick Action can forward the following events to the appropriate models:

```
#mouseDown
#mouseUp
#mouseUpOutside
#rightMouseDown
#rightMouseUp
```

For models that have been registered, using a call to Pick_RegisterModel(), and for which rollover events have been requested, the Pick Action behavior will also forward the following events:

```
#mouseEnter
#mouseWithin
#mouseLeave
```

The Pick Action behavior sends two parameters with every mouse event call.

- A pointer to the 3D sprite.
- A property list containing the intersection data returned by the `modelsUnderLoc()` method, for the current model. See "Picking" on page 242 for a description of the contents of this list.

These two parameters give the instance access to all the data that it may need to react to the mouse event.

mouseUpOutside

If the mouse is clicked on one model and released over another, the first model will receive a `#mouseUpOutside` event and the second model will receive a `#mouseUp` event without the associated `#mouseDown`. This is the same functionality as with 2D sprites.

If the mouse is clicked on a model, then dragged off the sprite and released, the clicked model will receive a `#mouseUp` event if the mouse was over a part of the model that is cropped by the edge of the sprite. If the model is not "under" the mouse, or if a different (registered) model is between the model and the mouse, then the model will receive a `#mouseUpOutside` event.

Advantages of adding instances to a model's userData list

You can achieve the same effect by adding a separate behavior to the 3D sprite for each feature that you wish to add to the 3D world. If there are many models that need to react in different ways to mouse events, you can end up with many behaviors. These behaviors may repeat the same ray-casting action, just to detect whether their model has been clicked. This would be inefficient, and would slow down the action.

If you place the Pick Action behavior on the sprite, you can execute one single `modelsUnderLoc()` command, and then inform only the model that the user clicked.

For certain actions, you can add the same instance of a script to the `userData` list of several different models. (This is how the FaceCamera.dir movie works with the Drag Model instance).

The code for a script that is attached directly to a model can be much simpler than the code of a behavior attached to the sprite since all the mouse event detection is taken care of by the Pick Action behavior. The instance on the `userData` list just needs to process the results of the mouse event.

Disadvantages

You can add behaviors to a 3D sprite by dragging and dropping them from a cast library. You can use the Property Inspector at the Behavior tab to check which behaviors are attached to the sprite, and to set their parameters.

You need to use code to add a script instance to the `userData` list of a model, and Director provides no window for you to inspect these instances. These instances can only be added at run-time. When you save the movie and quit, each model's `userData` list will be reverted to its original content.

stopEvent

There is a major difference between this 3D technique of placing instances on a model's `userData` list, and placing 2D behaviors on a sprite.

One behavior instance in a sprite's `scriptInstanceList` can use the `'stopEvent'` command to prevent an event from being passed on to later behaviors on the same sprite. However, the `stopEvent` command has no effect on the `call()` command used in the Pick Action script.

When you use `call(#someEvent)` to send a message to the list, the `on someEvent()` handler will be run in all instances in the list where it appears.

To work around this, one instance can add a `#stopEvent` property with a non-void value to the Ray Data List that is sent as the second parameter to all calls made to a model's `userData` list. Subsequent instances on the `userData` list can check for the presence of such a property, and choose not to execute any code if they find it.

If the Pick Action behavior detects that such a property has been added to the Ray Data List after a call to the `userData` list, it will execute the `stopEvent()` command, and no subsequent behaviors on the same 3D sprite will receive that event.

Sprite space and world space

You can map any point within a camera's field of view onto a single pixel at the surface of the 3D sprite. However, a single point on the surface of a 3D sprite defines an infinite number of positions in the 3D world.

Director provides you two functions for mapping 3D positions to 2D points and vice versa:

- `a2DPointOnSprite = camera.`**`worldSpaceToSpriteSpace`**`(a3Dposition)`

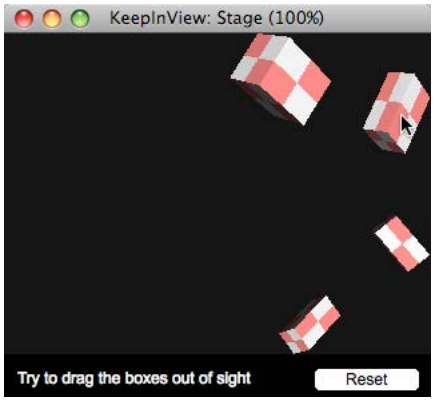
- `aArbitrary3Dposition= camera.spriteSpaceToWorldSpace(a2DPointOnSprite)`

worldSpaceToSpriteSpace()

If you know the co-ordinates of a point in 3D space, you can determine where that point will appear on the 3D sprite using the worldSpaceToSpriteSpace method. For a given camera at a given moment in time, this method will give either:

- A 2D point, measured from the top left corner of the sprite, if the 3D position is visible in the sprite, or
- VOID, if the 3D position is not within the camera's field of view.

To see an example of `worldSpaceToSpriteSpace()` in use, download and launch the movie KeepInView.dir.



You can drag a box to any position where all of its corners appear within the camera's field of view

Using worldSpaceToSpriteSpace()

When you click on a box, the `mouseDown()` handler calculates where the center of the box model appears in the sprite view. It uses this information to determine a 2D offset from the point where the user clicked to the apparent center of the model. It stores this value as a point in the property `pOffset`:

```
property pOffset -- offset between mouse and center of box model
on mouseDown(me)
  vSprite = sprite(me.spriteNum)
  vCamera = vSprite.camera
  vSpriteLoc = the mouseLoc - [vSprite.left, vSprite.top]
  vModel = vCamera.modelUnderLoc(vSpriteLoc)

  if not vModel then
    -- The user clicked outside any models
  else
    vWorldPosition = vModel.worldPosition
    vLoc = pCamera.worldSpaceToSpriteSpace(vWorldPosition)
    pOffset  = vSpriteLoc - vLoc
  end if
  -- More code...
end mouseDown
```

As you drag one of the boxes around inside the 3D world, the `mModelIsInView()` handler executes a line similar to this, for the proposed `worldPosition` of each of the corners of a box:

```
vSpriteLoc = sprite(1).camera.worldSpaceToSpriteSpace(vCorner)
```

If the resulting `vSpriteLoc` is `VOID`, then the box is not allowed to move to the proposed new position.

Note: *For more information on how the dragging feature is implemented, see "Dragging" on page 250.*

spriteSpaceToWorldSpace()

The function spriteSpaceToWorldSpace works in reverse. For input, it accepts a 2D point, representing an offset from the top left corner of the 3D sprite as a parameter. For output, it returns a 3D position vector representing one of the points in the 3D world that appears under that point in the sprite.

At that location in the sprite, there will be an infinite number of 3D points. The precise value of the point is rarely useful on its own. However, if you subtract the `worldPosition` of the camera from the value returned by `spriteSpaceToWorldSpace()`, you obtain a vector pointing from the camera into the 3D world through that point.

Here is a handler called `GetRayUnderLoc()` that accepts two parameters:

- `aSpriteLoc` : a 2D point, representing an offset from the top left corner of the 3D sprite
- `aCamera`: the camera of the 3D sprite

The GetRayUnderLoc()handler returns a unit vector which points from the camera into the world. In other words, `GetRayUnderLoc()` tells you the direction in which the mouse is pointing when it is at `aSpriteLoc`.

```
on GetRayUnderLoc(aSpriteLoc, aCamera)
  vCameraLoc = aCamera.worldPosition
  vDistantLoc = pCamera.spriteSpaceToWorldSpace(aSpriteLoc)
  vRay = vDistantLoc - vCameraLoc
  vRay.normalize()
  return vRay
on GetRayUnderLoc
```

Placing a model under the mouse

Here's a `mouseUp()` handler from a 3D behavior. It will place a model named "Ball" so that its center appears directly under the mouse at a distance of 200 world units from the camera:

```
on mouseUp(me)
  vSprite      = sprite(me.spriteNum)
  vCamera      = vSprite.camera
  vCameraPoint = vCamera.worldPosition
  vBall        = vSprite.member.model("Ball")

  vSpriteLoc   = the mouseLoc - [vSprite.left, vSprite.top]

  vWorldSpace  = vCamera.spriteSpaceToWorldSpace(vSpriteLoc)
  vRay         = vWorldSpace - vCameraPoint
  vRay.normalize()

  vBallPoint   = vCameraPoint + vRay * 200
  vBall.worldPosition = vBallPoint
end mouseUp
```

To see a demonstration of this handler, download and launch the movie PlaceBall.dir.

Dragging

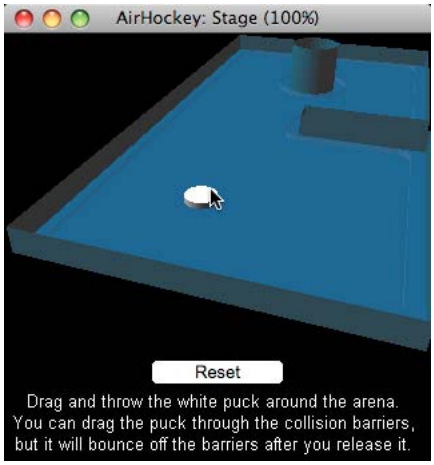
To drag a model about in 3D space using a 2D mouse, you must limit the movement of the model to one 2D surface within the world. Here are some examples:

- A horizontal plane where all points share the same vertical co-ordinate (parallel to the ground).
- The surface of a terrain model.
- A vertical plane, representing a wall.
- A plane that is perpendicular to the camera's zAxis (parallel to the plane occupied by the sprite viewport).

To map the 2D position of the mouse onto the appropriate plane in the 3D world, use vector and transform mathematics. For more information on 3D mathematics, see "3D mathematics" on page 361. In this article, you can find reusable solutions for dragging (but not mathematical explanations on how they work).

Dragging a model on a horizontal plane

To see an example of this download and launch the movie AirHockey.dir.



The air hockey puck will move over a horizontal plane as you drag it around

The Air Hockey behavior creates a white model named "Puck". The Throw Model in Y Plane script uses `spriteSpaceToWorldSpace()` to find a point under the mouse. It then imagines a line between the camera and that point, and works out where that line crosses the horizontal plane where the Puck model is, and moves it to that point. Here is a very simple behavior that does the same thing:

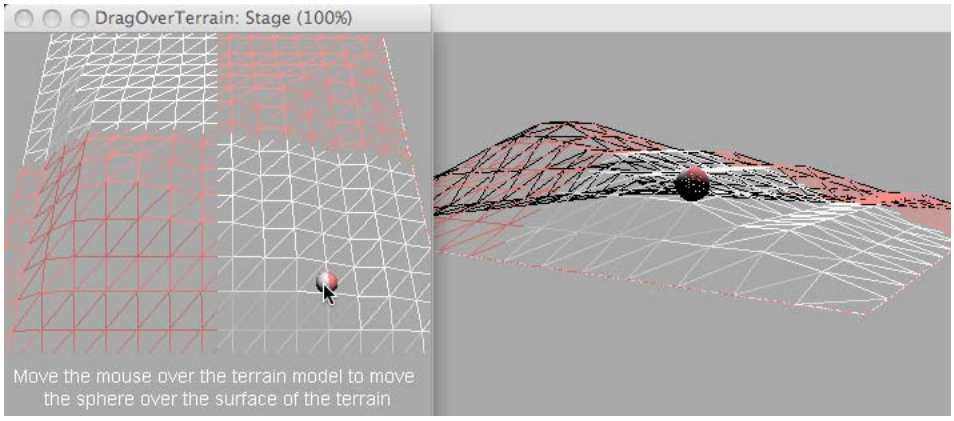
```
on exitFrame(me)
  vSprite      = sprite(me.spriteNum)

  -- Find the Puck model and its y position in world space
  vModel       = vSprite.member.model("Puck")
  vYValue      = vModel.worldPosition.y

  -- Find where the mouse is relative to the sprite
  vSpriteLoc   = the mouseLoc - [vSprite.left, vSprite.top]

  -- Find the camera position and a world point under the mouse
  vCamera      = vSprite.camera
  vCameraPoint = vCamera.worldPosition
  vWorldPoint  = vCamera.spriteSpaceToWorldSpace(vSpriteLoc)

  -- Find how far along the line between the camera and
  -- vWorldPoint to travel in order to have the correct value
  -- for the y coordinate
  vCameraY     = vCameraPoint.y
  vYOffset     = vCameraY - vWorldPoint.y
  vHeight      = vCameraY - vYValue
  vRatio       = vHeight / vYOffset

  -- Move that distance along the ray from the camera
  -- towards vWorldPoint
  vVector      = (vWorldPoint - vCameraPoint) * vRatio
  vModelPoint  = vCameraPoint + vVector

  -- Move the model to that point
  vModel.worldPosition = vModelPoint
end exitFrame
```

The Air Hockey movie also demonstrates how to throw a model, and how to make it bounce off barriers. For more details of these features, see "Linear motion" on page 261 and "Bouncing off a wall" on page 291.

Dragging a model over a terrain

To drag one model over the surface of a terrain model, you need to know where the ray sent into the 3D world from the mouse pointer's current position intersects with the surface of the terrain model. You can use `modelsUnderRay()` with the `#detailed` option to do this.

To see this in action, download and launch the movie DragOverTerrain.dir. You can use the Shockwave3D viewer window to look at the scene from a different angle.



Use modelsUnderRay() with the #detailed option to find where a ray from the mouse touches a model

The following handler places the group "Sphere_Parent" on the point on the surface of the model "Terrain", which appears directly under the mouse.

```
on exitFrame(me)
  vSprite    = sprite(me.spriteNum)
  vMember    = vSprite.member
  vParent    = vMember.group("Sphere_Parent")
  vTerrain   = vMember.model("Terrain")

  -- Find the point on the terrain under the mouse
  vCamera    = vSprite.camera
  vSpriteLoc = the mouseLoc - [vSprite.left, vSprite.top]
  vRayInfo   = [:]
  vRayInfo[#maxNumberOfModels] = 1
  vRayInfo[#levelOfDetail]     = #detailed
  vRayInfo[#modelList]         = [vTerrain]
  vRayData   = vCamera.modelsUnderLoc(vSpriteLoc, vRayInfo)

  if not vRayData.count then
    exit
  end if

  vPosition  = vRayData[1].isectPosition
  vParent.worldPosition = vPosition
 end exitFrame
```

You can use a similar technique to drag a model over the surface of any model. To place a model representing a painting on a vertical wall, you would simply use the Wall model in the `#modelList` of the `modelsUnderLoc()` options list.

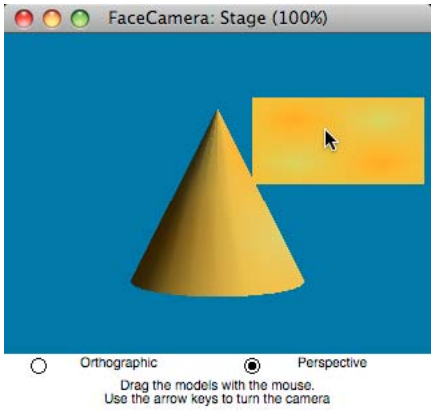
Dragging a model parallel to the sprite viewport

To drag a model in the plane that is parallel to the sprite viewport, you need to know which way the camera is pointing. This is given by the zAxis property of the camera's transform in world space.

```
vZAxis = aCamera.getWorldTransform().zAxis
```

Tip: *In fact, the camera's zAxis points out of the sprite towards you.*

The technique that you need to use is different depending on whether the camera is showing a `#perspective` projection or an `#orthographic` projection. The technique described here is for a `#perspective` projection. To find code for both techniques, download and launch the movie FaceCamera.dir.



The FaceCamera.dir movie demonstrates two techniques for dragging a model parallel to the sprite viewport.

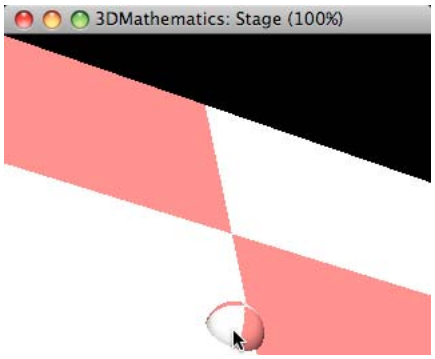
The FaceCamera.dir movie contains a Drag Model script. Here is a simplified version of a handler from that script:

```
on PlaceModelUnderMouse(aSprite, aModel)
  -- Find the camera, its zAxis and aModel in world space
  vCamera      = aSprite.camera
  vTransform   = vCamera.getWorldTransform()
  vCameraPoint = vTransform.position
  vAxis        = vTransform.zAxis
  vModelPoint  = aModel.worldPosition
  -- Find a (distant) point beneath the mouse, in the world
  vSpriteLoc = the mouseLoc - [aSprite.left, aSprite.top]
  vWorldPoint  = vCamera.spriteSpaceToWorldSpace(vSpriteLoc)

  -- Create a unit vector pointing from the camera to this point
  vRay         = vWorldPoint - vCameraPoint
  vRay.normalize()

  -- Do a little mathematics, based on the fact that the
  -- dotProduct of two perpendicular vectors is zero.
  vCameraToModel = vModelPoint - vCameraPoint
  vDistance      = (vCameraToModel * vAxis) / (vRay * vAxis)
  vNewPosition   = vCameraPoint + (vRay * vDistance)

  -- Move the model to the new position
  aModel.worldPosition = vNewPosition
end PlaceModelUnderMouse
```

The handler used in the Drag Model script also takes into account the offset from the origin point of the model that is being dragged and the point on the model on which the user clicked.

Additional example

The movie 3DMathematics.dir demonstrates a technique for dragging a model over an arbitrary plane.



Using RayCutsPlane() to drag a model across an arbitrary plane

Keyboard control

Director provides two events for handling keyboard input, six properties and a function:

- on keyDown: Triggered when the user first presses a key, but before the key is displayed in any editable field or text member. This event is also called at regular intervals if a key is held down.
- on keyUp: Triggered when the user releases a key, and after the display of a field or text member has been updated.
- _key: The character the user typed.
- keyCode: An integer related to the physical key that the user pressed (Mac) or the character that the user typed (Windows).
- commandDown, controlDown, optionDown, shiftDown: Returns TRUE if the associated modifier key is pressed.
- keyPressed() : Returns the last char to be pressed if no parameter is used, or a boolean if char or keyCode is used as a parameter.

To determine whether the Caps Lock key is pressed, you can use this free third-party Xtra extension.

3D sprites and keyboard focus

3D sprites can become the keyboardFocusSprite. When this is the case, `#keyDown` and `#keyUp` events are directed at the sprite, and can be treated with `on keyDown()` and `on keyUp()` handlers in a behavior on the 3D sprite.

If you have another editable sprite on the Stage, then the user will have to transfer the keyboard focus to the 3D sprite after editing the field or text sprite. To do this, they can click on the 3D sprite. Alternatively, if the editable member's `autoTab` property is `TRUE`, they can use the `TAB` key to move the keyboard focus to the next sprite.

3D members do not have an `autoTab` property that you can set. They behave as if this property was always `TRUE`. Suppose there is an editable sprite on the Stage and the user clicks on a 3D sprite to send keyboard input to that sprite. Suppose the user then presses the `TAB` key, perhaps by mistake. The keyboard focus will now be directed to the editable sprite; the 3D sprite will receive no further input until the user activates it again.

When the browser window containing a Shockwave movie loses focus, a 3D sprite in the movie will stop being the keyboardFocusSprite. You can place a behavior on the 3D sprite to detect when it loses keyboard focus, and display an overlay saying "Click here to continue" when that happens. This message will prevent users from becoming confused if their keyboard input suddenly stops having any effect on the 3D world. See "Customizing control keys" on page 259 for a demo of this technique.

Keyboard issues

Before you start planning a keyboard control system for your 3D movie, go through the main issues of using keyboard input:

- You cannot be sure what character will be produced by a given physical key. For different input languages, the arrangement of the characters on a keyboard is different. On an English-language keyboard, the first five letters are QWERTY. On some French-language keyboards, the same keys denote the letters AZERTY.
- The physical layout of a keyboard is not standardized. For a given input language, you may find certain keys in different places on different keyboards. Certain ergonomic keyboards make it difficult to press certain key combinations.
- The value of `_key.keyCode` may be different on Mac and on Windows for keyboards other than the standard US English layout.
- Many users for whom English is not a first language will have keyboards set to input Unicode characters, rather than Roman characters. On Windows, in Director 11.5, the `_key.keyPressed()` function does not work correctly. As a result, you may not be able to detect keyboard input correctly for many languages.
- In Director 11.5, the results you see in Director's authoring environment may be different from those you see in Shockwave.
- Director can react to multiple keys being pressed simultaneously. The number of simultaneous key presses that can be detected depends on the keys and on the end-user's operating system. For example, if a user holds down all 4 arrow keys, only 2 `#keyDown` events will be generated.

Detecting keyboard input

Director generates a on keyDown event when a key is first pressed. In an `on keyDown()` handler, you can use keyCode, _key, or keyPressed() to determine which key was pressed most recently.

The `#keyDown` event is repeated continuously while any key is held down, but the value of the `_key` properties may change if new keys are pressed. Director generates a `#keyUp` event when any key is released, but it does not provide any specific information about which key was released.

To test how Director reacts to keyboard input, download and launch the movie KeyPressed.dir. If you are working on Mac OS X, then you will find it useful to open Apple's Keyboard Viewer application at the same time.

Try pressing all 8 "home" keys (ASDF JKL;) at the same time. You will see that both Windows and the Mac OS X operating system only detects the first 6 keys that were pressed, and passes the input information to Director.

Monitoring which key was released

When Director generates a #keyUp event, it does not provide any information about which key was released. To work around this shortcoming, the KeyPressed.dir movie maintains a list of which keys are currently being held down.

```
global glKeysPressed
on startMovie()
  glKeysPressed = [:]
  glKeysPressed.sort()
end startMovie
on keyDown()
  vKey  = _key.key
  vCode = _key.keyCode
  if not glKeysPressed.findPos(vKey) then
    glKeysPressed.addProp(vKey, vCode)
  end if
end keyDown
```

When a `#keyUp` event is generated, the script checks which keys recorded in glKeysPressed are still being pressed, and deletes any that are not.

```
on CheckKeysPressed()
  ii = glKeysPressed.count
 repeat while ii
    vKey = glKeysPressed.getPropAt(ii)
    if not keyPressed(vKey) then
      vCode = glKeysPressed.getAt(ii)
      if not keyPressed(vCode) then
        glKeysPressed.deleteAt(ii)
      end if
    end if
   ii = ii - 1
  end repeat
end CheckKeysPressed
```

Note: *This handler checks for both `keyPressed(aChar)` and `keyPressed(aKeyCode)` before considering that a key has been released. See "Issues with keyPressed() on Windows" on page 258 and "Issues with keyPressed() on Macintosh" on page 257 for more details.*

Visualizing different keyboard layouts

To display a keyboard viewer on Mac OS X, you can open the System Preferences at the Input Menu tab of the International pane. Check the Keyboard Viewer and Show Input Menu In Menu Bar boxes. You now see an Input menu to the right of the menu bar. You can select Show Keyboard Viewer from this menu. You can also use the same tab in the System Preferences window to select a variety of different keyboard layouts.

Note that some layouts will produce input which will not display correctly in all fonts. In Director, choose a font that is appropriate for the input language.

On Windows, you can display an On-Screen Keyboard, by selecting Start > All Programs > Accessories > Accessibility > On-Screen Keyboard. This view allows you to visualize the keyboard layout, but it does not provide you with feedback on which keys are pressed.

To add an input layout on Windows, you need to choose Control Panel, then Regional and Language Options. Click the Languages tab and then the Details button. In the Text Services and Input Languages window, click Add, and then make your selections.

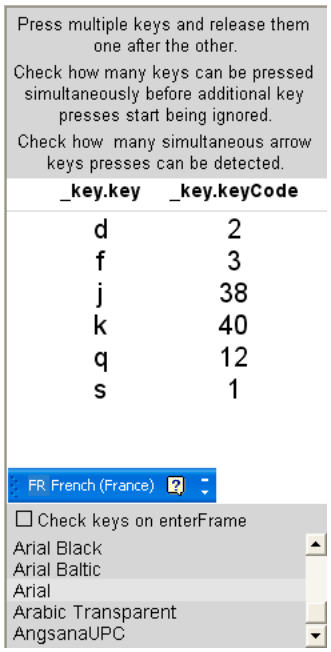
An input selection menu will appear near the right end of the task bar. You can select which input layout and language you wish to use there.

Characters, key and keyCode

On Macintosh, the value of `_key.keyCode` depends on the actual physical key that the user presses. On an English keyboard, the key beside Caps Lock is an A; on French keyboard it is a Q. In both cases, the value of `_key.keyCode` produced by pressing this key will be 0.

This means that, on Macintosh, you can use `_key.keyCode` to indicate exactly which physical key the user pressed.

On Windows, the value of `_key.keyCode` depends on the character that is generated when the key is pressed. Pressing A on an English keyboard or A on a French keyboard will lead to the same value of _key.keyCode, even though different physical keys are pressed.



Pressing the key beside the Cap Locks key produces Q in French, and a _key.keyCode of 12

Issues with keyPressed() on Macintosh

In Director 11.5, on Macintosh, the values of `_key.key` and `_key.keyCode` immediately after a `#keyDown` event represent the character that was typed and the key that was pressed respectively. However, `keyPressed(aChar)` assumes that the user is typing on a standard US keyboard. Here is a simple Movie Script handler that you can use to check this:

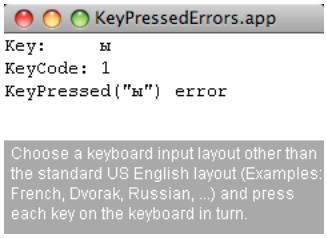
```
on keyDown()
  if not _key.keyPressed(_key.key) then
    put "KeyPressed("&QUOTE&_key.key&QUOTE&") error"
  end if
  if not _key.keyPressed(_key.keyCode) then
    put "KeyPressed("&_key.keyCode&") error"
  end if
end keyDown
```

Set the input keyboard layout for your computer to something other than a standard US English keyboard, launch your movie and press each key in turn. Watch the output in the Message window.

If you chose a French keyboard, for example, where the same Roman letters have a different layout, you will find that the `keyPressed()` function fails when it's used on *characters* that are not in the same place as on a US keyboard.

If you chose a keyboard that generates Unicode characters, such as Russian, then you will find that it fails on all characters and `keyCode`s, except for the key with a `keyCode` of `0`.

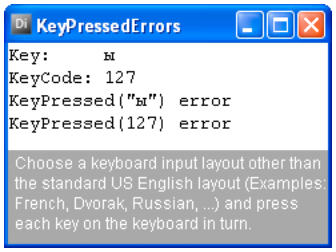
In a projector or in Shockwave, on Macintosh, `keyPressed(aKeyCode)` will succeed, but `keyPressed(aChar)` may fail.



keyPressed(aKeyCode) works correctly in Shockwave and projectors on Macintosh but keyPressed(aChar) may fail

Issues with keyPressed() on Windows

On Windows, `keyPressed()` will fail on Unicode input in all environments.



keyPressed(aKeyCode) fails with Unicode input in all environments

For the standard Roman characters used in US English, both `keyPressed(aChar)` and `keyPressed(aKeyCode)` will function correctly, regardless of the keyboard layout. This is because the value of `_key.keyCode` ignores the keyboard layout.

keyCode values for UTF-8 input

If the user's keyboard is set to input Unicode characters, rather than Roman characters, then Windows is unable to generate a usable value for `_key.keyCode`. On Windows, in Director 11.5, the value for `_key.keyCode` may not provide any usable values. In Shockwave and in a projector, the Director player will generate valid values for `_key.keyCode` on a Macintosh, but an unhelpful value of 127 for all keys on Windows.

Differences between the authoring environment and Shockwave

For Unicode input, in the authoring environment for Director 11.5, the value for `_key.keyCode` is consistently `0` on both Windows and Macintosh. This means that it is not possible to use `_key.keyCode` to distinguish any Unicode character from the Roman letter A.

Multiple simultaneous key presses

Depending on the operating system and the keyboard, Director can handle as many as six simultaneous key presses. Your computer may not behave the same way as your end-user's computer.

If you have complete control over the playback environment (for a kiosk, for example), then you can run a comprehensive test on which keys can be pressed simultaneously. If you are creating an application for general, cross-platform release, then you be very cautious about requiring the user to make multiple simultaneous key presses.

Before committing to a keyboard control system that requires multiple simultaneous key presses, try testing the KeyPressed.dir movie on a wide range of target machines. Verify whether or not the key combinations that you intend to use work on all the test computers.

Arrow Keys

Director can only inform you about the first two arrow keys that have been pressed. If the user presses three or more arrow keys simultaneously, your application may not receive any information about the third and fourth arrow key press, until one of the earlier keys is released.

Customizing control keys

Unless you know that all users of your 3D movie will have standard US English keyboards, allow the users to customize the keyboard controls.

On Windows, you cannot rely on keyPressed() being operational for users whose keyboards generate Unicode characters. On Macintosh, you cannot rely on `keyPressed(aChar)` returning the expected value for non US keyboards. See "Keyboard control" on page 254 for more details.

For an application, there are three safe strategies:

- Check `_key.commandDown`, _key.controlDown, _key.optionDown, or _key.shiftDown to see if the user is pressing a modifier key.
- Check the value of _key.keyCode to determine whether the user pressed an arrow key, a page key, or the delete or backspace key.
- Inside an `on keyDown()` handler, check the value of _key.key to determine which character appears on the key that the user just pressed. Such an action limits you to detecting a single character key at a time.

These three techniques will work cross-platform and with any keyboard layout. Note that pressing a modifier key will not trigger an `on keyDown()` handler. You may need to poll for the modifier keys in an `on enterFrame()` handler.

Note: *Typically, if you press a key and hold it down, the keyboard will send a first `#keyDown` event, then pause, and then send a regular stream of `#keyDown` events. Take this delay into account when designing the keyboard interactions.*

A screen for customizing controls

If you plan to allow the user to press character keys, then provide a screen where users can customize the keyboard controls. This screen can indicate the various actions that you have programmed into the game, and ask users to type the key that they want to use for that action.

You can store the `key`, `keyCode` and modifier data in a property list, to act as a look-up table. For example, after the user has visited the Customize Controls screen, your property list may look like this:

```
vControlsList = [126: #forward, "w": #forward, "W": #forward, 125: #back, "s": #back, "S":
#back, 124: #right, "d": #right, "D": #right, 123: #left, "a": #left, "A": #left, #Shift: #Run,
#control: #fire]
```

Property lists are case sensitive. If the user may be pressing the Shift key at the same time as a character key, you will need to include both characters if you wish to use.

Treating keyboard input

You can use an `on keyDown()` handler to detect when the user is pressing a character key, and an `on enterFrame()` handler to detect when the user is pressing an arrow key or a modifier key.

To see an example of this, using text display rather than real 3D actions, download and launch the movie CustomControlDemo.dir.

When it has keyboard focus, the behavior on the 3D sprite detects key presses and converts them into actions

The CustomControlDemo.dir movie contains an editable sprite. Before the 3D sprite can receive any keyboard events, the user must ensure that it has keyboard focus. See the Customizable Controls behavior on the 3D sprite for a suggestion on how to convert keyboard input into actions using a list of action keys.

Note: *The movie does not show you how to customize this list of action keys. It simply shows how to work with a customized list that has been created elsewhere.*

Motion

Movement in a 3D world can be controlled in several different ways.

- You can move 3D nodes around using Lingo or JavaScript code. See "Code-driven motion" on page 260.
- You can add a keyFrame motion to a model to move it through space. See "Keyframe animations" on page 274.
- You can add a bones motion to a model to change the shape of its internal mesh. See "Bones animations" on page 276.
- You can create animated particle emitters. See "Particles" on page 197.
- You can apply physical features, such as, mass and force, to models, and rely on a Physics member created by the Dynamiks xtra to control all the interactions in a deterministic way. See "Physics" on page 293 for details.

Code-driven motion

You can use Lingo or JavaScript code to place a node in 3D space, and to set its orientation and scale. See "Arranging objects in a 3D world" on page 202 for a description of the Lingo and JavaScript terms that you need to achieve this.

Types of motion

You may need to move nodes around in your 3D world in different ways. For example:

- Moving in a straight line. See "Linear motion" on page 261 for details.
- Moving between two known points. See "Interpolation" on page 264 for details.
- Following a pre-defined path, like a train. See "Following a path" on page 267 for more details.

Updating a node on every frame

To create the illusion that a node is moving smoothly, change the disposition of the node on a regular basis. This means using an event that is generated many times a second to trigger a handler or function that updates the transform of the node. You can achieve this result in the following ways:

- Using an event generated every frame. See "Using frame events wisely" on page 397.
- Using a `timeOut` object to call a handler or a function on a regular basis. See Create Timeout objects.
- Registering a script for a `#timeMS` event generated by the 3D cast member. See registerForEvent() for details.

For all the examples in this section, the technique adopted is to add a behavior, and a script or script instance to the `actorList` . By doing so, `#stepFrame` events for the behavior, script or script instance in question, are generated. This technique is not as efficient as using `#enterFrame` events, but it has four distinct advantages over the alternatives:

- It is simple to add any behavior, script or script instance to the `actorList`. Forwarding `#enterFrame` events to parent scripts and script instances requires additional steps.
- Adding a scripting object to the `actorList` adds the minimum additional overhead to the Director playback engine. When a `timeOut` object is created, it generates a whole range of events for its target object, many of which will not be required.
- This technique works equally well for non-3D features. Only 3D members can generate a `#timeMS` event.
- An `on stepFrame` handler can be triggered only when it is required.

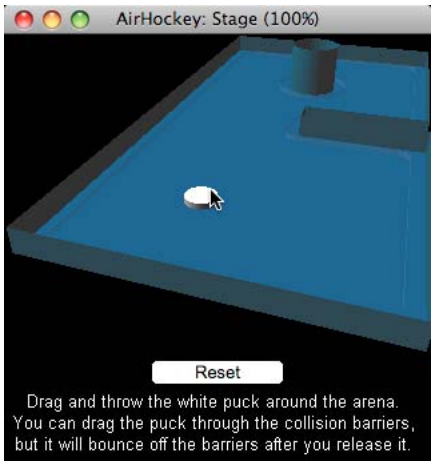
Linear motion

You can simulate the simple motion of an object in Director without the use of any extra extensions. From your high school physics classes you may remember the equations of linear motion. These equations relate distance, speed, time, and acceleration to one another.

- `distance = averageSpeed * time`
- `acceleration = (startSpeed - endSpeed) / time`
- `distance = startSpeed*time + acceleration*time*time / 2`

Using Lingo to simulate linear motion and friction

To see these equations implemented in Lingo, download and launch the movie AirHockey.dir.



Throw the air hockey puck and watch it slow down and stop

In the AirHockey.dir movie, drag the white puck and release the mouse button while the mouse is still moving. When you do so, the puck gains an initial speed and direction. The friction between the puck and the horizontal surface is simulated by an acceleration of `-0.001` world units per millisecond per millisecond. These three items — speed, direction and acceleration — are enough to calculate the trajectory of the puck.

In the AirHockey.dir movie, you will find a Movie Script named Linear Motion. This contains eight handlers. Four of these allow you to calculate the following information:

- How long it will take the puck to stop moving, due to friction
- How far the puck will have traveled after a given time
- How long it will take the puck to collide with a barrier
- At what speed the puck will be traveling when it collides with the barrier

To see how to detect a collision with the barrier, see "Rays" on page 285. To discover what happens when the puck collides with the barrier, see "Bouncing off a wall" on page 291. To see how to drag the puck around, see "Dragging" on page 250. This article deals with the movement of the puck between impacts.

Throwing the puck

How does the movie calculate the direction and speed with which you throw the puck? As you drag the puck around with the mouse, the Throw Model in Y Plane behavior in the AirHockey.dir movie calculates the latest position of the puck. It stores this position along with the current time in milliseconds in a property list. Here is a simplified version of the code.

```
property pPuck         -- model("Puck")
property pAcceleration -- -0.001
property plPositions   -- [<milliseconds>: <world position>, ...]
property pThrowTime    -- milliseconds when puck was released
property pThrowSpot    -- world position where puck was released
property pSpeed        -- speed of puck when it was released
property pDirection    -- direction of travel of puck on release
property pTimeToStop   -- milliseconds until puck stops moving
on mouseDown(me)
  pPuck = member("3D").model("Puck")
  pAcceleration = -0.001
  plPositions = [:]
   plPositions.sort() -- enables list.findPosNear()
  (the actorList).append(me)
end mouseDown
on stepFrame(me)
  -- ... other code to set the position of the puck
  plPositions.addProp(the milliseconds, pPuck.worldPosition)
end stepFrame
```

When you release the mouse button, the `mThrow()` handler is called. This works out where the puck was about 200 milliseconds before it was released, and where it was on the frame immediately before it was released. Below is a simplified version of the code.

```
vCount     = plPositions.count()
  vLast      = the milliseconds - 200 -- approximate time
  vIndex     = plPositions.findPosNear(vLast)
  pThrowTime = plPositions.getPropAt(vCount)
  vLast      = plPositions.getPropAt(vIndex) -- recorded time
  vElapsed   = pThrowTime - vLast
  vThen      = plPositions[vIndex]
  pThrowSpot = plPositions.getLast()
```

The variable `vThen` now contains the puck's position about 200 milliseconds ago, `pThrowSpot` contains the position where it was on the last frame, and `vElapsed` contains the time between these two positions. Subtracting one position from the other gives the displacement between the two positions.

The `mThrow()` handler calculates the speed and direction of the movement in two separate property variables, and also how long it will take the puck to come to a halt due to the effect of friction.

```
  pDirection  = pThrowSpot - vThen
  pSpeed      = pDirection.magnitude / vElapsed
  pDirection.normalize() -- unit vector
  pTimeToStop = -pSpeed / pAcceleration
```

Calculating the current position of the puck

After you release the puck, on every frame, the `mSlide()` handler calculates how much time has elapsed since the puck was released. It uses that information to determine where the puck is at the current point in time. Here is a simplified version of the handler. It incorporates the `getDistance()` handler from the Linear Motion script:

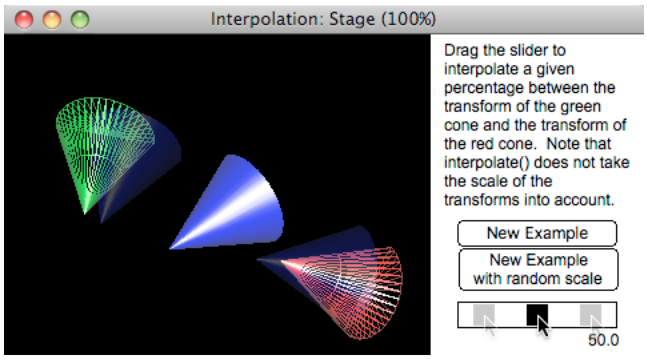
```
on mSlide(me)
  vElapsed = the milliseconds - pThrowTime
  if vElapsed > pTimeToStop then
    vElapsed = pTimeToStop
    (the actorList).deleteOne(me)
    -- This will be the last time the puck moves
  end if
  vDistance = (pSpeed + (pAcceleration*vElapsed)/2) * vElapsed
  vWorldPosition = pThrowSpot + pDirection * vDistance

  pPuck.worldPosition = vWorldPosition
end mSlide
```

The puck stops

Notice how the `mSlide()` handler checks whether the puck moment when the puck stops has passed. If it does not do this, the puck will come to a stop, and then start to accelerate in the opposite direction, as if friction behaved like an elastic band.

Interpolation

The interpolate() and interpolateTo() methods allow you to calculate a transform that is partway between one transform and another. To visualize what this means, download and launch the movie Interpolation.dir.



Dragging the slider will move the blue half-cone from the between the two other half-cones

Using interpolate()

The syntax for the interpolate method is:

```
outputTransform = transform1.interpolate(transform2, aPercentage)
```

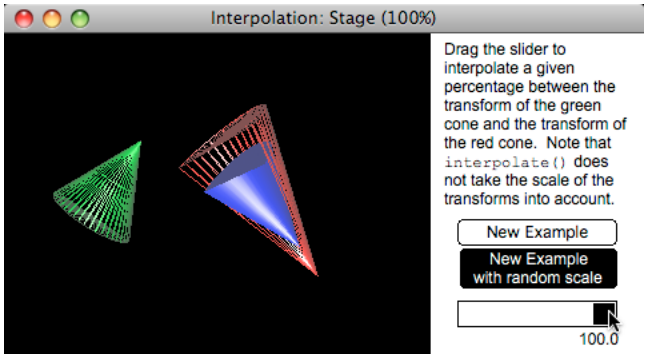
The value of `aPercentage` must be between `0.0` and `100.0`. If you use a value outside this range, a script error occurs.

The output is a transform, the scale of which will be `vector(1.0, 1.0, 1.0)`, regardless of the scales of the input transforms. The `position` and `rotation` properties of the output transform will be a weighted average of the `position` and `rotation` of the two other transforms.

If `aPercentage` is `0.0`, then the output is a transform whose `position` and `rotatation` are identical to that of `transform1`. If `aPercentage` is `100.0`, then the output is a transform whose `position` and `rotatation` are identical to that of `transform2`.

Neither `transform1` nor `transform 2` is modified by the `interpolate()` function.

To understand how `interpolate()` behaves when the input transforms have a scale other than 1, click on the New Example With Random Scale button.



The transform returned by interpolate() will always have a scale of 1

Try the following commands in the Message window:

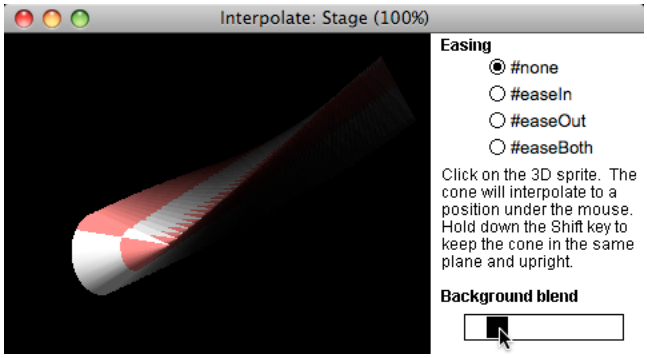
```
-- Lingo syntax
v3DMember = member("3D")
vBlueModel = v3DMember.model("Inter")
vStart = v3DMember.model("Start").transform
vEnd = v3DMember.model("End").transform
vBlueModel.transform  = vStart.interpolate(vEnd, 10.0)
// JavaScript syntax
v3DMember = member("3D");
vBlueModel = v3DMember.getPropRef("model", 2);
vStart = v3DMember.getPropRef("model", 1).transform;
vEnd = v3DMember.getPropRef("model", 3).transform;
vBlueModel.transform  = vStart.interpolate(vEnd, 10.0);
```

Examples

You can find examples of transform.interpolate() in the following movies:

- Interpolate.dir: See "Color buffer" on page 105 for details.
- WheelDemo.dir: See "Node hierarchy" on page 91 for details.
- KeyboardControl.dir: See "Moving to a given location" on page 225 for details.

These movies use a parent script named Interpolation. An instance of this script places itself on the `actorList` and performs the call to `interpolate()` once per frame, with the percentage value increasing from `0.0` to `100.0` over a given period of time.



The Interpolate.dir movie demonstrates the use of the Interpolate parent script to animate a model over time

Using interpolateTo()

The syntax for the interpolateTo() method is:

```
transform1.interpolateTo(transform2, aPercentage)
```

The value of `aPercentage` must be between `0.0` and `100.0`. If you use a value outside this range, a script error occurs.

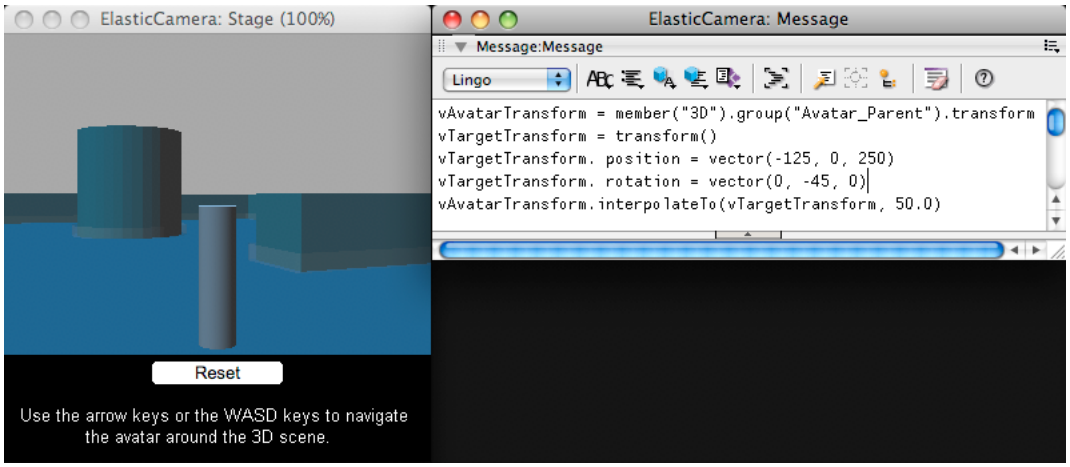
The `interpolateTo()` method modifies *`transform1`* by interpolating from the position and rotation of *`transform1`* to the `position` and `rotation` of *`transform2`* by the specified percentage. The original *`transform1`* is changed. Its `position` and `rotation` properties will be modified, but its `scale` will remain the same.

If `aPercentage` is `0.0`, then `transform1` will not be altered. If `aPercentage` is `100.0`, then the `transform1` will now have a `position` and `rotatation` that are identical to that of `transform2`.

Example uses

`InterpolateTo()` is useful if you want to move a node partway towards another one. For example, you can use `interpolateTo()` to give some elasticity to a third-person camera. Imagine that you attach a group named "RestPosition" as a child of the player's avatar, and place it in the ideal position for the camera when the avatar is standing still. On each frame, you can use `interpolateTo()` to move the camera 10% of the way from its current position towards the RestPosition group. When the avatar starts to run forward, the camera will lag a little behind. When the avatar slows down, the camera will ease back in towards the ideal rest position.

To see an example of this, download and launch the movie ElasticCamera.dir.



Using interpolateTo() gives a natural lag between the movement of the avatar and the movement of the camera

In the ElasticCamera.dir movie, take a look at the Elastic Third Person Camera script. The `on stepFrame()` handler includes a command similar to the following.

```
sprite(1).camera.transform.interpolateTo(aRestTransform, 10.0)
```

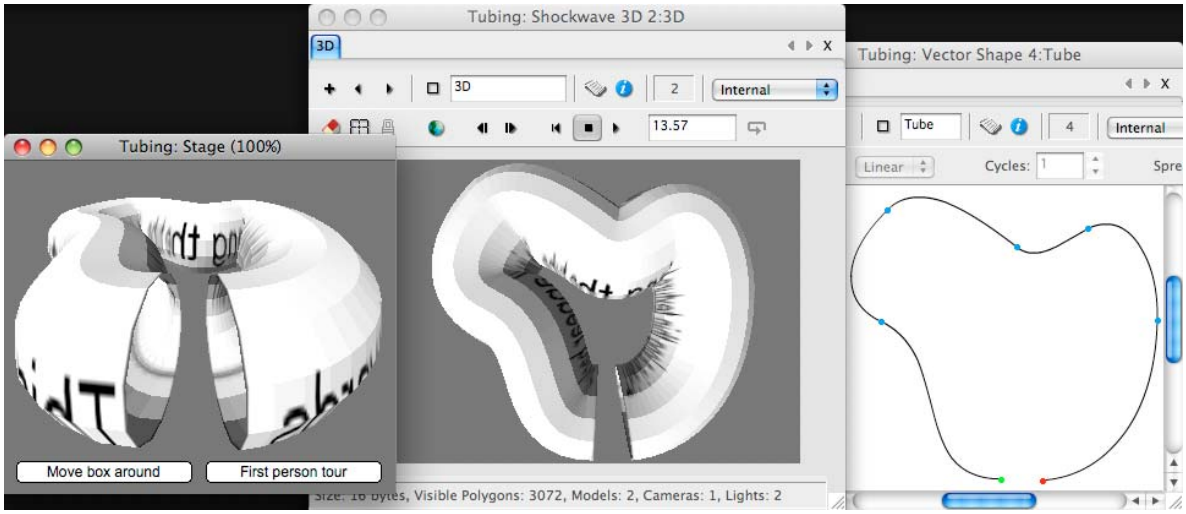
Launch the movie and the following commands in the Message window. You will observe that the avatar moves backwards and to the left, moving halfway towards the position `vector(-125, 0, 250)` and rotating halfway towards the rotation `vector(0, -45, 0)`. The camera will move smoothly after this point, using interpolateTo() several times on successive frames, until it comes to rest giving the view shown in the screenshot above.

```
-- Lingo
vAvatarTransform = member("3D").group("Avatar_Parent").transform
vTargetTransform = transform()
vTargetTransform. position = vector(-125, 0, 250)
vTargetTransform. rotation = vector(0, -45, 0)
vAvatarTransform.interpolateTo(vTargetTransform, 50.0)

// Javascript
vAvatarTransform = member("3D").getPropRef("group", 2).transform
vTargetTransform = transform();
vTargetTransform. position = vector(-125, 0, 250);
vTargetTransform. rotation = vector(0, -45, 0);
vAvatarTransform.interpolateTo(vTargetTransform, 50.0);
```

Following a path

You don't always want to move a node along a straight path. This article explains one way to move a node along a curved path. You can find an alternative solution at “Keyframe animations” on page 274.

To visualize the technique described here, download and launch the movie Tubing.dir.



The Tubing.dir movie creates a tube mesh resource from the curve defined by a vectorShape member

Creating a tube

The Tubing.dir movie uses a script named Bezier to Points to convert the smooth curve of a vectorShape member into a series of discrete points. When these points are joined together by straight lines, they give an approximation to the original curve.

The script named 3D Tube takes these points and generates a mesh resource from it. If you change the shape of the curve in the Tube vectorShape member and relaunch the movie, the tube that will be created will follow the curve that you created.

In the process of generating the tube, the 3D Tube script creates two lists:

- `pPointsList`: a list of the vector positions around which the tube is created
- `pRunningTotal`: a sorted list of distances from the beginning of the tube to the nth point in pPointsList. This list behaves like a tape measure that is fixed at the first point in `pPointsList`and pulled taut around all the points in `pPointsList`, measuring the distance to each point.

From pRunningTotal, it is easy to discover the total length of the path:

```
pTotalLength = pRunningTotal.getLast()
```

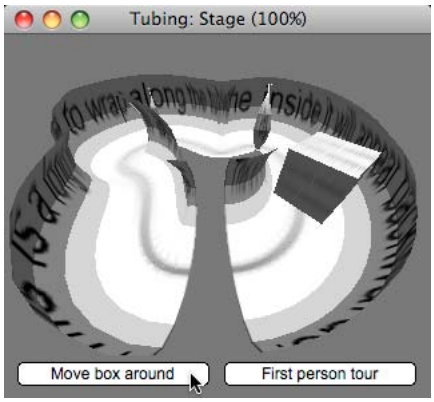
In your movies, you may not want to create a tube. All you need is the list of points, the cumulative list of distance, and the total length of the path. The tube just provides a visual cue about the shape of the path to follow.

Moving along the path

With the properties described in the previous section, you can calculate any position on the line. The Lingo handler `GetPosition()` accepts a floating point number between 0.0 (start of the path and 1.0 (end of the path) and returns a vector position along the line that joins all the dots.

```
property pPointList    -- list of n vector positions
property pRunningTotal -- sorted list of n-1 distances along path
property pTotalLength  -- final value in pRunningTotal
on GetPosition(aRatio)
  vDistance  = pTotalLength * aRatio

  vIndex     = pRunningTotal.findPosNear(vDistance)
  vLast      = pPointList[vIndex - 1]
  vNext      = pPointList[vIndex]
  vDirection = (vNext - vLast).getNormalized()

  vComplete  = pRunningTotal[vIndex - 1]
  vRemaining = vDistance - vComplete

  vPosition  = vLast + vDirection * vRemaining
  return vPosition
end GetPosition
```

Note the use of list.findPosNear() on a sorted list, to give the index of the next value in the sorted list, which is equal to or greater than the requested value.

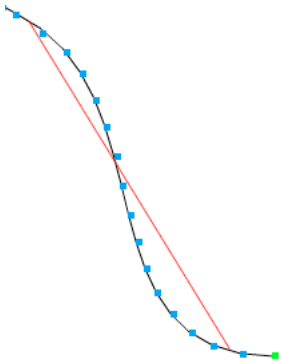


You can apply the position along the path to any kind of node, including cameras

Looking along the path

To get the node to turn to face along the path, you can simply ask for a position slightly further along the path, and use node.pointAt() to make the node face in that direction.

Choose carefully how far ahead to look. Ideally, the position must be on the line segment one or two points further along the path. If the point to look at is sometimes on the same segment, the node will alternate between rotating and not rotating, and the effect will not be natural. If it's too far ahead, the node may tend to point in the wrong direction.



Choose a pointAt position that is just a little more than one segment ahead (right)

Pre-defined animations

You can create animated movements in a third-party 3D design application and export them along with your 3D world as a W3D file.

The following are the two types of pre-defined animations in Director:

- A Keyframe animation is a time-based animation sequence. A keyframe motion stores the value for the transform of a node at key points along a timeline. When the animation is played back, the transform of the node is interpolated from one key frame to the next, over time. You can add the Keyframe player modifier at runtime to a model created in Director.
- A Bones animation modifies the model's geometry over time. Bones animations apply a sequence of movements to invisible bones a 3D model. As the bones move, so the mesh of the model changes shape. A bones animation can be used to create movements for a character, such as walking, running or standing idle. It can also be used to deform models in more dramatic ways. For example, you can use a bones animation to simulate the way the bodywork of a car crumples on impact.

  The individual movements within a Bones animation may be defined by keyframes. However, a Keyframe animation on its own will not change the shape of the model's mesh.

  Although it is possible to add the Bones player modifier at runtime, there is no reason to do so. The Bones player modifier is automatically attached to the bones of animated models exported from a 3D-modeling application. Since the required bones information cannot be assigned in Director, it has to exist before the model is imported into Director.

Like all modifiers, the bones and keyframe player modifiers can be attached only to models.

To play these animations back inside your Director movie, you will need to use either the `#keyframePlayer` modifier or the `#bonesPlayer` modifier. You can control the playback of pre-defined animations through scripting.

To use a keyframe animation on a camera or a light, attach the keyframe modifier to an invisible model, and make the camera or the light a child of the model.

You can combine the two types of animation. You can, for example, combine a "run in place" bones animation with a "move around the room" keyframe animation.

You cannot create new animations within Director. You can however create new motions to map an existing animation to specific bones in an animated model. However, in Director 11.5, there is no method for obtaining the names of the bones within an animated model. See Motions for more details.

Motions

A 3D cast member can contain a set of motions authored in your 3D-modeling application. Motions are either of the type `#keyframe` or `#bones`. A motion with a type of `#none` can act as an inactive placeholder. To play back a motion, use the appropriate modifier.

Play list

The keyframe and bones players manage a queue of motions. The first motion in the play list is the motion that is currently playing or paused. When that motion finishes playing, it is removed from the play list and the next motion begins.

Motions on the playList can be blended into each other as they are playing, in which case they will play simultaneously. See "Motion blending" on page 272 for more details.

Adding a motion to the playList

You can use two commands to add a motion to the playList.

- *model.motionsPlayer*`.queue()`
- *model.motionsPlayer*`.play()`

Both `queue()` and `play()` accept a series of parameters which allow you to determine where the motion starts playing, when it ends, whether it loops, and how quickly it plays. See the Scripting Dictionary entries for details.

The following example adds two motions to the playList for the model named "Bronco":

```
-- Lingo syntax
vModel = member("3D").model("Bronco")
vModel.bonesPlayer.queue("Buck")
vModel.bonesPlayer.play("Rear")
```

The `queue()` command places the named motion at the end of the playList. It will start to play when all the motions already in the queue have finished playing.

The `play()` command places the named motion at the beginning of the playList and starts to play it. The motion that was previously playing is stopped, and pushed back to the second position in the playList.

In the example given above, the Buck motion was placed first in the queue, and then immediately replaced in the first position by the Rear motion. As a result, the motions play in the order Rear, then Buck. If `autoblend` were `FALSE`, the `playList` looks like this (edited for clarity):

```
time    0: [[#name:"Buck", #endTime:813]]
time    1: [[#name:"Rear", #endTime:446], [#name:"Buck", #endTime:813]]
-- The motion Rear now plays
time  447: [[#name:"Buck", #endTime:813]]
-- The motion Rear is finished, so Buck can now play
time 1260: []
-- The motion Buck is now also finished
```

Motions are removed from the play list automatically when they are complete.

Removing a motion from the playList

You can use two commands to remove a motion from the `playList`

- model.motionsPlayer.playNext() removes the motion that is currently playing, and starts playing the next motion in the `playList`.
- *`model.motionsPlayer`*`.deleteLast()` removes the motion that is at the end of the playList. If there is only one motion on the `playList`, it will stop that motion as it is playing.

You can only remove motions from the beginning or the end of the playList. If you need to remove a motion from the middle, you will need to use `deleteLast()` multiple times until the unwanted motion has been removed. You will then have to use `queue()` to replace the desired motions that you deleted earlier.

Motion blending

If model.motionsPlayer.autoBlend is TRUE, an ending motion blends smoothly into the next motion using the model.motionsPlayer.blendTime property to determine how long the blend must be. You can control this manually by setting `autoBlend` to `FALSE` and using model.motionsPlayer.blendFactor to control the blend frame by frame.

Animation properties

The `#keyframePlayer` and `#bonesPlayer` modifiers share many properties and methods.

- model.motionPlayer.playing() indicates whether a model is executing a motion.
- model.motionPlayer.playlist is a linear list of property lists containing the playback parameters of the motions that are queued for a model.
- model.motionPlayer.currentTime indicates the local time, in milliseconds, of the currently playing or paused motion.
- model.motionPlayer.playRate is a number that is multiplied by the scale parameter of the play() or queue() command to determine the playback speed of the motion.
- model.motionPlayer.playlist.count returns the number of motions currently queued in the playlist.
- model.motionPlayer.currentLoopState indicates whether the motion plays once or repeats continuously.
- model.motionPlayer.autoBlend indicates whether the modifier creates a linear transition to the currently playing motion from the motion that preceded it.
- model.motionPlayer.blendTime indicates the length of the transition created by the modifier between motions when the modifier's autoblend property is set to TRUE.
- model.motionPlayer.blendFactor indicates the degree of blending between motions when the modifier's autoBlend property is set to FALSE.
- model.motionPlayer.positionReset indicates whether the model returns to its starting position after the end of a motion or each iteration of a loop.
- model.motionPlayer.rotationReset indicates the rotational element of a transition from one motion to the next, or the looping of a single motion.
- model.motionPlayer.rootLock indicates whether the translational component of the motion is used or ignored.
- model.motionPlayer.lockTranslation indicates whether the model can be displaced from the specified planes.

Animation Methods

- model.motionPlayer.pause() halts the motion currently being executed by the model.

- model.motionPlayer.play() initiates or unpauses the execution of a motion.
- model.motionPlayer.playNext() initiates playback of the next motion in the playlist.
- model.motionPlayer.queue() adds a motion to the end of the playlist.
- model.motionPlayer.removeLast() removes the last motion from the playlist

Animation Events

You can tell Director to call a given handler in a given script or script instance whenever an animation starts or ends. You can use two different methods to do this:

- member3D.registerForEvent()
- model.registerScript()

The `member3D.registerForEvent()` method calls the given handler for all animation events on all models in a given 3D member. The `model.registerScript()` calls the given handler only for animation events on the given model.

Both methods require the following parameters:

- `eventName`: either `#animationStarted` or `#animationEnded`
- `handlerName`: symbol handler name for callback
- `scriptObject`: script or script instance containing the given handler

Note: *Director makes no attempt to check whether you used a valid eventName or whether the given script object does indeed contain the named handler.*

`#animationStarted` is sent when a motion begins playing. If blending is used between motions, the event is sent as soon as the transition begins.

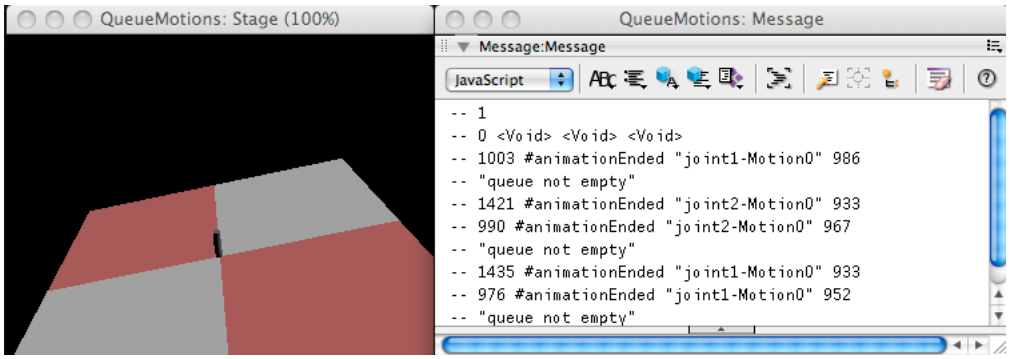
`#animationEnded` is sent when a motion ends. If blending is used between motions, the event is sent when the transition ends.

Each time one motion replaces another `#animationStarted` is sent for the new motion before `#animationEnded` is sent for the previous one. When `autoBlend` is `TRUE`, there may be some delay between the next motion starting and the previous motion ending.

Director will send three arguments to the declared handler includes three arguments:

- The event type: either `#animationStarted` or `#animationEnded`
- The name of the motion
- The current time of the motion in milliseconds

To see an example of using events to queue motions in the playList, download and launch the movie QueueMotions.dir.



An #animationEnded event will trigger the on StartQueue() handler and refill the motion playList

See "Events" on page 355 for information on the other events that can be generated by a 3D member. See "Animation event callback" on page 358 for examples of animation callback handlers.

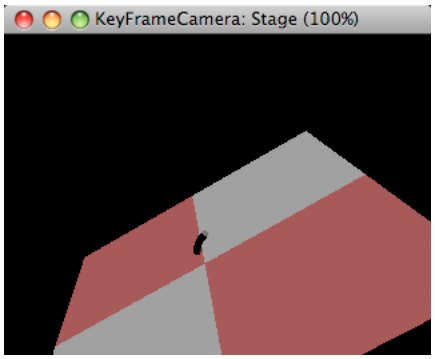
Keyframe animations

The keyframePlayer modifier shares all its properties, methods and events with the bonesPlayer modifier. See "Pre-defined animations" on page 270 for details.

Unlike the Bones animations, keyframe animations can be applied to any model. Keyframe animations do not require any particular internal structure in the models they are applied to. To add the keyframePlayer modifier to a model at runtime, use model.addModifier().

Note: *It is not possible to add modifiers to cameras, groups or lights. However, it is possible to add the keyframePlayer modifier to an invisible model, and to add a camera, group or light as the child of the invisible model.*

To see an example of this, download and launch the movie KeyFrameCamera.dir.



The KeyFrameCamera.dir movie animates a camera by attaching it as a child to an invisible model

Adding the keyframePlayer to an invisible model

The following code creates a model using the 1x1 unit DefaultPlane resource, and then makes the model invisible before attaching the keyframePlayer modifier to it:

```
-- Lingo syntax
v3DSprite = sprite("3D")
v3DMember = v3DSprite.member
vResource = v3DMember.modelResource(1)
vModel = v3DMember.newModel("Camera_parent", vResource)
vModel.visibility = #none
vModel.addModifier(#keyframePlayer)
vModel.addChild(v3DSprite.camera, #preserveParent)
// JavaScript syntax
v3DSprite = v3DSprite("3D");
v3DMember = v3DSprite.member(;
vResource = v3DMember.getPropRef("modelResource", 1);
vModel = v3DMember.newModel("Camera_parent", vResource);
vModel.visibility = symbol("none");
vModel.addModifier(symbol("keyframePlayer"));
vModel.addChild(v3DSprite.camera, symbol("preserveParent"));
```

You can use a similar technique to add the keyframePlayer modifier to visible models.

Playing a motion

The following code adds to the model a custom motion that was created in a third-party 3D design application and exported with the current 3D world:

```
-- Lingo syntax
vMotion     = v3DMember.motion(4)
vMotionName = vMotion.name
vModel.keyframePlayer.play(vMotionName, TRUE)
//JavaScript syntax
vModel     = v3DMember.getPropRef("model", 10);
vMotionName = vMotion.name;
vModel.getPropRef("keyframePlayer", 1).play(vMotionName, true);
```

This code sets the motion to loop.

Pausing playback of motions

The following code adds to the model a custom motion that was created in a third-party 3D design application and exported with the current 3D world.

```
-- Lingo syntax
vModel.keyframePlayer.pause()
//JavaScript syntax
vModel.getPropRef("keyframePlayer", 1).pause();
```

The code sets the motion to loop.

Removing all motions from a model

To remove a motion from the playList before it is played, use `removeLast()`.

```
-- Lingo syntax
ii = vModel.keyframePlayer.playList.count
repeat while ii
  vModel.keyframePlayer.removeLast()
  ii = ii - 1
end repeat
// JavaScript syntax
vCount = vModel.getPropRef("keyframePlayer", 1).playList.count;
for (ii=0;ii<vCount;ii++) {
  vModel.getPropRef("keyframePlayer", 1).removeLast();
}
```

Looping a motion between custom limits

The following code below will add `member("3D").motion(4)` to the `playList` of `vModel`:

```
-- Lingo syntax
vLooped = TRUE
vStartTime = 500
vEndTime = 600
vRate = 0.1
vOffset = 250
vModel.keyframePlayer.queue(vMotionName, vLooped, vStartTime, vEndTime, vRate, vOffset)
// JavaScript syntax
vLooped = true;
vStartTime = 500;
vEndTime = 600;
vRate = 0.1;
vOffset = 250;
vModel.getPropRef("keyframePlayer", 1).queue(vMotionName, vLooped, vStartTime, vEndTime,
vRate, vOffset);
```

When it is this motion's turn to play, it will start playing 250 milliseconds from the beginning, then play up to millisecond 600, and then continue to cycle between 500 and 600. It will play at one-tenth of its normal speed.

For information on blending motions, see autoBlend, blendTime, and blendFactor.

Bones animations

The bonesPlayer modifier shares many of its properties, methods and events with the keyframePlayer modifier. See "Pre-defined animations" on page 270 for details.

Bones animations require a particular internal structure in the models they are applied to. If you attempt to apply a bones animation to a model which does not have the correct bone structure, it will fail silently.

Bones animations cannot be created in Director. You must use a third-party 3D design application, then export the 3D world and the animations as a W3D file. Creating bones animation in a 3D modeling application can be complex, but it results in more natural-looking movements.

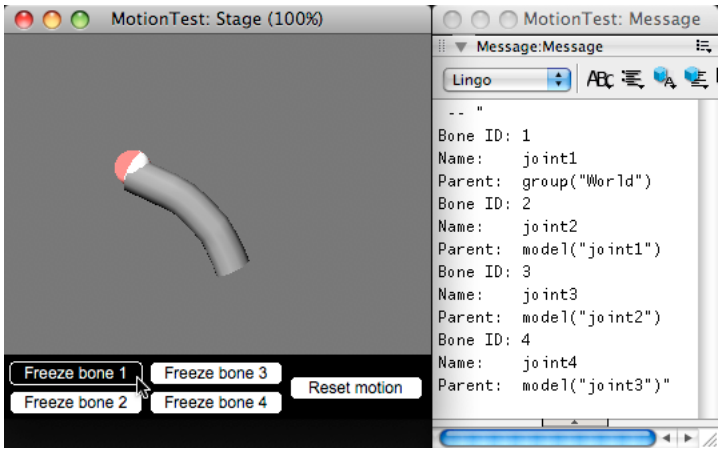
For bones animation, each motion contains a list of tracks, and each track contains the keyframes for a particular bone. A bone is a segment of the skeleton of a model. For example, track 14 of the motion named Run can be named RtKneeTrack and move a bone named RtKnee. These names are defined in the 3D modeling application.

Bones

When you import a W3D file containing bones animation, each bone may appear as a separate model. To discover which models are the bones of a particular modelResource, you can use modelResource.getBoneID(), using the name of each model in turn. If the result is a non-zero integer, the named model is a bone in the modelResource.

You can work out the parent-child hierarchy of bones by checking the parent of each of the bone models.

To see an example of this, download and launch the movie MotionTest.dir. Watch the output in the Message window as the movie launches.



The MotionTest.dir movie contains a very basic bones animation

Bone properties

- `model.bonesPlayer.bone[boneId].transform` indicates the transform of the bone relative to the parent bone. You can find the `boneId` value from the name of the bone by using the modelResource.getBoneID() method of the model resource.

  You can read the properties of a bone's transform, but you cannot set any of its properties directly. You can however replace the bone's transform with a standard transform which you can then control.

  When you set the transform of a bone, it is no longer controlled by the current motion, and cannot be returned to the control of the motion. To end manual control, you can use model.bonesPlayer.playNext() to remove the bone from the playList. Manual control will end automatically when the current motion ends. You can experiment with this in the MotionTest.dir movie

- `model.bonesPlayer.bone[boneId].worldTransform` returns the world-relative transform of the bone. This property gives you a means to attach a model to a bone. For example, on every `exitFrame`, you can set the transform of a Hat model to the `worldTransform` of a character's Head bone.

  See the MotionTest.dir movie for an example.

Motion mapping

You can create new motions by combining existing motions. For example, a walking motion can be combined with a shooting motion to produce a walk-and-shoot motion. This is available only with Bones player animations.

See motion.map() for more details.

update()

You can use model.update() to force a model with bones animation to update without having to wait for the screen to be rendered. This will ensure that the correct position data is available to your scripts.

Playing a motion

The following code adds to the model a custom motion that was created in a third-party 3D design application and exported with the current 3D world:

```
-- Lingo syntax
vMotion     = v3DMember.motion(4)
vMotionName = vMotion.name
vModel.bonesPlayer.play(vMotionName, TRUE)
//JavaScript syntax
vModel     = v3DMember.getPropRef("model", 10);
vMotionName = vMotion.name;
vModel.getPropRef("bonesPlayer", 1).play(vMotionName, true);
```

This code sets the motion to loop.

Pausing playback of motions

The following code adds to the model a custom motion that was created in a third-party 3D design application and exported with the current 3D world:

```
-- Lingo syntax
vModel.bonesPlayer.pause()
//JavaScript syntax
vModel.getPropRef("bonesPlayer", 1).pause();
```

This code sets the motion to loop.

Removing all motions from a model

To remove a motion from the playList before it is played, use `removeLast()`.

```
-- Lingo syntax
ii = vModel.bonesPlayer.playList.count
repeat while ii
  vModel.bonesPlayer.removeLast()
  ii = ii - 1
end repeat
// JavaScript syntax
vCount = vModel.getPropRef("bonesPlayer", 1).playList.count;
for (ii=0;ii<vCount;ii++) {
  vModel.getPropRef("bonesPlayer", 1).removeLast();
}
```

Looping a motion between custom limits

The following code adds `member("3D").motion(4)` to the playList of vModel:

```
-- Lingo syntax
vLooped = TRUE
vStartTime = 500
vEndTime = 600
vRate = 0.1
vOffset = 250
vModel.bonesPlayer.queue(vMotionName, vLooped, vStartTime, vEndTime, vRate, vOffset)
// JavaScript syntax
vLooped = true;
vStartTime = 500;
vEndTime = 600;
vRate = 0.1;
vOffset = 250;
vModel.getPropRef("bonesPlayer", 1).queue(vMotionName, vLooped, vStartTime, vEndTime, vRate,
vOffset);
```

When it is this motion's turn to play, it will start playing 250 milliseconds from the beginning, then play up to millisecond 600, and then continue to cycle between 500 and 600. It will play at one-tenth of its normal speed.

For information on blending motions, see autoBlend, blendTime, and blendFactor.

Collisions

All that you can see in a virtual 3D world are pixels. The models are just the projection of abstract objects onto your screen. To give these pixels the illusion of solidity, you must prevent them from appearing to intersect with each other.

Collision detection requires testing on each frame whether each moving model has attempted to pass through every solid model in the scene. In a busy scene, this can require very intensive use of the computer processor. To improve performance, it is important to limit the number of calculations to those situations where a collision is likely.

Detecting collisions

Director provides a number of ways of detecting collisions between models.

- The `#collision` modifier manages the detection and resolution of collisions. It was introduced in Director 8.5 and has not evolved since. It can handle simple collisions that occur between models that are moving relatively slowly compared to their size. See “Collision modifier” on page 280.
- The simplest way to prevent two models from colliding with each other is to ensure that they never get close enough to each to touch. See “Custom collision detection” on page 283 for a discussion of some fast techniques.
- When only one model is moving in an otherwise static world, you can cast a ray along the direction of motion of the moving model to see whether there is another object in its path. You only need to cast a ray when the moving model changes direction. See “Rays” on page 285 for more details.
- The Ageia Physics Xtra is a high-performance tool that helps developers create 3D worlds in which objects interact. The Xtra performs calculations to determine the results of collisions, factoring in object properties such as mass, velocity, and rotation. See “Physics” on page 293 for more details.

Resolving collisions

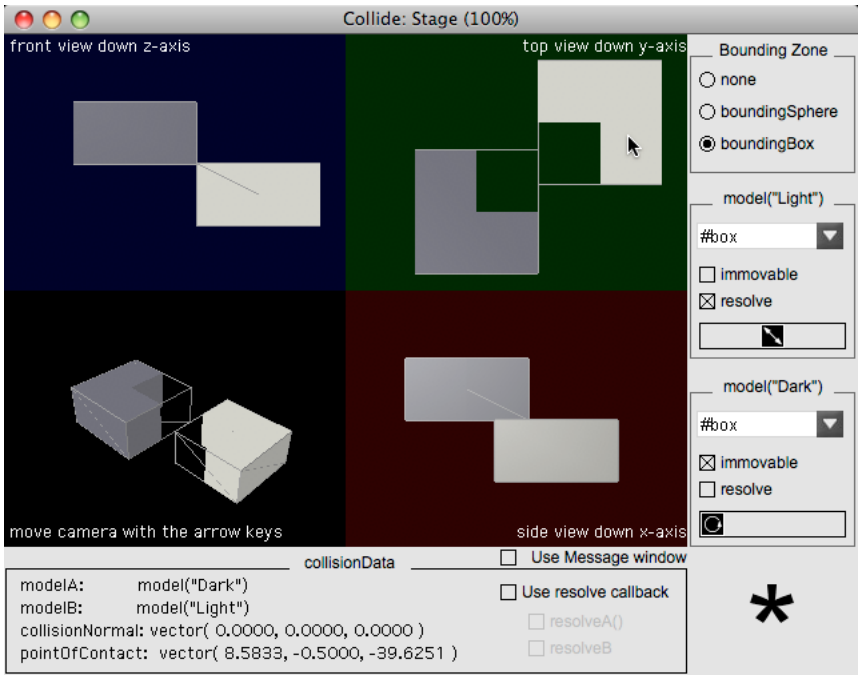
When a collision is detected, you decide what the result of the collision will be. Different situations require different solutions. Here are some examples.

- An avatar controlled by the user will simply be prevented from passing through a solid object. You may want to deflect the movement of the avatar so that it moves along the obstacle. See "Sliding along a wall" on page 237 for more details.
- An avatar that moves over a terrain needs to remain at the surface of the terrain at each step. See "Hugging Terrain" on page 238 for more details.
- A ball that collides with a wall will rebound and start traveling in a new direction. The impact with the wall may reduce its speed. See "Bouncing off a wall" on page 291 for more details.
- A ball that encounters a sloping surface may roll down the surface. See "Terrains" on page 319 for more details.

Collision modifier

When you attach the collision modifier to models, it will generate an event when the relative movement of two models may result in a collision. It gives you the option of stopping the movement of the models at the point near where the collision would occur.

To test the collision modifier, download and launch the movie Collide.dir.



Set the collision modifier properties and drag the models around. An asterisk appears when a collision is detected.

Collide.dir allows you to set the properties of the collision modifier and visualize collision data.

Adding the collision modifier to a model

The following command adds the `#collision` modifier to model 2, named "HighImpact", of the member "3D":

```
-- Lingo syntax
member("3D").model("HighImpact").addModifier(#collision)
put member("3D").model("HighImpact").modifier
 -- [#collision]
// JavaScript syntax
member("3D").getPropRef("model", 2).addModifier(symbol("collision"));
trace(member("3D").getPropRef("model", 2).modifier)
// <[#collision]>
```

Removing the modifier

You can use the command aModel.removeModifier() to remove the `#collision` modifier.

```
-- Lingo syntax
member("3D").model("HighImpact").removeModifier(#collision)
// JavaScript syntax
member("3D").getPropRef("model", 2).removeModifier(symbol("collision"));
```

Collision properties

Adding the collision modifier to a model gives you access to four properties:

- model.collision.enabled: `TRUE` or `FALSE` (`TRUE` by default). If `TRUE`, then collisions between this model and other models will trigger a collision event.
- model.collision.mode: `#sphere`, `#box` or `#mesh`. Determines what form of simplified geometry is used for collision detection. The default `#sphere` mode is the fastest and `#mesh` the slowest.
- model.collision.resolve: `TRUE` or `FALSE` (`TRUE` by default). If `TRUE` and you attempt to move this model to a position where a collision is detected with another model, then this model will be moved back in a straight line along its trajectory to a position where no collision is detected.
- model.collision.immovable: `TRUE` or `FALSE` (`FALSE` by default). Setting this property to `TRUE` will not prevent you from moving the model. It simply tells the collision modifier that any movement of this model is to be ignored. To improve performance, the collision modifier will not check whether any immovable models are in collision with each other.

Collision events

You can generate a callback when a collision occurs using any of the following methods:

- model.collision.setCollisionCallBack()
- model.registerScript()
- member3D.registerForEvent()

The `setCollisionCallBack(aHandler, aCodeObject)` method is a shortcut for *model*`.registerScript(#collideWith,aHandler, aCodeObject)`. You can use `registerForEvent(#collideAny, aHandler, aCodeObject)` to generate a callback whenever any collision is detected between any two models with the `collision.modifier`.

collisionData

When a collision is detected, the collision modifier will call any handler that is registered for collision events and send a collisionData object as a parameter to the call. The collisionData object has four properties.

- collisionData.modelA is one of the models involved in the collision.

- collisionData.modelB is the other model involved in the collision.
- collisionData.pointOfContact is the world position of the collision.
- collisionData.collisionNormal is the direction of the collision.

Note: *In the Collide.dir movie, this data is displayed below the 3D sprites.*

The collisionData object also has two methods:

- collisionData.resolveA() is the direction of the collision.
- collisionData.resolveB() is the direction of the collision.

You can call either of these methods with a parameter of TRUE or FALSE, to override the current `model.collision.resolve` property for the model in question.

In the Collide.dir movie, you can select whether these calls are made by toggling the Use Resolve Callback check box.

Testing with Collide.dir

The Collide.dir movie does not display the `collision.enabled` property. Instead, the collision modifier is removed if you choose `none` or `#Lingo sphere` as the `collision.mode`. You can visualize the `boundingSphere` for the both models, or show a bounding box, to help see why a collision is detected with the different modes. Note that the bounding spheres may not seem to touch when a collision is detected. This is because only the vertex points shown by the sphere wire frame are actually on the surface of the sphere. The wires themselves take a shortcut across the interior of the sphere.

If you set the `collision.immovable` property for both models to `TRUE`, then no collisions are detected between them.

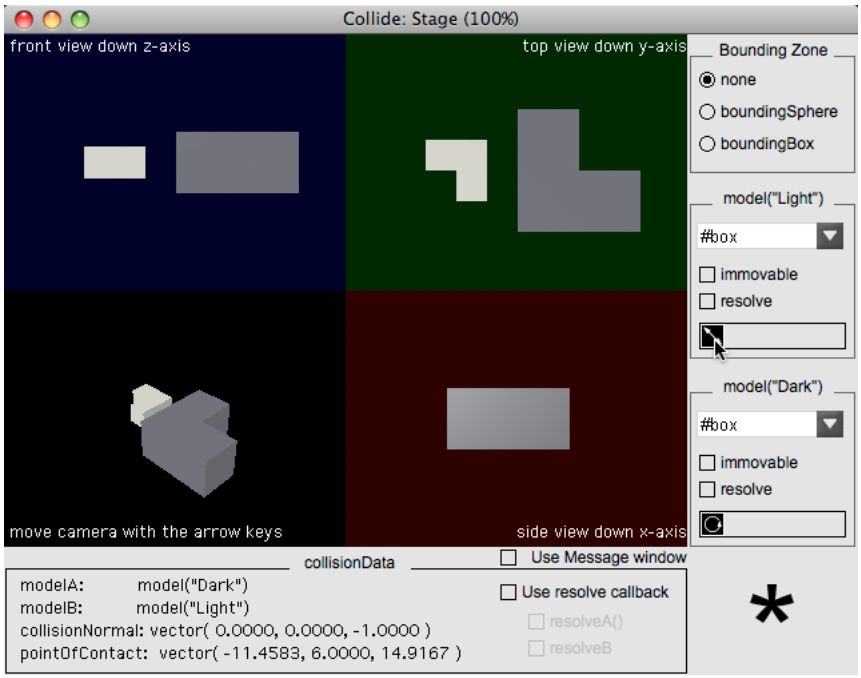
If you put the two models in a collision state, and then set `collision.immovable` to `FALSE` and `collision.resolve` to `TRUE` for one of the models, then you will be unable to move that model.

Bugs in the collision modifier

The collision modifier does not always report the `collision.normal` value accurately. To check this, select the Use Message Window check box. When you do so, the `collisionNormal` and `pointOfContact` details are shown in the Message window. Notice how the `collisionNormal` value tends to vary unpredictably, and how it tends to align itself with the world's axes.

The `#mesh` value for collision.mode is supposed to work on the actual mesh geometry of the model's resource, but this is apparently not the case.

You can use a slider to scale the Light model. (Double-click on the slider to reset the scale to 1). Notice that when you make the Light model bigger, the collision modifier behaves as if you made it smaller and vice versa.



When models are scaled, collisions may be reported incorrectly

Note: *There are also other bugs which the Collide.dir movie is not designed to illustrate. These include issues with fast-moving models, the detection of shallow collisions, and the ability to correctly resolve collisions. In many cases, you will find that there are better solutions than relying on the collision modifier.*

Custom collision detection

In some cases, you can perform collision detection very simply, without the need for any modifiers or xtra extensions. This article will look at one of these cases. The purpose is to help you to visualize the process of collision detection and how a collision is resolved.

Collision between two spheres

The simplest case is the detection of a collision between two spheres. Every node has a node.boundingSphere property, which provides the center and the radius of the smallest sphere that will fit snugly around all the vertices in the node and in its children. The coordinates and dimensions are given with reference to the world. If a spherical model has no children, then its `boundingSphere` gives a perfect description of its own volume.

Here is a handler that will tell whether the `boundingSphere` of one model intersects that of another.

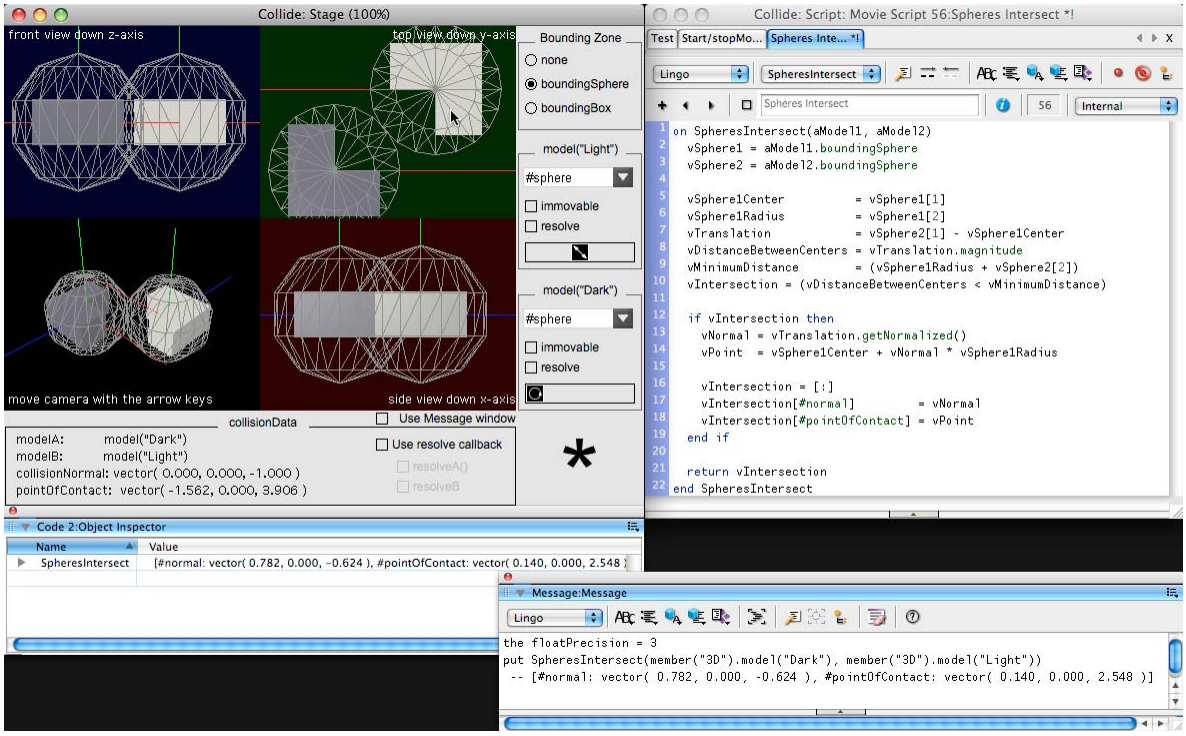
```
on SpheresIntersect(aModel1, aModel2)
  vSphere1 = aModel1.boundingSphere
  vSphere2 = aModel2.boundingSphere
  vDistanceBetweenCenters = (vSphere2[1] - vSphere1[1]).magnitude
  vMinimumDistance  =  vSphere1[2] + vSphere2[2]
  vIntersection = (vDistanceBetweenCenters<vMinimumDistance)
  return vIntersection
end SpheresIntersect
```

To test this handler, you can download the movie Collide.dir. Add a Movie Script in one of the lower-numbered cast slots, and give it the scriptText above. Now launch the movie and watch the expression below in the Object Inspector.

```
SpheresIntersect(member("3D").model("Dark"), member("3D").model("Light"))
```

If you set the movie up to use the `#sphere` mode of collision detection with the collision modifier and to show the `boundingSphere` for both models, you will be able to check that this script has the same level of accuracy as the collision modifier.

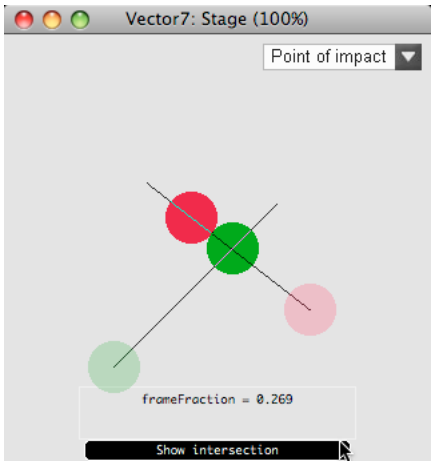
If you delete your script, the Spheres Intersect script in slot 56 will do the same job, and also provide more feedback on the intersection if there is one.



A simple Lingo script can be more accurate than the collision modifier

Resolving a collision between two spheres

After a collision has been detected between two spheres moving relative to each other, you need to work out where the initial point of impact had been. If both spheres are moving at a constant speed, then you need to solve a quadratic equation. To see this done for you, download and launch the movie Vector7.dir. Use the popup menu in the top right corner to jump to the Point Of Impact marker.



Finding the point in space and time where two moving spheres will collide

The solid colored circles indicate the starting position of the spheres. The lighter colored circles indicate the position where the spheres will be at the end of the frame if they did not collide. Click on Show Intersection to see where the spheres will first come into contact with each other.

The code to calculate this is the `FindImpact()` handler in the Impact behavior. The code that solves the quadratic is in the Quadratics movie script.

Rays

Director gives you five techniques for sending a ray into a 3D world, like a laser pointer, to detect what objects can be found in a given direction from a given point.

- camera.modelUnderLoc()
- camera.modelsUnderLoc()
- member3D.modelsUnderRay()
- physics.rayCastClosest()
- physics.rayCastAll()

The two camera methods (`modelUnderLoc()` and `camera.modelsUnderLoc()`) are explained in detail at "Picking" on page 242. The two physics methods (`rayCastClosest()` and `physics.rayCastAll()`) are explained in more detail in "Ray casting" on page 321.

modelsUnderRay()

This function sends a ray from a given point in the 3D world in a given direction. It returns VOID or a list of models encountered along the way. The method allows a number of different parameters. For more details, see member3D.modelsUnderRay(). The syntax discussed here is:

```
vData = member3D.modelsUnderRay(aPoint, aDirection, aOptionsList)
```

The options list must be a property list which can have the following properties and values:

```
[#maxNumberOfModels: <integer>,
 #maxDistance:       <positive float>,
 #levelOfDetail:     <#simple | #detailed>,
 #modelList:         [<model>, ...]]
```

For example:

```
[#maxNumberOfModels: 1, #levelOfDetail: #detailed, #modelList:
[member("3D").model("Target")]]
```

When used with this syntax and the value `#detailed` for the `#levelOfDetail` property, the output from the function is a list of property lists, with the format:

```
[[#model:         <model under the given loc>,
  #distance:      <float distance to intersection point>,
  #isectPosition: <vector worldPosition of intersection point>,
  #isectNormal:   <vector normal of the face at intersection>,
  #meshID:        <integer id of mesh to which the face belongs>,
  #faceID:        <integer id of intersected face>,
  #vertices:     [<vector>, <vector>, <vector>],
  #uvCoord:      [#u: <float>, #v: <float>]], ...]
```

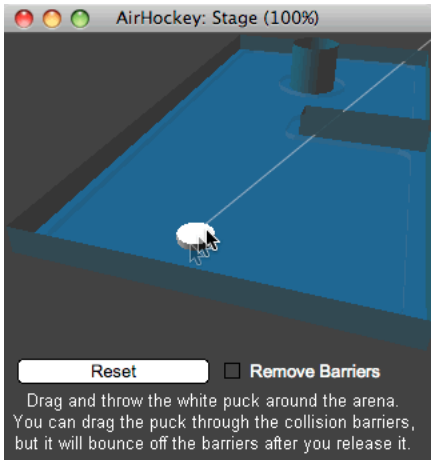
The `#uvCoord` property gives you information about which pixel in the texture image was touched by the ray. See "Mapping a texture to a mesh resource" on page 150 for more details.

Detecting obstacles in the direction of motion

Imagine a world with one moving object: the player's avatar. Imagine an obstacle some distance away from the avatar. Imagine that the avatar starts moving and keeps traveling in the same direction. In this case, you only need to send out one ray to know how far the avatar will need to travel before colliding with the obstacle. If you know the speed of the avatar, you can calculate the time when the impact will occur.

The concept of using a ray to detect collisions for an avatar steered by the user is explained in detail in "Not walking through objects" on page 222. This article treats the simpler process required to deal with the movement of a model traveling in a straight line.

To see a very simple game that illustrates the idea, download and launch the movie AirHockey.dir.



The direction of movement of the mouse defines the direction of the invisible ray

The Air Hockey behavior creates a white model named "Puck". You can drag the Puck model around. (There is no collision detection while you are dragging it). Release the mouse button while you are dragging the puck to throw it inside the blue tray area. If your throw is successful, the puck will collide with the walls and bounce back.

To see how to drag the puck around, see "Dragging" on page 250. To see how to make the puck move after it has been released, see "Linear motion" on page 261. To discover what happens when the puck collides with the barrier, see "Bouncing off a wall" on page 291.

Using modelsUnderRay()

In the AirHockey.dir movie, the movement of the puck is controlled by the Throw Model in Y Plane behavior. When you throw the Puck model, the `mThrow()` handler in that behavior calculates the current position of the Puck and its direction of motion. See Linear motion for details.

Below is a simple handler that takes this information, in the form of the `aPosition` and `aDirection` parameter and returns information about the point where the Puck will collide with one of the barriers. The handler also needs information on which 3D member contains the world (`aMember`), and which model is used for the barrier (`aBarrierModel`).

```
on GetImpactData(aMember, aPosition, aDirection, aBarrierModel)
  vRayInfo                      = [:]
  vRayInfo[#maxNumberOfModels] = 1
  vRayInfo[#levelOfDetail]      = #detailed
  vRayInfo[#modelList]          = [aBarrierModel]
  vImpact = aMember.modelsUnderRay(aPosition,aDirection,vRayInfo)
  vImpactData = vImpact.getLast() -- VOID if no barrier detected
  return vImpactData
end GetImpactData
```

The output is either `VOID` (if you released the Puck outside the blue tray), or a property list containing three values that are important for treating the collision.

- `#distance`: a floating point distance to the barrier.
- `#isectPosition`: the point on the barrier where the impact will occur.
- `#isectNormal`: a unit vector pointing out at right angles from the barrier. This will be used to calculate the direction in which the Puck will travel after it bounces.

In the AirHockey.dir movie, the `mGetImpact()` handler also calculates the time of the impact from the value for `#distance`. It will use the value for `#isectPosition` as the starting point for the trajectory after the Puck bounces. It will use the value for `#isectNormal` to calculate the direction of the trajectory after the bounce. See "Bouncing off a wall" on page 291 to know how to treat of the bounce.

Additional examples

You can find other examples of collision detection with rays in the movies associated with the following articles:

- "Not walking through objects" on page 222
- "Moving to a given location" on page 225
- "Third-person camera" on page 230
- "Hugging Terrain" on page 238
- "Moving to a new zone" on page 240

2D barriers

Using modelsUnderRay() to send a ray out into 3D space can be costly in terms of the computer processing time. This article describes ways to keep the processing time to a minimum.

To detect a model in its path, a ray needs to perform the following steps.

- For each model, check if its boundingSphere is in the line of fire of the ray.
- If so, for each face in the model, starting with the nearest, check whether its normal is facing towards the ray.
- If so:
  - Calculate where the ray passes through the infinite plane which contains the face.
  - Calculate if the point of intersection with the plane lies within the face.

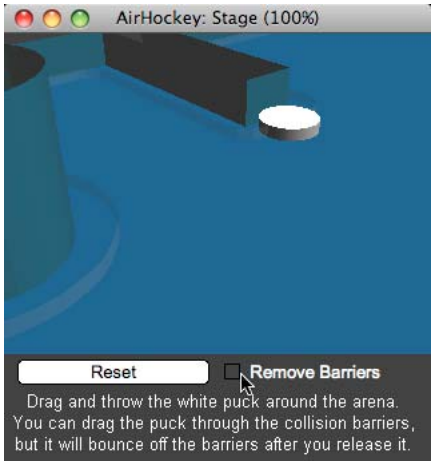
You can reduce the time it takes to calculate the result of a ray cast in a number of ways.

- Perform the ray cast on as few models as possible. To do this, use the `#modelList` option in the modelsUnderRay() call.
- Use models with as few faces as possible.
- Indicate the maximum distance at which a result is meaningful to allow the ray to ignore any models whose bounding sphere is out of range.

The graphic quality of a scene often depends on the details. Details require many models and a high polygon count. Rather than performing a ray cast on the scene that the user can see, consider creating a second scene that remains invisible to the user, specifically for raycasting. This invisible scene can use vastly simplified geometry.

Simplified collision geometry

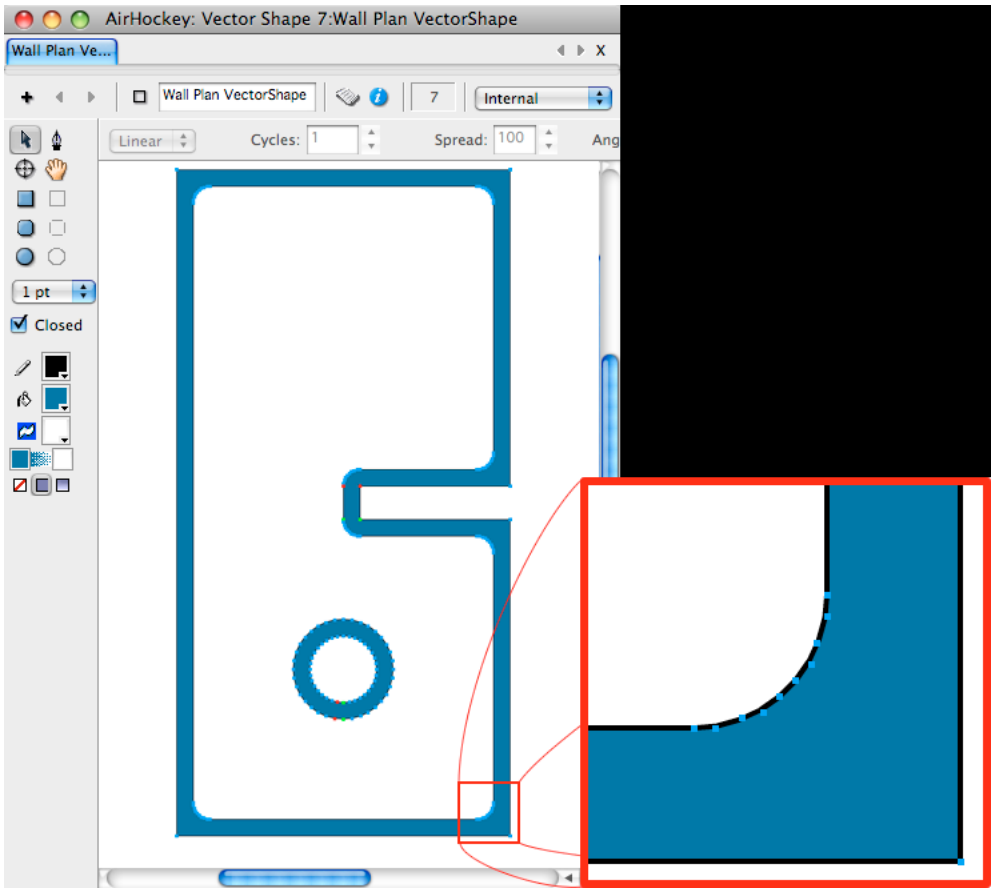
To see how to create such a simplified scene, download and launch the movie AirHockey.dir.



At the foot of each wall, a thin ribbon mesh acts as a collision barrier

Note: *In the AirHockey.dir movie, for clarity, the barriers are visible in the world. To improve performance, you can remove the barriers from the world. When a* `modelsUnderRay()` *call is made, (see 25.05.03 - Rays) the models to detect can be explicitly added to the* `#modelList` *options parameter. The models need not be in the world at all. If they are not in the world, the Director playback engine will not waste any time trying to display them.*

In this movie, the invisible barrier models are created from vertex data stored in a vectorShape member. You may prefer a different system for defining the floor plan of the models.



The floor plan for the barrier models can be stored in a vectorShape member

The barriers are designed with gently rounded interior corners. This makes it easy and natural to steer an avatar automatically out of a corner. See "Sliding along a wall" on page 237 for more details.

You may prefer to use fewer points than that are used here. Fewer points in the corners lead to fewer polygons in the final mesh. More points leads to smoother turning in corners.

You can find two handlers that take the vectorShape vertexList data and convert it to a list of 3D vectors in the Air Hockey behavior. They are called `mGetVertexList()` and `mConvertTo3D()`.

Building the barrier mesh

The barrier mesh is built on the fly. Here is a `CreateMesh()` handler:

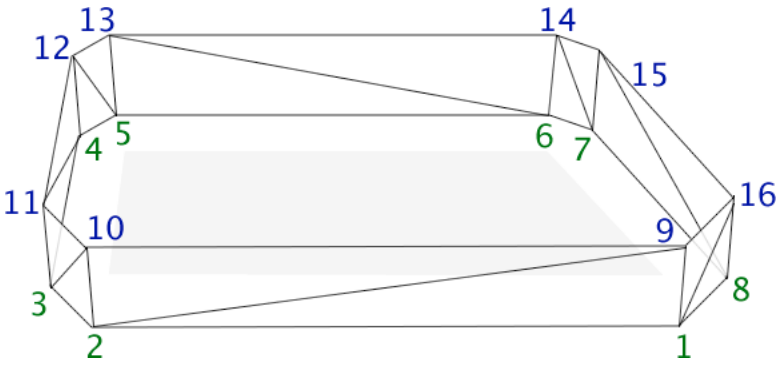
```
on CreateMesh(a3DMember, aVertexList, aName, aHeight)
  vFaceList = []

  vCount = aVertexList.count
  repeat with ii = 1 to vCount
    vVertex = aVertexList.getAt(ii).duplicate()
    vVertex.y = aHeight
    aVertexList.append(vVertex)

    if ii - 1 then
      vFaceList.append([ii, ii - 1,          ii + vCount - 1])
      vFaceList.append([ii, ii + vCount - 1, ii + vCount])

    else
      vFaceList.append([ii, vCount,     vCount * 2])
      vFaceList.append([ii, vCount * 2, vCount + 1])
    end if
  end repeat

  vNormalCount = 0 -- Set by generateNormals.build()

  vMeshResource = a3DMember.newMesh( \
aName, \
vFaceList.count, \
aVertexList.count, \
vNormalCount \
)
  vMeshResource.vertexList = aVertexList
  vCount = vFaceList.count
  repeat with ii = 1 to vCount
    vMeshResource.face[ii].vertices = vFaceList[ii]
  end repeat

  vMeshResource.generateNormals(#flat)
  vMeshResource.build()

  return vMeshResource
end CreateMesh
```

This handler accepts four parameters:

- `a3DMember`: the 3D cast member that contains the world
- `aVertexList`: a list of vectors describing a shape in the horizontal plane
- `aName`: a unique string that is not used for any existing modelResource
- `aHeight`: a positive number

The handler doubles the length of `aVertexList` by adding a new set of points that are `aHeight` units above the original set. It also creates a series of face definitions.

The following illustration shows a low-polygon barrier that can be placed around a building to prevent access to it:



Eight vector positions in aVertexList are extruded to make 16 vertex points, linked together as 16 faces

The figures in green represent the vectors used in `aVertexList` at the moment the `CreateMesh()` handler is called. The figures in blue represent the additional 8 points that are added.

Notice the order in which the original vertices are defined. Turning clockwise will create a mesh that faces outwards for a barrier that prevent entry. Turning counter-clockwise will create a mesh that faces inwards for a barrier that prevents exit.

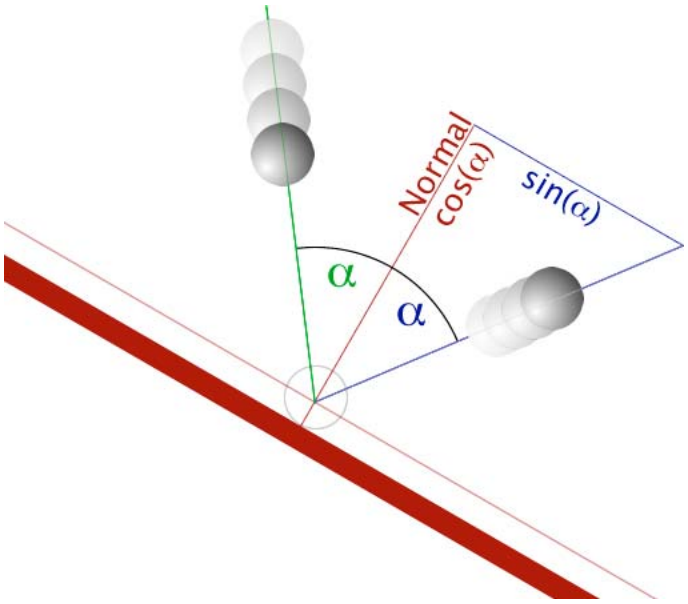
This mesh is never intended to be seen. All that matters is the geometry. There is no need to provide `colorList` or `textureCoordinate` information.

Bouncing off a wall

The behavior of an object after a collision depends on the object, and in particular, how much energy is restituted to the object after the collision. When an inelastic object, such as a wet rag, collides with a solid object, such as a wall, it loses all its energy and simply drops to the ground. A wet rag has very low *restitution*.

An elastic object has high restitution. When an elastic object, such as a ball, collides with a wall, two things happen:

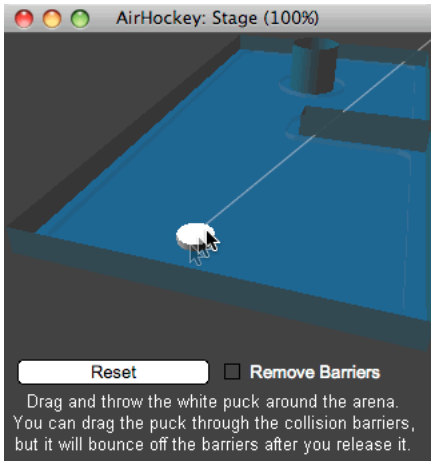
- The ball loses some of its energy to the wall. This energy is usually in the form of heat and sound. As a result, the speed of the ball immediately after impact is less than it was immediately before the impact. The difference in speed is determined by the restitution of the two colliding objects.
- The direction of motion is reflected in the wall. The angle of incidence is equal to the angle of reflection.



The change in trajectory of a ball as it bounces off a wall

This article shows how to calculate the movement of a round object after collision with a flat surface. A round object strikes a flat surface at only one point. A collision between objects with more complex geometry may result in many separate impacts, each of which will have an effect on the final result. In most cases, the rebounding object will start to spin. Treatment of more complex collisions is best handled by the Physics simulation provided by the Dynamiks xtra extension. See "Physics" on page 293 for more information.

To see an example of a round object rebounding of a flat surface, download and launch the movie AirHockey.dir.



After you throw the Puck model, it will slow down gradually and bounce off the walls

To see how to drag the Puck model around, see "Dragging" on page 250. To see how to make the Puck move after it has been released, see "Linear motion" on page 261. To see how to detect where and when the next collision with a barrier will occur, see "Rays" on page 285.

Calculating the new speed and direction

The following handler, `Bounce()`, calculates the new speed and direction of an object after a simple collision:

```
on Bounce(aSpeed, aDirection, aImpactNormal, aRestitution)
  vNewSpeed      = aSpeed * aRestitution
  vCosine        = -aDirection.dotProduct(aImpactNormal)
  vSine          = (1 - vCosine * vCosine)
  vVertical      = aDirection.crossProduct(aImpactNormal)
  vPerpendicular = aImpactNormal.crossProduct(vVertical)
  vPerpendicular = vPerpendicular.getNormalized() * vSine
  vNewDirection  = aImpactNormal * vCosine + vPerpendicular
  return [#speed: vNewSpeed, #direction: vNewDirection]
end Bounce
```

The handler accepts four parameters:

- `aSpeed`: a non-negative floating-point number in world units per millisecond
- `aDirection`: a unit direction vector, indicating the direction of travel
- `aImpactNormal`: a unit direction vector at right angles to the collision surface
- `aRestitution`: a floating-point number between `0.0` (all energy is lost) and `1.0` (all energy is returned to the bouncing object)

Note that `aDirection` and `aImpactNormal` will be pointing in opposite directions if the collision is exactly at right angles to the wall. In this case, the new direction after the bounce will be the same as `aImpactNormal`, or `-aDirection`.

You can find similar code in AirHockey.dir movie. Look in the Vector Mathematics script and in `mBounce()` handler of the Throw Model in Y Plane behavior.

The handler uses both trigonometry and vector mathematics. In particular, it uses the following mathematical operations:

- The `dotProduct()` of two unit vectors is equal to the cosine of the angle between them
- The `crossProduct()` of two vectors is a vector at right angles to both vectors.
- *`aVector`*`.getNormalized()` returns a vector with the same direction as `aVector` but with a `magnitude` of 1.0. The output vector has a length of one unit.

For more information on vector mathematics, see "Vector methods and operations" on page 364.

Physics

The Physics Xtra is a high-performance tool that helps developers create 3D worlds in which objects interact. The Xtra performs calculations to determine the results of collisions, factoring in object properties such as mass, velocity, and rotation. Forces can be applied, and objects can be connected to each other with constraints. The constraints available are six degrees of freedom joints, linear joints, angular joints, and spring joints.

Additionally, terrains and raycasting are supported. A terrain is similar to a bumpy plane that is finite in two dimensions and defines an elevation along the third. Raycasting is the mechanism of collision detection with rays. Raycasting can be done against all types of rigid bodies and terrains.

With this Xtra, developers can focus on game play and user interaction, and not worry about creating a real-time physics engine with Lingo scripts.

The Physics (Dynamiks) Xtra is a fully integrated rigid body physics simulation engine for Adobe® Director®. The dynamics Xtra is supported on Windows and Macintosh platforms.

You can use the Dynamiks xtra extension to control the interactions between the models in your 3D in a simulation of the laws of classical mechanics.

- You can create `rigidBody` objects with properties such as `mass`, `friction` and elasticity (or `restitution`). The models associated with these `rigidBody` objects will behave as if they were solid. See "Rigid bodies" on page 307 for more details.
- You can choose what collision geometry these `rigidbody` objects will use. In certain cases, a simple sphere or bounding box will give you enough control. In other cases you can provide a convex or concave hull as a `proxy` shape, or construct concave shape from multiple convex shapes. See "Rigid body proxies" on page 317 for more details.
- A `terrain` is an extension of the concept of the `rigidbody`. A `terrain` provides a surface that `rigidBody` objects cannot penetrate. A `rigidBody` placed below the surface of a `terrain` will automatically move to the surface. See "Terrains" on page 319 for more details.
- You can detect `rigidbody` objects using ray casting. See "Ray casting" on page 321 for more details.
- You can control collisions between `rigidBody` objects, and between `rigidBody` objects and `terrain` objects on an individual basis. You can register a handler to be called every time a collision occurs. See "Collisions" on page 279 for more details.
- You can create constraints between `rigidBody` objects, so that they move together or mutually limit each other's movements. You can also create a constraint between a `rigidBody` object and a fixed point. Constraints can be angular or linear or springs. See "Joints and springs" on page 330 for more details.
- A linear constraint allows movement in all directions, but constrains the orientation of the object to a given axis angle. See "D6Joint method and properties" on page 343 for more details.
- An angular constraint allows the object to rotate freely, but constrains its movement in space. See 25.06.13 - Angular joint properties for more details.
- You can create virtual springs that connect two `rigidBody` objects, or that connect a `rigidBody` object to a fixed point. A spring will resist extension, compression or both, and will try to return to a rest length. See "Spring properties" on page 336 for more details.
- You can create sophisticated joints that connect two `rigidBody` objects, or that connect a `rigidBody` object to a fixed point. These six-degrees-of-freedom joints can control the freedom of movement of the objects relative to each other for six different motions. They can also be used to drive a linear movement or a rotation. See "D6Joints" on page 338 for more details.
- You can create soft objects, using `cloth`. See "Cloth" on page 348 for more details.
- You can use a character controller to manage the movement of characters within a scene. See "Character controller" on page 352for more details.

Physics member

To create a physics simulation you need a Physics cast member. To insert a new Physics member into your cast library, choose Insert > Media Element > Physics.

A Physics member acts as an interface with the Dynamiks xtra extension, and it stores the properties used by a particular 3D member. Do not drag the Physics member onto the Stage or into the Score window. If you do, it will simply appear as a red rectangle with a red diagonal cross in it, which brings no visual enhancement to your movie.

Use Lingo or JavaScript code for all your interactions with a Physics member.

For details on how to set up a Physics member, see "Physics world" on page 298.

Controlling a physics simulation

Creating a credible physics simulation in a virtual 3D world means learning how to use a large number of properties and methods, and then letting the physics simulator take control.

The physics simulator does exactly what you tell it to do. It takes the initial conditions that you provide it with, and calculates what the virtual world will look like in a fraction of a second's time. It then takes that new situation, and performs the same calculations again. It repeats this process many times a second.

In the example movies provided elsewhere in this documentation, all interactions are controlled by visible code from Lingo or JavaScript handlers. If the code behaves in a way that you did not expect, you can add a breakpoint to a script, and step through it line-byline. You can follow the values of different properties in the Object Inspector. You can check at each point where there is a difference between your expectations and the actual outcome.

In a physics simulation, provide little direct access to the process. You control the initial conditions, and from time to time your code reacts to particular situations when they arise. For most of the time, you rely on the hidden code of the Dynamiks xtra extension to make the events in the world evolve in a coherent manner.

Learning to use the Dynamiks xtra means understanding the principles on which it runs. This section of the documentation is designed to help you reach that understanding. Many of the demo movies are designed to allow you to explore the different settings for a particular Physics object.

Simulations and shortcuts

Before exploring what a physics simulation can do, it is useful to consider how it is different from the real world.

In the real world, time appears as a constant flow. In a reel of film projected in a movie theater, each frame a separate image of the state of the world at a given point in time. If you watch a film frame by frame, the objects in the world jump from one place to another. The illusion that the objects are moving fluidly is generated in our brains.

A 3D simulation does not mimic the real world accurately. It takes advantage of the way our brains process information, and takes shortcuts. Most of the time the results appear natural, but the shortcuts can lead to strange behavior. Here are some points that you need to consider:

- A Physics simulation uses discrete steps to simulate movements.
- A Physics simulation uses unnaturally perfect starting situations.
- There are only a finite number of floating-point numbers that a Physics simulation can use to control actions and interactions. This can lead to rounding errors that accumulate over time, or to situations where the values go out of range.
- A Physics simulation uses invisible agents to apply forces and impulses.

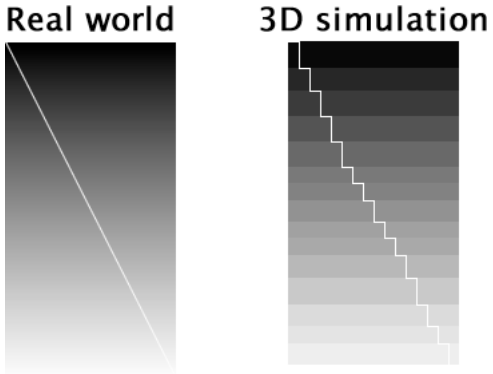
See "Physics world properties" on page 300 for an interactive illustration of these concepts.

A Physics simulation is designed to provide interactions between independent objects. If you want to create a mechanism where the movement of different objects needs to be locked precisely together, you may find it easier to write your own custom code.

Step-wise simulations

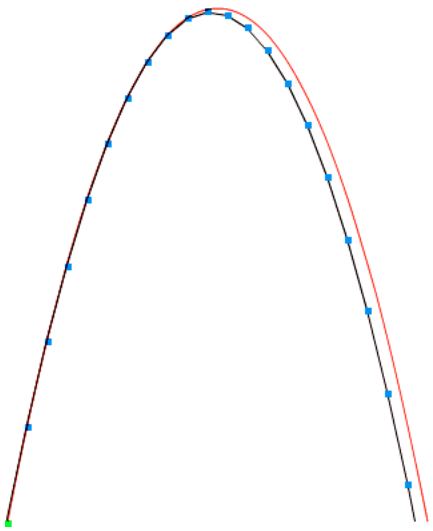
In a 3D world, time does not flow. It jumps in discrete time intervals from one frame to the next. In a physical simulation within a 3D world, many calculations are made between each frame to calculate the position of objects in the world in the next frame.

To speed up the calculations, approximations are used. For example, imagine an object that is slowing down to a stop. In the real world (assuming perfect conditions), the object's speed will decrease at regular rate. A 3D simulation will behave as if the speed of the object remained constant between each update, and then dropped suddenly at the next update time.



Changes in a 3D simulation occur in steps

Below is an illustration of the idealized movement of a ball thrown into the air (red curve) and the simulated movement of a virtual ball, calculated at discrete intervals of time.



A perfect curve and an approximation created from straight lines

In most cases, the results will appear similar to what would happen in the real world. A slight deviation from the theoretical result is perfectly acceptable. However, in some circumstances, this step-wise simulation can lead to unnatural behavior.

An ideal world?

In the real world, nothing is perfect. In a 3D physics simulation, everything is deterministic. A real-world object may start to spin as it falls, no matter how carefully you place it. In a 3D physics simulation, the object will only start to spin if you explicitly tell it to.

In the real world, a falling object will be deformed when it strikes the ground. The precise nature of that deformation at that moment in time will make it bounce in a particular way. In a 3D physics simulation, an object has an idealized shape, and it will bounce in an idealized way. A pointed object may bounce on its point and maintain its balance for an unnatural number of times.

However, in Director 11.5, the Dynamiks xtra does not allow you to reproduce exactly the same effect every time with exactly the same starting conditions. For example, it will not reproduce the same shot in a billiards game twice in a row. If you need perfect reproducibility (for example, to play a multiuser billiards game, with the same results on both users' computers), you need to write your own custom solution.

Iterative processes

A 3D simulation repeats the same calculations over and over, with the result of each calculation based on the outcome of the previous one. If the values that you set for the different parameters of your 3D world are carefully chosen, the simulation will remain stable. If the parameters are incompatible with each other and with the scene that you are creating, then the simulation may become chaotic. See more details here.

This theory is particularly important if you are using a large scale for your scene. The Physics simulator works well when gravity is near to `vector(0, -9.81, 0)` (for scenes with a vertical yAxis) or `vector(0, 0, -9.81)` (for scenes with a vertical zAxis). This implies that one world unit is equivalent to one meter.

A camera cannot see closer than one world unit away. If you have a scene where you need to get as close as an inch or a centimeter away from an object, you will need to increase the scale of everything in the world. This means scaling gravity, so that it represents a realistic value in world units.

If you create a scene where one world unit represents an inch or a centimeter, you will need to use a value for gravity in the range `-380.0` to `-981.0`. Before embarking a project at this scale, test whether the interactions that you want to create will be stable with a value for gravity in this range.

Invisible forces

In the real world, you can often tell when a force is acting on a body. You can feel and hear the wind blowing, or you can hear an engine revving, or you use your own muscles to apply the force.

When you apply a force or an impulse or set a velocity in a Physics simulation, you want to make sure that the value and the direction that you are setting are correct. However, the data you use to do this is abstract: a mathematical vector. This makes it complicated to debug projects that include physics simulations.

Spending a couple of hours creating a movie of your own to test one isolated aspect of a Physics object is often time well spent. If you have any difficulty visualizing a concept, or have problems with getting part of a Physics scene to work the way you expect, create a simple test movie. You can use the demo movies from this section as a starting point or as inspiration.

Physics world

Before you can use a Physics cast member to simulate a physical world, you need to initialize the cast member. To do this, you need to call physicsMember.init(). To get the simulation to run, you need to call physicsMember.simulate() once per frame.

Initialization

The `init()` command takes five parameters:

- The 3D cast member which contains the scene to be simulated
- A scalingFactor vector (whose value is ignored in Director 11.5)
- A timeStepMode, whose value may be `#automatic` or `#equal`.
- A timeStep, whose value is ignored if you choose `#automatic` for the mode. If you want to use the `#equal` mode with the same results as if you were using the `#automatic` mode, you can use `1.0 / _movie.frameTempo` for the timeStep.
- The number of subSteps to use between each screen update. More `subSteps` create more realistic simulations, but use more processing time. A recommended value for this parameter is `5`.

The following lines of code initialize the Physics member named "Physics" to use the 3D member named “3D” for a simulation in real time:

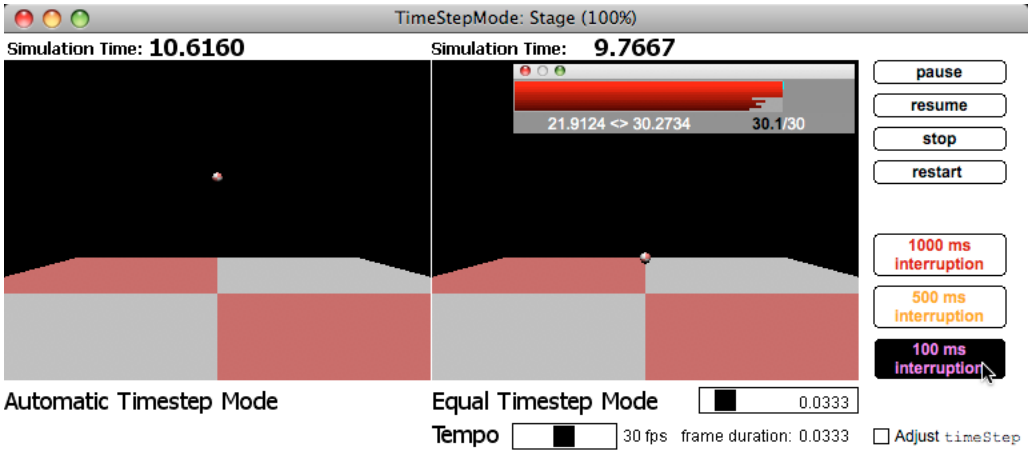
```
-- Lingo syntax
vPhysics  = member("Physics")
v3DMember = member("3D")
vScale    = vector(1, 1, 1) -- necessary but ignored
vMode     = #automatic
vTimeStep = 1.0 / _movie.frameTempo -- ignored in this case
vSubSteps = 5
vPhysics.init(v3DMember, vScale, vMode, vTimeStep, vSubSteps)
// JavaScript syntax
vPhysics  = member("Physics");
v3DMember = member("3D");
vScale    = vector(1, 1, 1); // necessary but ignored
vMode     = symbol("automatic");
vTimeStep = 1.0 / _movie.frameTempo; // ignored in this case
vSubSteps = 5;
vPhysics.init(v3DMember, vScale, vMode, vTimeStep, vSubSteps);
```

timeStepMode

The following are the two possible values for `timeStepMode`.

- `#automatic`: In this mode, the Physics simulation calculates how much time has elapsed since the last update, and moves the simulation forward by that amount of time. This mode is easier to work with, but it may break down if there are long interruptions between updates. For example, if the user switches to another application and back, or if a lengthy process in the movie itself monopolizes the computer processor.
- `#equal`: In this mode, the Physics simulation advances by the same time interval on every update. If the `timeStep` is equal to the duration of a Director frame (`1.0 / _movie.frameTempo`), then the result is almost equivalent to using the `#automatic` mode with better recovery after long interruptions. If the `timeStep` is significantly different from the duration of a Director frame, then the animation may seem to run too fast or too slow. In particular, if your movie is unable to play at the expected frame rate on a slow machine, the entire scene may update slowly.

To compare the differences between `#automatic` and `#equal` timeStep modes, download and launch the movie TimeStepMode.dir. You may also want to use the fpsMeter MIAW tool, to show the actual frame rate on a real-time basis. See "Performance" on page 387for instructions on how to install this tool.



A timeStepMode of #equal handles interruptions better than #automatic

The TimeStepMode.dir movie shows two versions of the same simulated scene side by side. The scene simulates a ball the size of a basketball bouncing after it is dropped from the height of a basketball net.

Try clicking the Interruption buttons. Both modes will freeze during the interruption. The `#automatic timeStepMode` simulation will try to catch up after the interruption. Depending on the timing and duration of the interruption, it may lose control of the simulation altogether. (If this happens, click the Restart button). The `#equal timeStepMode` scene will continue the simulation from the point at which it was interrupted.

Try varying the `_movie.puppetTempo` by dragging the Tempo slider. Notice how the `#automatic timeStepMode` simulation will adjust well to the new frame duration, so long as the frame rate does not drop too low. The pace of action in the `#equal timeStepMode` scene will change as the frame rate changes.

timeStep

To work around this drawback of the `#equal timeStepMode`, you can change the value of timeStep on each frame. See "simulate()" on page 299 for more details.

simulate()

To get the Physics simulation to run, you need to call physicsMember.simulate() once per frame. The best event handler to use for this is `on enterFrame`. See "Using frame events wisely" on page 397 for more details on this event handler.

You can make the `#equal timeStepMode` run smoothly, even when the frame rate fluctuates by altering the value of `timeStep` to take into account the elapsed time. To avoid the disadvantages of the `#automatic timeStepMode`, you can ignore long interruptions.

The `on enterFrame()` handler below sets the value of `member("Physics").timeStep` to the number of seconds since the last update, but it ensures that this value never goes above `0.1`. In other words, it ignores any interruptions longer than 100 milliseconds. If the movie's frame rate remains above 10 frames per second (or 100 milliseconds duration per frame), then the Physics simulation will adapt itself smoothly to any changes in tempo.

```
property pLastUpdate
on enterFrame(me)
  vMS = the milliseconds
  vElapsed = vMS - pLastUpdate
  pLastUpdate = vMS
  vTimeStep = min(vElapsed / 1000.0, 0.1)
  member("Physics").timeStep = vTimeStep

  member("Physics").simulate()
end enterFrame
```

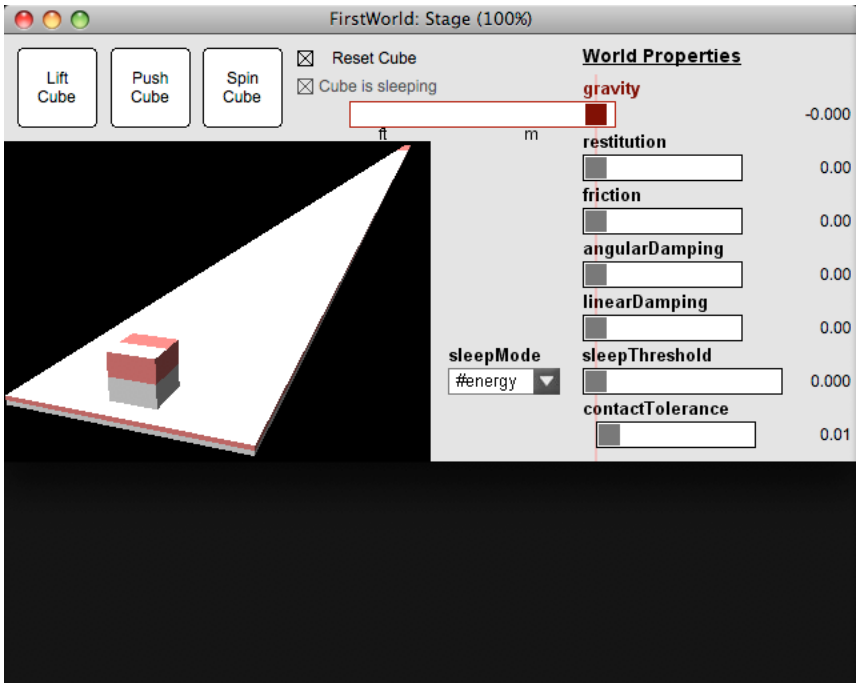
In the TimeStepMode.dir movie, you can test the efficiency of this technique by checking the Adjust TimeStep check box.

Physics world properties

A newly initialized Physics world has no gravity, and all its other properties are set to minimum values. A Physics world has eight properties whose values can be modified at any time. These values act as defaults for all `rigidBody` objects for which no value has been set for these properties.

- gravity (cannot be set independently on individual objects)
- restitution
- friction
- linearDamping
- angularDamping
- sleepThreshold
- sleepMode
- contactTolerance

To explore the properties of a Physics world, download and launch the movie FirstWorld.dir.



A simple Physics world with default values for all parameters

Use the various sliders and buttons in the FirstWorld.dir movie to test what happens when you modify the values for these properties.

Apart from `gravity`, every `rigidBody` in the Physics simulation can have its own values for the properties illustrated in the FirstWorld.dir movie. (See "Rigid body properties" on page 311). The values that you set for these properties will apply to all `rigidBody` objects and `terrains` for which you have not explicitly set their own values.

gravity and dimensions

The value that you use for gravity depends on the dimensions of your models. In the FirstWorld.dir movie the Cube has a length, width and height of 1 unit. If you consider 1 unit to be 1 meter, then setting a gravity to `vector(0, -10, 0)` will give you a realistic simulation. If you consider 1 unit to be 1 foot, then you can use `vector(0, -32, 0)`. Click on the M and FT sprites just beneath the gravity slider to use more scientifically precise values.

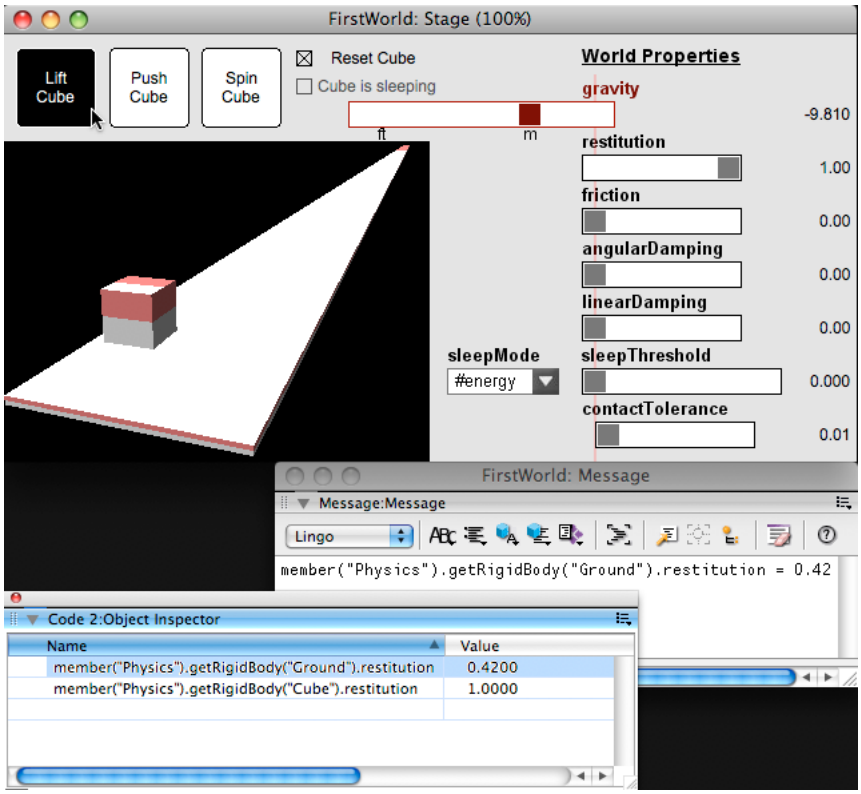
To simulate gravity on earth, use a value that is about 5 to 6 times the height of an average-sized human character, as measured in world units. (This assumes that you are using the #automatic mode for the simulation).

Unless you have a non-zero value for `gravity`, `friction` will have no effect on objects lying on top of one another. If you alter the value for `gravity`, then the effect of friction will also change. To test this, click the Push Cube button. Click the Friction slider bar, to set a non-zero value for `friction`, then click the Gravity slider to set a negative value for `gravity`. Only after `gravity` has been set will the Cube start to react to `friction`.

restitution

Restitution defines how much energy is returned to an object after it collides with another. The overall result depends on the restitution values for both bodies. Altering the value in the Restitution slider will affect both the Cube and the Ground. You can set the restitution value for each object separately. This command will set the restitution of the first `rigidBody` named "Ground" to `0.42`.

```
-- Lingo syntax
member("Physics").getRigidBody("Ground").restitution = 0.42
// JavaScript syntax
member("Physics").getRigidBodies()[1].restitution = 0.42;
```



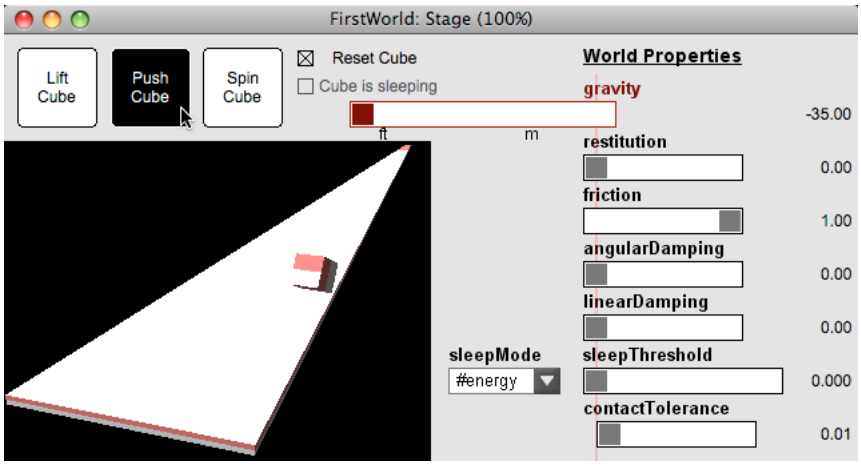
Testing how the Cube bounces when the value of its restitution changes

Test what happens when you alter the restitution of the Ground and the Cube rigidBody objects separately to get a feel of how this property works.

Note: *Setting* `restitution` *to its maximum value of* `1.0` *can lead to rounding errors where the energy of a bouncing object actually increases. Be cautious of setting restitution above* `0.95`*.*

friction

Friction depends on the force with which two objects are pressed together. To test this, set a non-zero value for both `gravity` and `friction`, and then click the Push Cube button. You can see that modifying either `gravity` or `friction` will have an effect on the behavior of the Cube.



Friction makes the Cube roll over and over, like a gaming dice, as it slows down

Friction affects both linear movement and angular movement. Try the same experiment using the Spin Cube button.

linearDamping

Damping is similar to friction in some ways, and different in others. the following are the 'damping' properties:

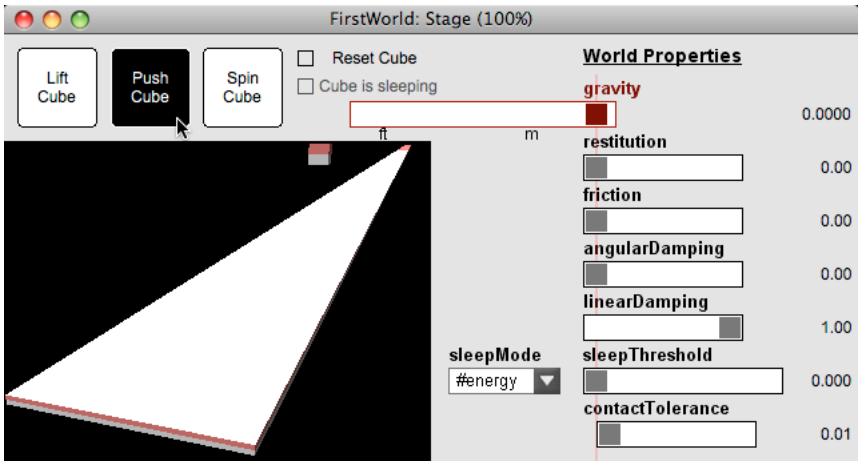
- `linearDamping` only affects linear movement
- `angularDamping` only affects angular movement.

The value of `linearDamping` determines how much energy is removed from an object moving through space, on every step. Like `friction`, a non-zero value for `linearDamping` will slow the object down. However, it is not dependent on any contact with another surface, and so it is not dependent on gravity. While friction can make an object roll on a surface, `linearDamping` will have no effect on its rotation.

To test this, do the following:

1. Restart the FirstWorld.dir movie.
2. Set a fairly high value for `linearDamping`.
3. Click on Lift Cube.
4. Deselect the Reset Cube check box so that the Cube will remain suspended in mid air.
5. Click the Push Cube button.

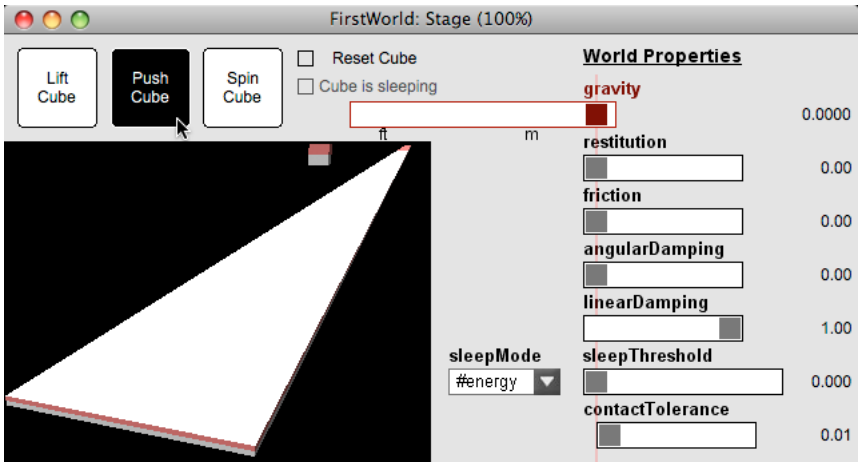
In zero gravity, the Cube will float above the ground, and come to rest some distance away.



linearDamping removes linear momentum from an object moving through space

AngularDamping

Angular damping has a similar effect as linear damping, except that it affects rotation only. To test, click the Spin Cube button. Setting a non-zero value for `angularDamping` will make the rotation of the Cube slow down and eventually stop. (Altering the value for `gravity` will not have any effect).



angularDamping removes angular momentum a rotating object

sleepThreshold

When you create a `rigidBody`, you can choose to make it `#dynamic`. See "Rigid bodies" on page 307 for more details.

For each dynamic `rigidBody`, the Physics simulation will check for possible interactions between it and every other object. The simulation will then update the dynamic `rigidBody`'s position on every frame. In a scene with many objects, this can lead to a colossal number of calculations on every frame. Too many calculations will adversely affect performance.

At any given time, many dynamic objects may not be moving. They may be struck by another moving object, but they will not themselves strike any non-moving objects. You can instruct the Physics simulation to consider that these objects are *sleeping*.

Here is an experiment to test this:

1 Type the following command in the Message window, but do not execute it yet.

```
member("Physics").getRigidBody("Cube").isSleeping = TRUE
```

2 Launch the FirstWorld.dir movie.

3 Set a very low negative value for `gravity` or a fairly high value for `restitution`.

4 Click the Lift Cube button.

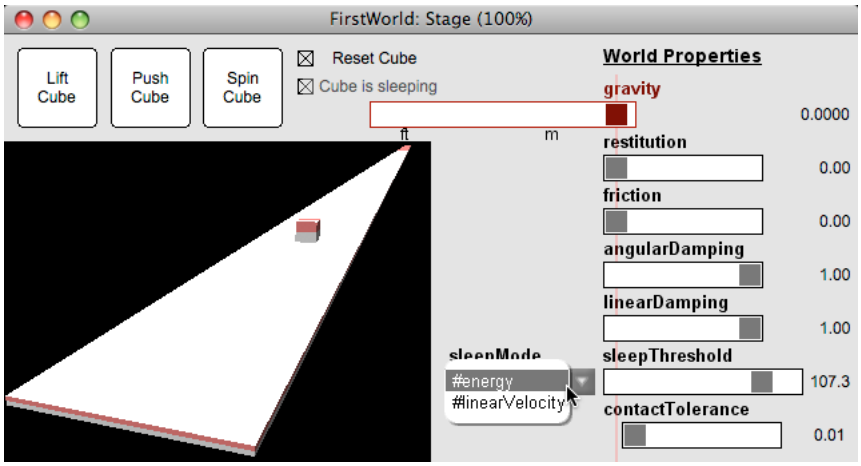
5 In the Message window, execute the command you typed earlier.

If you act quickly enough, you will be able to freeze the Cube in mid air.

The `sleepThreshold` property allows you to tell the Physics simulation when to consider that an object has stopped moving, and that it must start sleeping. Setting it to a non-zero value will mean that all objects that are affected by `friction`, `linearDamping` or `angularDamping` will eventually start sleeping.

The Physics Simulation behavior in the FirstWorld.dir movie checks the `isSleeping` property of the Cube `rigidBody` on every frame. When this value becomes `TRUE`, it checks the Cube Is Sleeping check box. You cannot interact with this check box button directly.

Be careful about setting the value too high because if the value is high, the objects can freeze in mid movement. The most efficient value will depend on what length a world unit represents in the real world, and on the value of `mass` property for dynamic `rigidBody` objects.



An exaggerated value of sleepThreshold can make objects go to sleep in unnatural positions

The value of `sleepThreshold` is zero by default. This means that, by default, no dynamic `rigidBody` objects ever sleep.

To test this, create a Lingo Movie Script and paste the following scriptText into it:

```
on testSleeping()
  vValue      = 1.0
  vDamping    = 0.5
  vSleepValue = 0.0

  the floatPrecision = 14
  repeat while vValue > vSleepValue
    vValue = vValue * vDamping
    put vValue

    if _key.keyPressed(SPACE) then
      exit
    end if
  end repeat

  the floatPrecision = 4
end testSleeping
```

From the Message window, execute the command below:

```
testSleeping()
```

Press the space bar when you are bored of seeing the Message window print out strings of zeros.

sleepMode

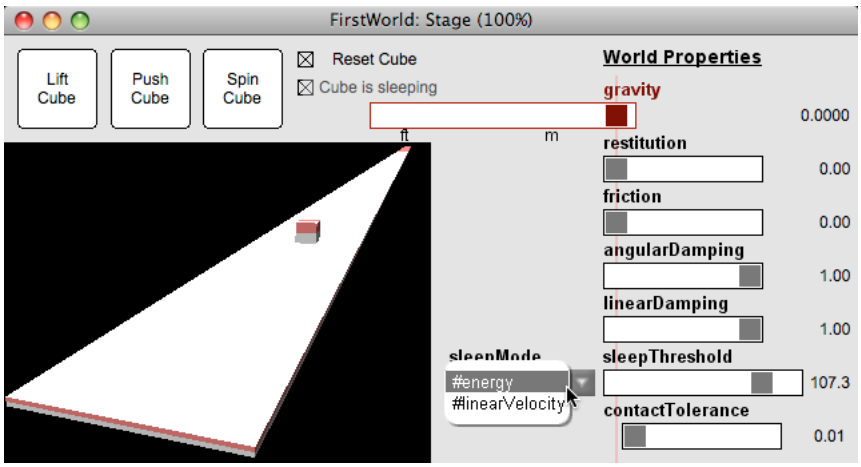
The `sleepMode` property can take one of two values:

- `#energy`
- `#linearVelocity`

The effect will be similar in both cases if you rely only on `friction` to reduce the speed of objects.

However, if you use `linearDamping` or `angularDamping`, you will see different effects for objects that move through space and objects that rotate. Choosing `#linearVelocity` will bring objects moving through space to a halt when their speed falls below the given threshold, but will allow objects that rotate to keep spinning even at very slow speeds.

To test this, set up the world properties as shown in the screenshot below:



Comparing #energy and #linearVelocity for sleepMode for linear and angular motion

1 Choose a high value for `angularDamping`, `linearDamping` and `sleepThreshold`.

2 Click the Push Cube button, and note how far the Cube moves before it comes to a halt.

3 Use the popup menu to change from `#energy` to `#linearVelocity` for the `sleepMode`.

4 Click the Push Cube button again.

5 Compare the distance the Cube traveled this time. It is likely to have stopped much sooner when `#linearVelocity` is selected.

6 With `#linearVelocity` selected for the `sleepMode`, click on the Spin Cube button. Count the seconds as the Cube turns until the Cube Is Sleeping check box becomes checked.

7 Use the popup menu to change from `#linearVelocity` to `#energy` for the `sleepMode`.

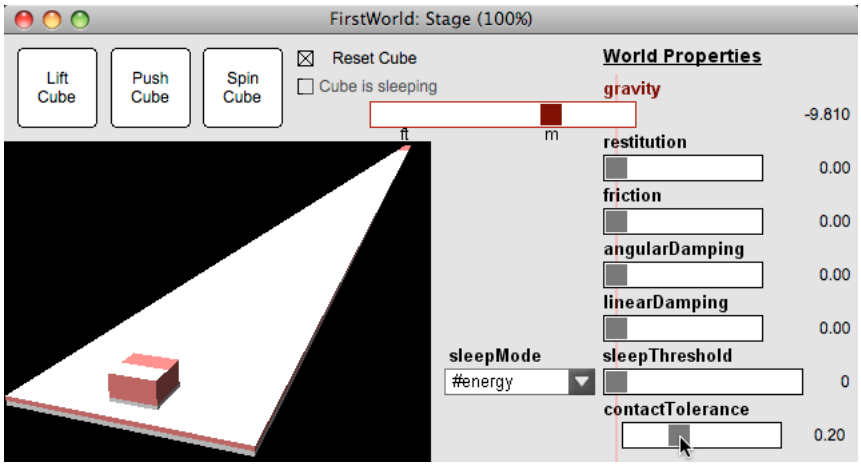
8 Click the Spin Cube button again.

9 Compare the number of seconds that the Cube spins before it comes to a halt.

You can see that the `#linearVelocity` setting is more aggressive with linear motion through space and less aggressive with angular motion. If, for example, you want to simulate a bicycle accident where the bicycle comes to a sudden stop but the wheels of the overturned bicycle keep spinning for a long time, choose the `#linearVelocity` setting.

contactTolerance

The `contactTolerance` property determines how many world units an object must penetrate into another before a collision is detected. A recommended value is 2% of the object's longest dimension. For the cube, which has sides 1 unit long, a good value would be `0.02`.



Increasing contactTolerance allows objects to interpenetrate deeper before a collision is detected

If you set this value too high, then collisions may fail to be detected at all, and unexpected behavior may occur.

Rigid bodies

To make a 3D model behave as if it were solid, you can create a rigidBody object for it. A rigidBody object is an abstract code object with no visible representation on the screen. Often, its geometry will be much simpler than the geometry of the visible model that it is attached to. The Physics simulation will use the geometry of each rigidBody object to detect collisions. Using simplified geometry improves performance.

Creating a new rigidBody

To create a simple rigidBody, you can use the physicsMember.createRigidBody() method. To create a rigidBody with more complex geometry, use the physicsMember.createRigidBodyFromProxy() method. See "Rigid body proxies" on page 317 for more details.

The `createRigidBody()` method takes up to five parameters:

1. A unique name for the rigidBody that you want to create.
2. The name of an existing 3D model.
3. A body proxy symbol indicating what type of simplified geometry to use for the rigidBody. This can take one of the following values:
   - `#box`
   - `#sphere`
   - `#convexShape`
   - `#concaveShape`
4. A body type symbol indicating whether the Physics simulation must check whether the movement of the rigidBody must trigger collision detection with other moving bodies. This symbol can take either of the following values:
   - `#static`
   - `#dynamic`

   Using a body type of `#dynamic` with a body proxy of `#concaveShape` will cause a script error. To create a dynamic rigidBody with concave geometry, you will need to use a proxy template. See "Rigid body proxies" on page 317 for details.
5. If you used `#concaveShape` for the body proxy parameter, you can use `#flipNormals` as a fifth parameter, to indicate that the normals for the rigidBody must be flipped to face inwards. This is useful for creating collision detection inside enclosed spaces like rooms. See "Collisions" on page 279 for an example.

Note: *Before calling `createRigidBody()` you must add the `#meshDeform` modifier to the model that is to be associated with the rigidBody object. When you do so, the Physics xtra gets access to the geometry of the model, and allows it to create the proxy geometry that it uses for collision detection.*

The Lingo handler below requires four parameters.

- `aPhysicsMember`: a pointer to a Physics member
- `a3Dmodel`: a 3D model in the 3D member that was used to initialize the Physics member
- `aShape`: either `#box`, `sphere`, `#convexShape` or `#concaveShape`
- `aBodyType`: either `#static` or `#dynamic`

The handler ensures that the #meshDeform modifier is attached to the model, and then creates a rigidBody object inside the Physics member.

```
on CreateARigidBody(aPhysicsMember, a3DModel, aShape, aBodyType)
  vName = a3DModel.name
  a3DModel.addModifier(#meshDeform)
  vRigidBody = aPhysicsMember.createRigidBody( \ vName, \ vName, \ aShape, \ aBodyType)
  return vRigidBody
end CreateARigidBody
```

The following commands use this handler to create a dynamic rigidBody for a model called "Ball", using a spherical body proxy, and then displays the properties of the proxy geometry:

```
-- Lingo syntax
vPhysics = member("Physics")
vBall    = member("3D").model("Ball")
vBallRB  = CreateARigidBody(vPhysics, vBall, #sphere, #dynamic)
put vBallRB
 -- rigidBody("ball")
put vBallRB.properties
 -- [#radius: 0.500, #center: vector( 0.000, 0.000, 0.000 )]
// JavaScript syntax
vPhysics = member("Physics");
vBall    = member("3D").getPropRef("model", 1);
<model("Ball")>
vBallRB  = CreateARigidBody(vPhysics, vBall, symbol("sphere"), symbol("dynamic"));
<rigidBody("ball")>
trace( vBallRB.properties);
// <[#radius: 0.500, #center: vector( 0.000, 0.000, 0.000 )]>
```

Using flipNormals

The commands below create a rigidBody named "Room" and associate it with a 3D model named box. The rigidBody is created with `#concaveShape` geometry and the normals flipped to face inwards.

```
-- Lingo syntax
vRB = member("Physics").createRigidBody("Room", "Box", #concaveShape, #static, #flipNormals)
put vRB
 -- rigidBody("Room")
put vRB.properties
 -- [#numVertices: 8, #numFaces: 12, #vertexList: [vector( 4.500, -2.000, 0.500 ), vector(
4.500, 2.000, -0.500 ), vector( 4.500, 2.000, 0.500 ), vector( -4.500, 2.000, 0.500 ), vector(
4.500, -2.000, -0.500 ), vector( -4.500, -2.000, -0.500 ), vector( -4.500, 2.000, -0.500 ),
vector( -4.500, -2.000, 0.500 )], #face: [[#vertices: [5, 2, 3]], [#vertices: [5, 3, 1]],
[#vertices: [4, 3, 2]], [#vertices: [4, 2, 7]], [#vertices: [8, 1, 3]], [#vertices: [8, 3, 4]],
[#vertices: [5, 1, 8]], [#vertices: [5, 8, 6]], [#vertices: [5, 6, 7]], [#vertices: [5, 7, 2]],
[#vertices: [7, 6, 8]], [#vertices: [7, 8, 4]]]]
// JavaScript syntax
vRB = member("Physics").createRigidBody("Room", "Box", symbol("concaveShape"),
symbol("static"), symbol("flipNormals"));
<rigidBody("room")>
trace(vRB.properties);
// <[#numVertices: 8, #numFaces: 12, #vertexList: [vector( 25.000, 25.000, -25.000 ), vector(
25.000, -25.000, -25.000 ), vector( -25.000, -25.000, -25.000 ), vector( 25.000, 25.000, 25.000
), vector( 25.000, -25.000, 25.000 ), vector( -25.000, 25.000, -25.000 ), vector( -25.000, -
25.000, 25.000 ), vector( -25.000, 25.000, 25.000 )], #face: [[#vertices: [1, 2, 3]],
[#vertices: [1, 3, 6]], [#vertices: [2, 1, 4]], [#vertices: [2, 4, 5]], [#vertices: [7, 3, 2]],
[#vertices: [7, 2, 5]], [#vertices: [8, 4, 1]], [#vertices: [8, 1, 6]], [#vertices: [8, 6, 3]],
[#vertices: [8, 3, 7]], [#vertices: [7, 5, 4]], [#vertices: [7, 4, 8]]]]>
```

Note: *Attempting to use `#flipNormals` as a parameter for `createRigidBody()` can provoke a script error. Do not use it when the body proxy symbol is #sphere or #box. If you use it with a body proxy symbol of `#convexShape`, no error will occur, but no action will occur either. The normals will not be flipped.*

Accessing a rigidBody

There are three methods for obtaining a pointer to a rigidBody

- physicsMember.getRigidBody() returns a pointer to the rigidBody whose name you pass as a parameter, or VOID if no rigidBody with that name exists.
- physicsMember.getRigidBodies() returns a list of all the rigidBody objects in the Physics world.
- physicsMember.getSleepingBodies() returns a list of all the dynamic rigidBody objects in the Physics world which are currently not moving.

Note: *If you use an integer as the parameter to* `getRigidBody()`*, a script error occurs. To access a rigidBody by its integer index number, use the following syntax:*

```
vRB = physicsMember.getRigidBodies()[aIndex]
-- Lingo syntax
put member("Physics").getRigidBody("Ball")
 -- rigidBody("ball")
put member("Physics").getRigidBody("non-existant")
 -- <Void>
put member("Physics").getRigidBodies()
 -- [rigidBody("bat"), rigidBody("ball")]
put member("Physics").getSleepingBodies()
 -- [rigidBody("ball")]
// JavaScript syntax
trace(member("Physics").getRigidBody("Ball"));
// <rigidBody("ball")>
trace(member("Physics").getRigidBody("non-existant"));
// undefined
trace(member("Physics").getRigidBodies());
// <[rigidBody("bat"), rigidBody("ball")]>
trace(member("Physics").getSleepingBodies());
// <[rigidBody("ball")]>
```

Deleting a rigidBody

To delete a rigidBody from a Physics member, you can use deleteRigidBody().

```
-- Lingo syntax
vRB = member("Physics").getRigidBody("bat")put put member("Physics").deleteRigidBody(vRB)
 -- 1
put member("Physics").deleteRigidBody("ball")
 -- 0
// JavaScript syntax
vRB = member("Physics").getRigidBodies()[1];
<rigidBody("bat")>
member("Physics").deleteRigidBody(vRB);
1
vName = member("Physics").getRigidBodies()[1].name;
ball
member("Physics").deleteRigidBody(vName);
1
```

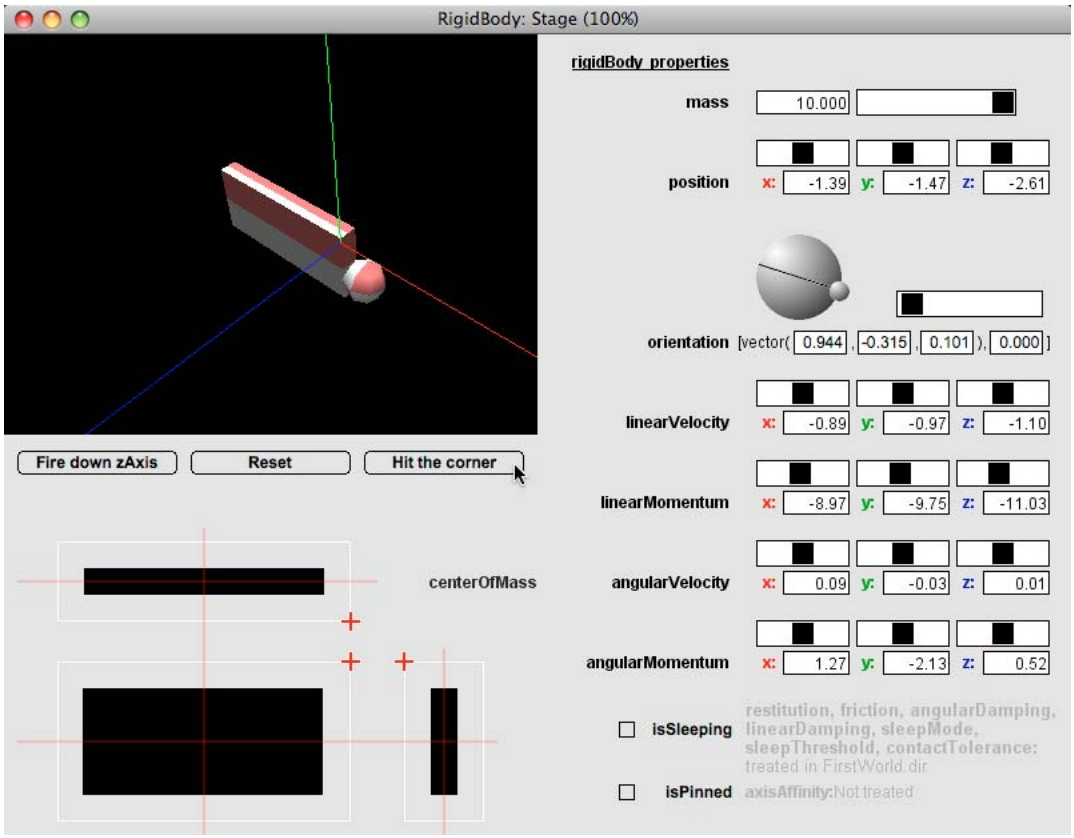
Note: *Calling* `deleteRigidBody("aNonExistantBodyName")`*, and then attempting to create a rigidBody with that name can lead to a script error.*

Rigid body properties

RigidBody objects have many properties. To make them easier to learn, they are grouped in categories below.

- Get-only properties
- Properties inherited from the Physics world
- Mass and position properties
- Static properties
- Motion properties
- Other properties

For a demonstration of the properties inherited from the Physics world, see "Physics world properties" on page 300. To experiment with the remaining properties, download and launch the movie RigidBody.dir.



RigidBody.dir allows you to experiment with many rigidBody properties

For better performance, close the Cast window when working with the RigidBody.dir movie. Director will no longer need to update the thumbnails of the text members in real-time.

Get-only properties

A `rigidBody` object has five properties whose value is set at the moment that you call `createRigidBody()`, and which cannot be changed later. Attempting to alter the value of these properties will provoke a script error.

- rigidBody.name: the string name of the rigidBody.
- rigidBody.model: the model with which the rigidBody is associated.
- rigidBody.shape: `#box`, `#sphere`, `#convexShape` or `#concaveShape`
- rigidBody.properties: a property list describing the geometry used by the rigidBody. The precise structure of the list will depend on the `shape` of the rigidBody. Examples are given below.
- rigidBody.type: `#dynamic` or `#static`

The following are examples of different `properties` lists for different shape. The contents and layout of the output has been edited for clarity. To see the full results, download and launch the movie Rigid Body Proxy Shapes.dir, then try the same commands in the Message window.

```
put member(1).getRigidBodies()[2].shape
 -- #sphere
put member(1).getRigidBodies()[2].properties
 -- [#radius: 44.534,
     #center: vector( 0.011, 0.734, -10.411 )]
put member(1).getRigidBodies()[1].shape
 -- #box
put member(1).getRigidBodies()[1].properties
 -- [#length: 52.138,
     #width:  89.067,
     #height: 11.572,
     #center: vector( 0.011, 3.479, -4.689 )]
put member(1).getRigidBodies()[3].shape
 -- #convex
put member(1).getRigidBodies()[3].properties
 -- [#numVertices: <integer>,
     #numFaces:    <integer>,
     #vertexList: [<vector>, ...],
     #face: [[#vertices: [<integer>, <integer>, <integer>], ...],
     ...]]
```

Properties inherited from the Physics world

The default values for following 7 properties are set by the equivalent properties for the Physics member itself. See "Physics world properties" on page 300 for an illustration of these properties.

- rigidBody.angularDamping
- rigidBody.linearDamping
- rigidBody.contactTolerance
- rigidBody.friction
- rigidBody.restitution
- rigidBody.sleepMode
- rigidBody.sleepThreshold

Mass and position properties

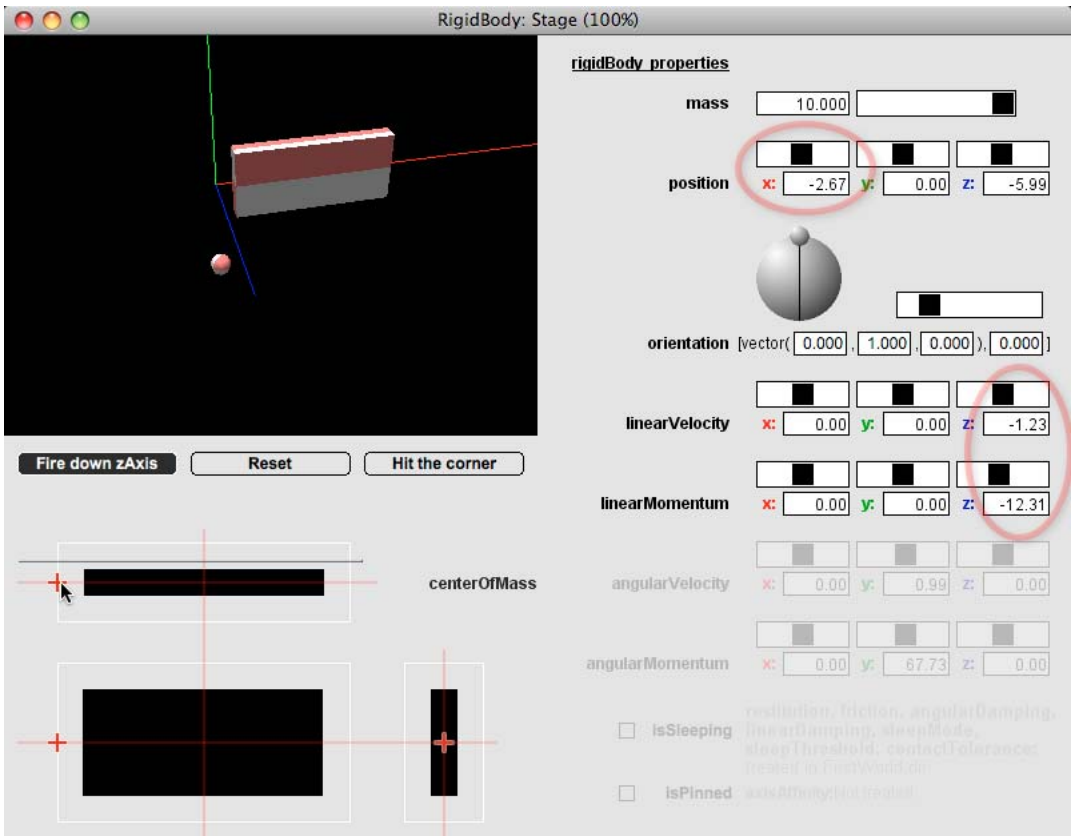
- rigidBody.mass: a non-negative number. The default value of mass is `0.0`, but rigidBody objects with a mass of zero behave the same way as objects with a mass of `1.0`. A rigidBody with a `mass` of zero will fall under the influence of `gravity`, and will have a `linearMomentum` equal to its `linearVelocity`.

  It is good practice to set the mass of all `#dynamic` rigidBody objects.

- rigidBody.centerOfMass: a position vector, relative to the origin point of the model's geometry. The value by default is `vector(0, 0, 0)`. This defines the point around which the rigidBody will rotate.
- rigidBody.position: a vector giving the current world position of the rigidBody. This will be the same as the `worldPosition` of the model associated with the rigidBody.
- rigidBody.orientation: a list containing a vector axis direction and a scalar angle. This will be the same as the transform.axisAngle property of the model associated with the rigidBody.

If you change the mass of a moving rigidBody on the fly, the value of its `linearVelocity` and `angularVelocity` will remain unchanged, but the value of its `linearMomentum` and `angularMomentum` properties will be adjusted automatically.

To test these properties, launch the RigidBody.dir movie. Click the Fire Down zAxis button, to make a ball with a mass of 1 strike the block. Check the value of `linearVelocity` and `linearMomentum`. Click the Reset button, alter the mass of the block, and then click the Fire Down zAxis button again. Notice how the change affects both the relative movements of the ball and the block, and how the `linearVelocity` and `linearMomentum` are related. Now try moving the `centerOfMass` to an off-center position and try again.

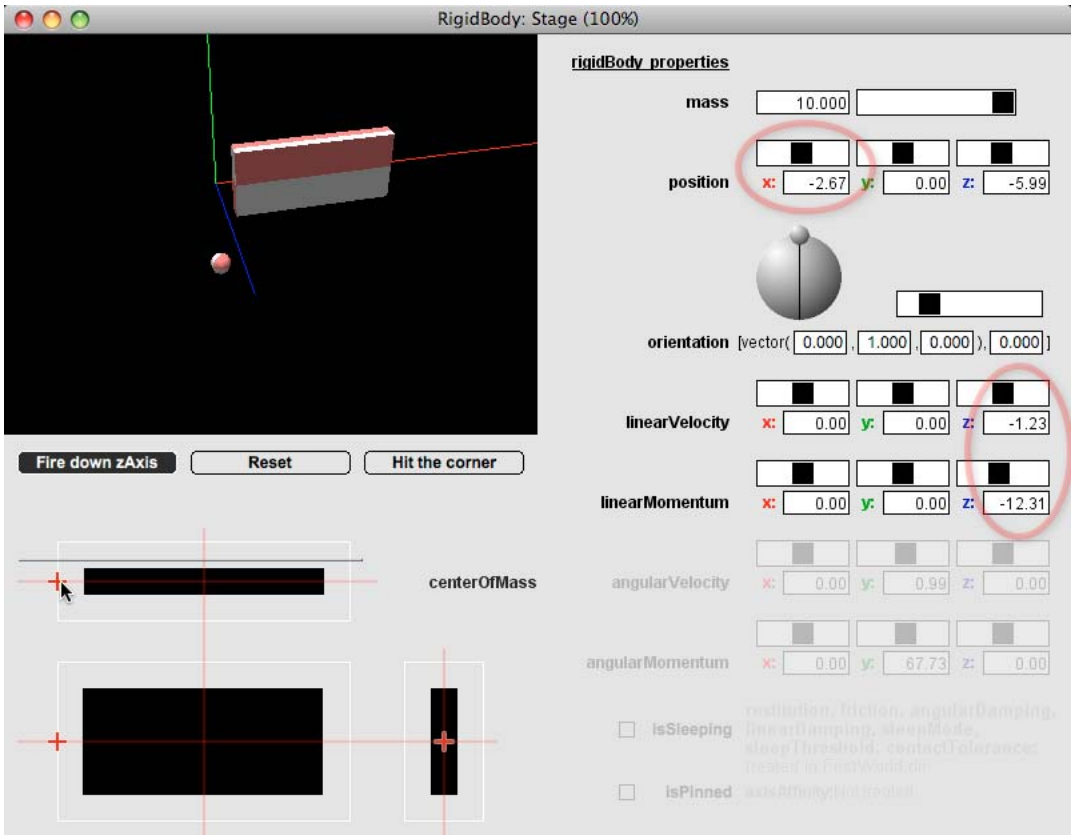


Testing how mass and centerOfMass affect how a rigidBody behaves after a collision

In the test above, notice how the `linearVelocity` and `linearMomentum` indicate that the `centerOfMass` is moving at a constant rate along the zAxis. However, the `position` of the rigidBody oscillates on the xAxis as the block rotates around its `centerOfMass`.

Try dragging the pinhead to change the orientation axis, and dragging the position and orientation angle sliders to see how this affects the block.

Note: *When the `orientation` angle is zero or 360°, the `orientation` axis will automatically be set to `vector(1, 0, 0)`. If you want to rotate a rigidBody around a different axis, be sure that you set both the axis and the angle each time.*



The orientation axis is automatically set to vector(1, 0, 0) for angles of 0 and 360

For a comparison of rigidBody.attemptMoveTo() and setting the rigidBody.position property, see "Rigid body methods" on page 316.

Note: *Notice how the orientation angle moves backwards and forwards between 0 and 360 and that the orientation axis flips through 180° each time the angle reaches 0 or 360.*

Static properties

A rigidBody created with a body type of `#static` will never move as the result of a collision. You can stop a `#dynamic` rigidBody from moving in two different ways.

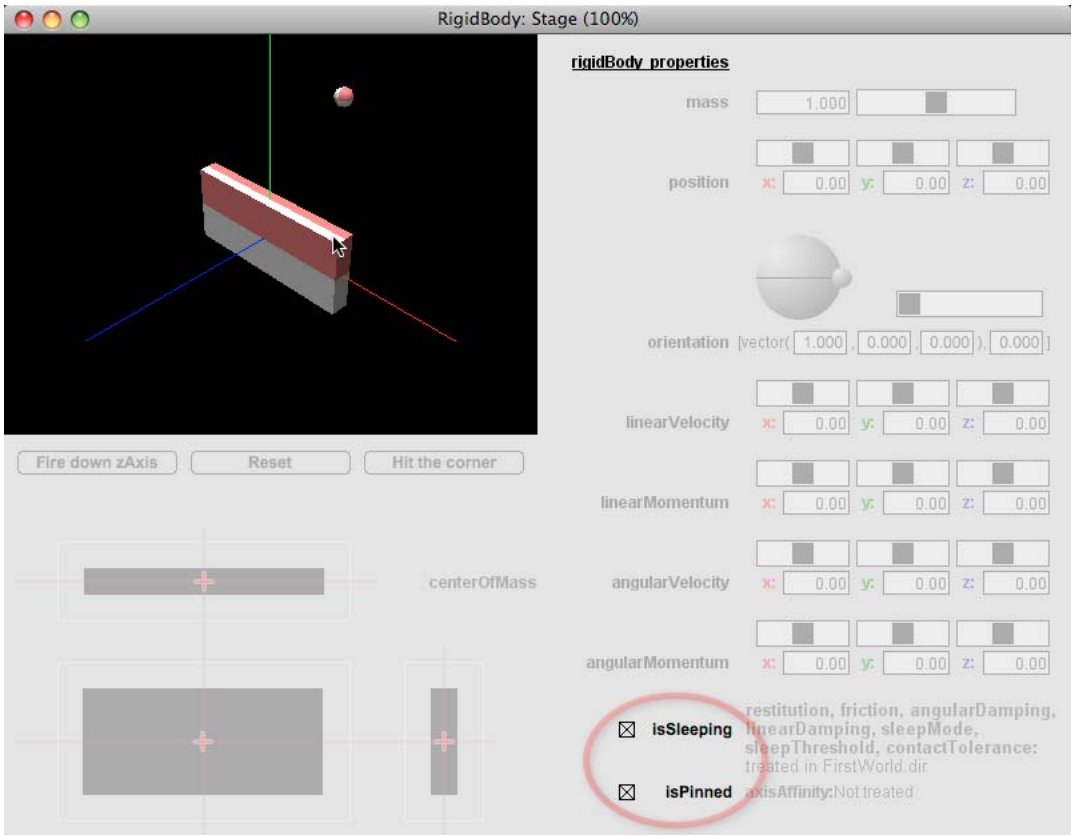
- rigidBody.isPinned can be `TRUE` or `FALSE`. If you set it to `TRUE`, then a dynamic rigidBody will behave like a static one. If it is struck by a moving dynamic rigidBody, it will remain pinned to its current position.

- rigidBody.isSleeping can be `TRUE` or `FALSE`. If you deliberately set it to `TRUE` it will freeze the movement of the object. Its velocity and momentum properties will be reset to zero. If a sleeping rigidBody is struck by another rigidBody, then it will stop sleeping and move in reaction to the collision.

  You can set the sleepThreshold property for the Physics world, or for each rigidBody individually to determine when a slowly moving object will be considered to have come to rest. See "Physics world properties" on page 300 for an illustration.

To experiment with these properties, launch the RigidBody.dir movie, and click the 3D sprite to fire a cannonball at the block. After the impact, select the isSleeping button. If you time your action right, you can arrange for the cannonball to strike the block a second time, and wake it up from its sleep.

You can also pin the block in position, by selecting the isPinned button. When you click on the block, the cannonball will bounce of it, but the block itself will not move.



Testing the isPinned property of rigidBody objects

Motion properties

- rigidBody.linearVelocity is a vector indicating the speed and direction of the motion of the `centerOfMass` of the rigidBody.
- rigidBody.linearMomentum is a vector obtained by multiplying the `linearVelocity` of the rigidBody by its `mass`. Momentum is conserved in elastic collisions. When two elastic objects collide in space, they exchange their momentum along the direction that is orthogonal to their surface of contact.
- rigidBody.angularVelocity is a vector indicating the speed and direction of the rotation of the rigidBody around its `centerOfMass`.

- rigidBody.angularMomentum is a vector obtained from a calculation combining the `angularVelocity` of the rigidBody and its `mass`. Click here for more details.

Changing the value of a velocity property will change the value of the associated momentum property, and vice versa.

Other properties

- rigidBody.axisAffinity is a property that is used only with concave rigidBody objects. See "Rigid body properties" on page 311 for more details on concave rigidBody objects.
- rigidBody.userData is a property that can be used to hold any Lingo value. Unlike the `node.userData` and `member3D.userData` properties, `rigidBody.userData` is initially `VOID`, not a property list. You can set it to a property list if you require.

Rigid body methods

RigidBody objects have five methods that allow you to move them in different ways.

- rigidBody.applyForce()
- rigidBody.applyLinearImpulse()
- rigidBody.applyAngularImpulse()
- rigidBody.applyTorque()
- rigidBody.attemptMoveTo()

To test the effects of these methods, download and launch the movie RigidBody.dir and test the following commands in the Message window. Click the Reset button between each command. Try changing the `mass` of the Box rigidBody. Note how this affects the velocity and momentum values. Try altering the position of the `centerOfMass` to see how this affects the movement.

Force and linearImpulse

A force is used to accelerate an object. An impulse provides it with momentum. If the rigidBody is initially at rest, you can imagine the difference being pushing a truck to get it started (force) or jumping off a moving truck and running beside it (momentum). In the first case, it takes the truck some time to accelerate to its top speed. In the second, you are already traveling at the top speed when you start running.

Apply a force at the centerOfMass:

```
member(3).getRigidBody("box").applyForce(vector(0, 0, -10))
```

Apply a force off-center. Note that the force can be applied outside the physical limits of the rigidBody itself:

```
member(3).getRigidBody("box").applyForce(vector(0, 0, -10), vector(10, 0, 0))
```

Modify the momentum of the rigidBody.

```
member(3).getRigidBody("box").applyLinearImpulse(vector(0,0,-10))
```

Apply an impulse off-center:

```
member(3).getRigidBody("box").applyLinearImpulse(vector(0,0,-10), vector(10, 0, 0))
```

Note that when you apply an off-center force along the zAxis, the resulting rotation is around an axis where the z value is zero. The rotation always starts at right angles to the initial force.

Torque and angularImpulse

A torque is a turning force applied to an object, like using a wrench. An angular impulse provides an object with angular momentum. You can think of the difference as between starting to push a merry-go-round in a children's park (force) and jumping off the merry-go-round while it is already in motion (momentum).

```
member(3).getRigidBody("box").applyTorque(vector(0,0,-10))
member(3).getRigidBody("box").applyAngularImpulse(vector(0,0,-10))
```

Note that torque and an angularImpulse applied along the zAxis create a rotation around the zAxis. Use the right hand rule to determine the direction of the rotation. Point your thumb along the axis of rotation and curl your fingers. A positive rotation will be in the direction in which your fingers point.

attemptMoveTo()

This method checks if the target position is free of obstacles. If so, the rigidBody is moved to the required position. If not, it does not move at all.

When you set the position property of a rigidBody directly, you can move one rigidBody into a collision position with another. If either body is free to move, it will move to a collision-free position on the next simulation step.

Reset the RigidBody.dir movie, and then try the example below in the Message window. Notice how the ball does not collide with the block until the fourth call, when its position is set so that it intersects with the block. Compare the results you achieve by setting the position with the results of using attemptMoveTo(), where no movement will occur if it will end in a collision.

```
member(3).getRigidBody("ball").position = vector(0, 0, 3)
member(3).getRigidBody("ball").attemptMoveTo(vector(0, 0, 2))
member(3).getRigidBody("ball").attemptMoveTo(vector(0, 0, 0.3))
member(3).getRigidBody("ball").position = vector(0, 0, 0.3)
member(3).getRigidBody("ball").position = vector(4, 0, -0.5)
member(3).getRigidBody("ball").attemptMoveTo(vector(4, 0, 0.5))
```

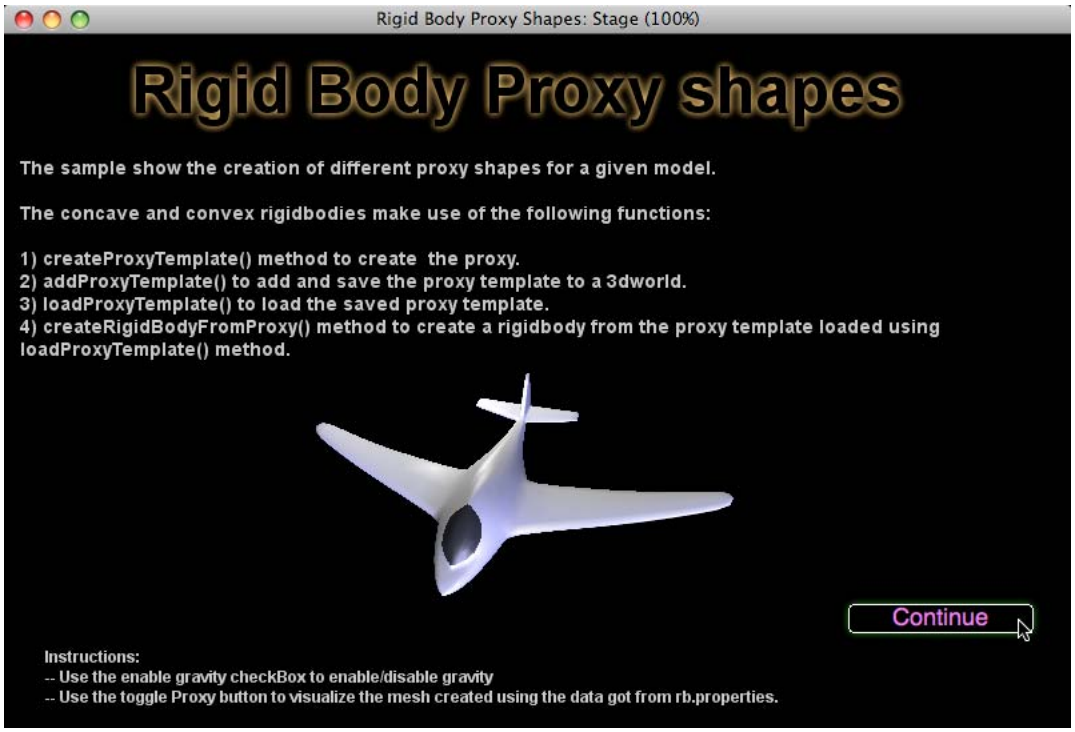
Reset the movie again, then check the isPinned button, and try the last command again. Notice how the behavior is different when the block is prevented from moving.

Rigid body proxies

The Dynamiks xtra allows you to define both convex and concave proxy templates. Use the following methods to create proxy templates and to create rigidBody objects from them:

- physicsMember.addProxyTemplate()
- physicsMember.createRigidBodyFromProxy()
- physicsMember.createProxyTemplate()
- physicsMember.loadProxyTemplate()

For a complete demonstration on how to use these methods, download and launch the movie Rigid Body Proxy Shapes.dir.



The Rigid Body Proxy Shapes.dir movie contains code to illustrate the creation of a rigidBody from a proxy template

Dynamic concave rigid body parameters

The following parameters are required for a dynamic concave rigid body:

- Depth
- Concavity
- Merge Volume

Depth

The Depth parameter specifies the maximum level of recursion for cutting the body. Valid values for this parameter are integers in the range [0,10].

Note: *The value of the* `Depth` *parameter does not generally exceed 7.*

The concave shape is cut recursively into smaller bodies along a plane. The plane for each cut operation is along the longest edge of the oriented bounding box of the input mesh.

Concavity

The volume of `Concavity` is computed as a percentage of the total volume of the input hull. Highly concave tiny bumps, which are insignificant in relation to the total volume, are ignored. If the volume of `Concavity` is below a threshold, it is considered "convex enough" and no further recursions are carried out.

Valid values of the Concavity parameter are real numbers in the range `[0,100]`. Suggested values for this parameter are from 0 through 20.

Merge Volume

The Merge Volume parameter determines how the convex meshes are cleaned and combined to form the complete body. The convex hull pieces created by decomposing the concave body are arranged in descending order of volume. Beginning with pieces having the largest volume, the volumes of the pieces are compared in sets of two. This procedure is repeated to determine the volume of the combined body.

If the combined volume is within a specified percentage threshold, the separate pieces are discarded, and the combined hull is retained. This procedure is repeated until no two hulls can be merged.

Valid values for this parameter are real numbers in the range `[0,100]`. Suggested values for the `Merge Volume` parameter are from 0 through 30.

Recommendations

Tweak the parameters carefully, because a proxy model can affect the performance of Director. Ensure that the concave mesh is closed. Ray-casting from inside a concave rigid body can cast itself. Keep in mind this scenario while ray-casting.

Terrains

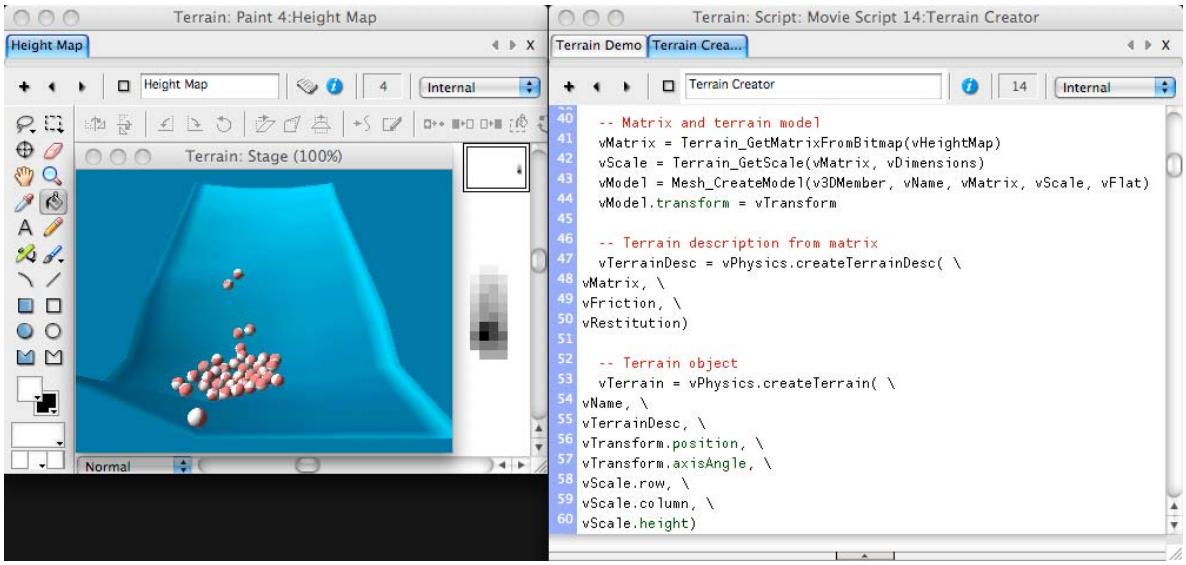
A terrain is a special kind of rigid body. It consists of a surface and extends downwards indefinitely. Any rigidBody placed below the surface will be moved vertically upwards on the next simulate step and placed on the surface.

A terrain is similar to a bumpy plane that is finite in width and length and defines an elevation at each point within that area. All values below the elevation are treated as being inside the terrain volume. A terrain is not a closed shape, but rather the boundary of a volume that extends downward for a certain distance.

Creating a terrain in Director involves the following steps:

1. In a third-party 3D design application, use terrain generator tools and generate the terrain mesh.
2. Save the corresponding height map as a director compatible file format (BMP, JPEG, or PNG) using the terrain generator tool.
3. Import the file into Director and create a height-map matrix by reading the file contents. For example, if you are using an image file, each pixel has to be read and the height has to be computed using the pixel information.
4. Create a mesh in director 3D using the height map information, at the appropriate scale.
5. Pass the height map matrix to the physicsMember.createTerrainDesc() method, along with values for `friction` and `restitution`.
6. Pass resulting `terrainDesc` object and various other parameters to the physicsMember.createTerrain() method

To see this in action, and to find a Movie Script that does steps 3 to 6 for you, download and launch the movie Terrain.dir. This movie takes a tiny bitmap and converts it into slope for a virtual egg-rolling ceremony.



The Terrain Creator script generates a terrain model and a terrain Physics object automatically

Creating custom terrains

You can take the Terrain Creator script and use it as is in your own projects. The Terrain_Create handler expects a property list with seven or eight properties:

- `#member`: 3D member used to display your scene.
- `#physics`: Physics member used to simulate the scene.
- `#name`: a unique string, not used either for a modelResource or a model name.
- `#heightMap`: a grayscale bitmap member, with the highest points shown in white and the lowest points shown in black. The bitmap can have any dimensions.
- `#dimensions`: a property list with positive floating point values for the `#width`, `#height` and `#length` (in world units) of the final terrain model. These values can be completely independent of the size of the height map in pixels.
- `#friction`: a floating point value between 0.0 and 1.0.
- `#restitution`: a floating point value between 0.0 and 1.0.
- `#transform`: the world-relative transform of the terrain in its final position.
- `#flat`: (optional). If this value is either `#flat` or `TRUE`, the terrain mesh will be created using `meshResource.generateNormals(#flat)`. If not, a smooth mesh will be generated.

Points to note

The matrix required to generate the `terrainDesc` object obliges you to create a mesh with its origin point at the far left corner. None of the coordinates in the resulting mesh can be negative. You will need to take this into account when positioning the terrain in 3D space.

It is recommended to use several small terrain objects, placed side by side. Using a single large terrain can lead to performance issues. For your own projects, you may want to find the optimal balance between multiple terrain objects, the terrain dimensions, and the polygon count for each individual terrain.

If you plan to modify this script, or to write our own terrain generator script, know that the matrix considers that *rows* run from front to back and *columns* are arranged from left to right. The Terrain Creator script takes this into account when reading in the height map data from the image.

The code in the Terrain Creator script is fully commented.

Ray casting

The Dynamiks xtra provides the following ray casting techniques that work in a similar way to member3D.modelsUnderRay():

- physicsMember.rayCastClosest()
- physicsMember.rayCastAll()

These methods work on both rigidBody objects and on terrains and take two parameters:

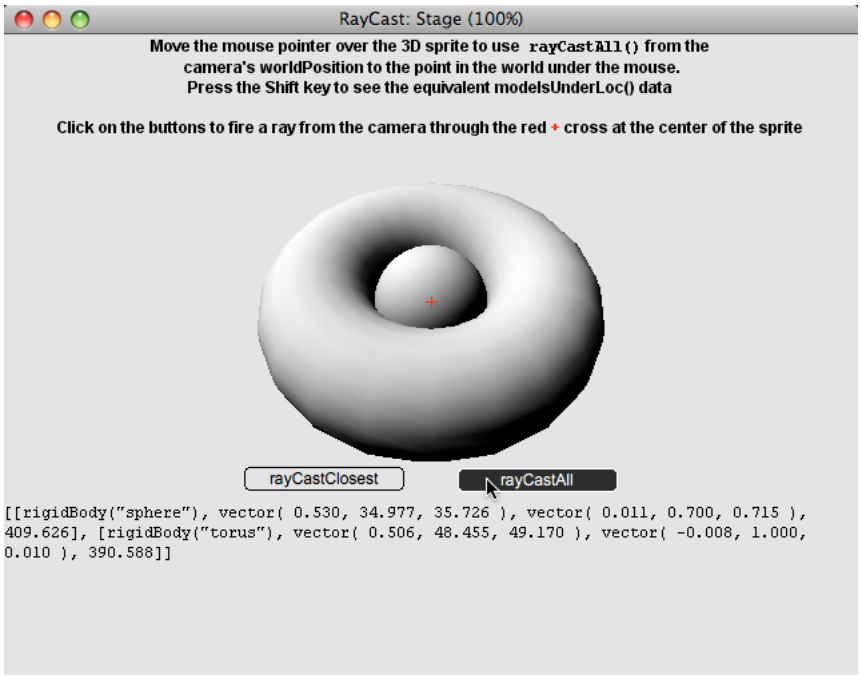
- A vector origin point, in world coordinates
- A vector direction

The output is a list (or a list of lists) containing the following data in the following order:

- Reference to the rigidBody or terrain object
- Point of intersection with the object, in world coordinates
- Normal to the point of intersection
- Distance from the origin of the ray to the point of intersection

Note: *camera.modelsUnderLoc() does not have an equivalent. However, you can achieve the same results by using the world position of the camera as the origin point of the ray. Then, use camera.spriteSpaceToWorldSpace() to obtain a point in the world from which to determine a direction.*

To see a demonstration of these methods, download and launch the movie RayCast.dir.



The result of member("Physics").rayCastAll() from the camera through the center of the sprite

Comparison with modelsUnderRay()

The results returned by the Physics member calls will not be identical to those returned by member3D.modelsUnderRay() or camera.modelsUnderLoc(), when used with the `#detailed` parameter.

Here is a comparison of the equivalent sections of the output from each method.

- `rayCastAll()`

```
[[rigidBody("sphere"),
  vector( 0.530, 34.977, 35.726 ),
  vector( 0.011, 0.700, 0.715 ),
  409.626], ...]
```

- `modelsUnderLoc()`

```
[[#model:          model("Sphere"),
  #distance:       409.717,
  #isectPosition: vector( 0.530, 34.911, 35.662 ),
  #isectNormal:    vector( 0.104, 0.646, 0.756 ), ...]
```

Here are the main differences:

- The 3D member methods return more details because they have access to more information, such as mesh, face and uv data.
- With member3D.modelsUnderRay(), you can choose to limit the ray-cast to a limited number of models over a limited distance. You do not have this option with the Physics ray casting methods.
- The 3D model and the Physics rigidBody do not use the same geometry. In most cases the rigidBody geometry is simpler. This gives a slight variation in the position and angle of the intersection. It can also mean that holes and concavities in models are not present in their associated rigidBody object.

In this particular case, the rigidBody for the torus is created using `#convexShape` geometry. This fills in the hole in the centre. In the Ray Cast Test behavior, you can find the line of code in the `mCreateTorus()` handler that creates the rigidBody, and alter it to use `#concaveShape` geometry instead.

Collisions

By default, a Physics simulation will resolve all collisions automatically. After a collision between two rigidBody objects, or between a rigidBody and a terrain, the simulation will work out a new trajectory for each of the objects. To do so, it will refer to the following properties of the colliding objects:

- rigidBody.angularMomentum
- rigidBody.linearMomentum
- rigidBody.restitution

If one object is a terrain or is defined as `#static`, or has its `isPinned` property set to `TRUE`, that body will not move.

You may want to deactivate collision detection for certain objects at certain times, or to be informed of when collisions between particular objects occur.

Controlling collision detection

To customize control of collisions detection, you can use the following methods:

- physicsMember.disableCollision(): allows rigidBody objects to pass through each other with no collision management applied to them. You can use this method in three different ways to:
  - Disable collision management between one object and all others
  - Disable collision management between a given pair of objects
  - Disable all collision management for the entire scene
- physicsMember.getCollisionDisabledPairs(): returns a list of the pairs of objects for which you have explicitly disabled collision management using `disableCollision()`.
- physicsMember.enableCollision(): resumes applying collision management to rigidBody objects for which `disableCollision()` had been used earlier.

Controlling collision callbacks

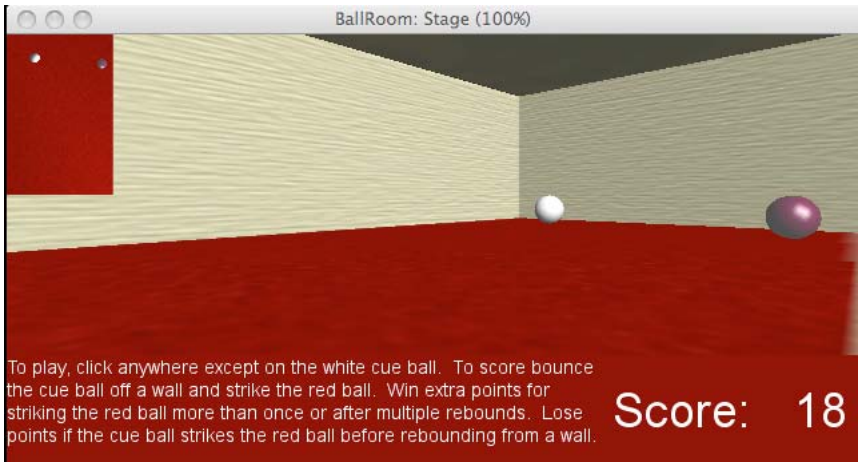
To customize the control of which collisions generate a callback event, you can use the following methods:

- physicsMember.registerCollisionCallback(): registers a handler to deal with all the reports of collisions generated.
- physicsMember.enableCollisionCallback(): starts generating collision callback events for particular objects
- physicsMember.disableCollisionCallback(): allows you stop generating collision callback events for particular objects. You can use this method in three different ways:
  - To disable collision callback when one particular object collides with any other object.
  - To disable collision callback when a given pair of objects collide.
  - To disable all collision callback for the entire scene.

- physicsMember.getCollisionCallbackDisabledPairs(): allows you to create pairs of objects which can collide with each other without generating a collision callback event, but which will continue to generate collision callback events when they collide with other objects.
- physicsMember.removeCollisionCallback(): stops triggering callbacks to the handler registered by `registerCollisionCallback()`.

Experimenting with collision detection

To see an example of how collision detection can be set up, download and launch the movie BallRoom.dir. This movie demonstrates a major issue with the way you need to design your 3D world to optimize for collision detection. It suggests various code-based solutions, and a design-based solution.



The BallRoom.dir movie uses collision detection to keep score

BallRoom.dir is a very simple game created from three rigidBody objects.

- `rigidBody("Room")` is based on a cuboid mesh model, and set to use `#concaveShape` geometry, with its normals turned to face inwards. This means that other objects can move around inside it, with collisions being detected on its inside surfaces. Because of its concave geometry, this must be created as a `#static` rigidBody.
- `rigidBody("Ball1")` is a dark red sphere, with a `#dynamic` body type.
- `rigidBody("Ball2")` is a white sphere, with a `#dynamic` body type.

The camera is placed in one of the corners of the room, and adjusted so that the whole area of the room is visible, with the exception of the ground immediately in front of the camera. The inset camera in the top left corner shows the room as viewed from the top.

After all the rigidBody objects have been created, the Collision Demo behavior executes commands similar to the following code.

```
member("Physics").registerCollisionCallback(#FilterCollision, me)
member("Physics").enableCollisionCallback()
```

The first call indicates that any collision callbacks that are generated has to be handled by the `on FilterCollision()` handler of the Collision Demo behavior. The second tells the Physics member to start generating callbacks for all collisions. No callbacks will be made until both commands have been made.

Using `enableCollisionCallback()` with no parameters means that callbacks for all existing rigidBody objects will be enabled.

Collision callbacks contact reports

Each time there is a collision between either Ball and the Room, or between the two Balls, the `FilterCollision()` handler will be called. It will receive a linear list of ContactReport objects. The Physics member is initialized to use five substeps, and there are two dynamic rigidBody objects. This means that up to ten ContactReports can be generated per frame.

To visualize what a ContactReport looks like, launch the BallRoom.dir movie, then execute the following command in the Message window to hijack the callback events:

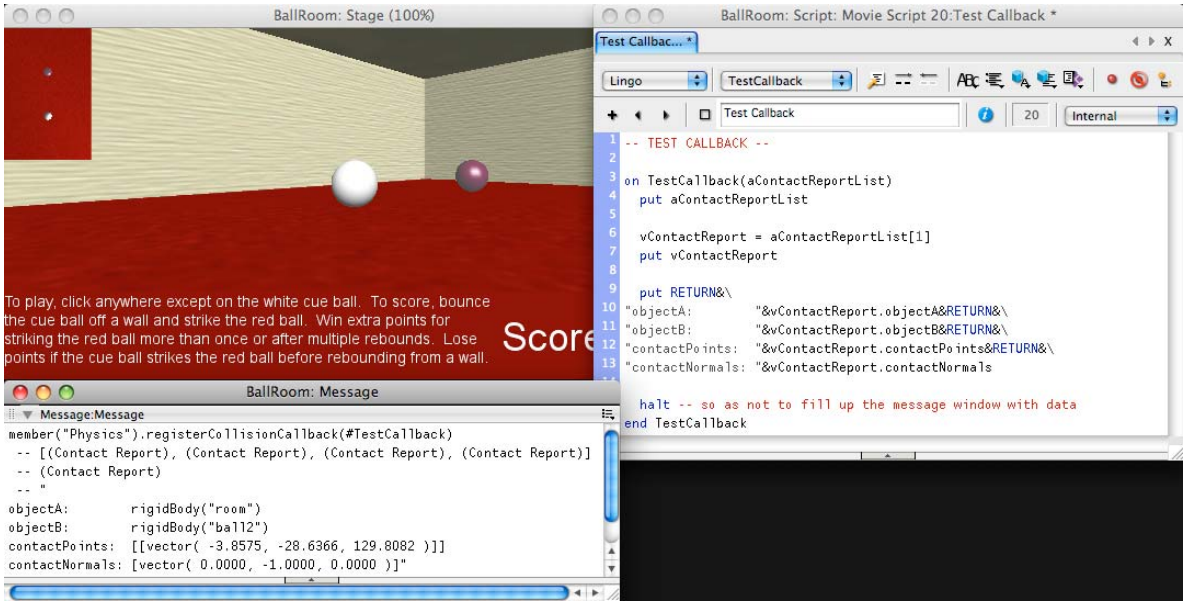
```
member("Physics").registerCollisionCallback(#TestCallback)
```

Now, click on the 3D sprite. You can see the following output printed into the Message window:

```
 -- [(Contact Report), (Contact Report), (Contact Report)]
 -- (Contact Report)
 -- "
objectA:         rigidBody("room")
objectB:         rigidBody("ball2")
contactPoints:   [[vector( -3.8575, -28.6366, 129.8082 )]]
contactNormals: [vector( 0.0000, -1.0000, 0.0000 )]"
```

Then the movie will halt to prevent spamming your Message window with endless reports.

You can find the `#TestCallback()` handler in the Test Callback movie script.



Using a test handler to inspect the data sent to a collision callback

Notice that the `objectA` and `objectB` values are listed in the order in which the rigidBody objects were created. `RigidBody("room")` was created before `rigidBody("ball2")`, so it appears first in the physicsMember.getRigidBodies() list, and is given top billing in the ContactReport.

The `contactPoints` and `contactNormals` values are both lists. In the illustration above, they both contain only one vector each. This is because the ball in question is just in contact with the base of `rigidBody("room")`. If the ball were also in contact with a wall, it would be touching `rigidBody("room")` in two places, so both lists contain two vectors.

In the case shown above, the `contactNormal` value is `vector(0.0, -1.0, 0.0)`. This vector points downwards. It indicates that `objectA` touches `objectB` at a point where the normal to the surface of `objectB` is facing downwards.

Filtering for pertinent collisions

The Physics simulation will send all ContactReports for all collisions between registered objects to a single handler. In the current simulation, this means that it will receive ContactReports on every frame saying that the ball objects are in contact with the bottom surface of the room object. However, the game play is set up so that only contacts with the walls or between two balls are of any interest. The role of the `FilterCollision()` handler is to filter out all these unneeded reports. (A better technique is explained in "Preventing unwanted collisions" on page 328).

In this game, it is enough to test whether the first (and only) `contactNormal` vector in the ContactReport is pointing downwards. Only ContactReports which indicate a collision where the normal is not parallel to the yAxis are passed on to the `mCheckCollisions()` handler, which keeps the score.

Here is a simplified version of the handler:

```
on FilterCollision(me, aContactReportList)
  ii = aContactReportList.count
  repeat while ii
    vContactReport = aContactReportList[ii]
    vNormal        = vContactReport.contactNormals[1]
    if vNormal.y < -0.999 then
      aContactReportList.deleteAt(ii)
    end if
    ii = ii - 1
  end repeat
  me.mCheckCollisions(aContactReportList)
on FilterCollision
```

This handler iterates backwards through the list of ContactReports, deleting those that are of no interest.

In your own projects, develop your own criteria for treating or ignoring ContactReports.

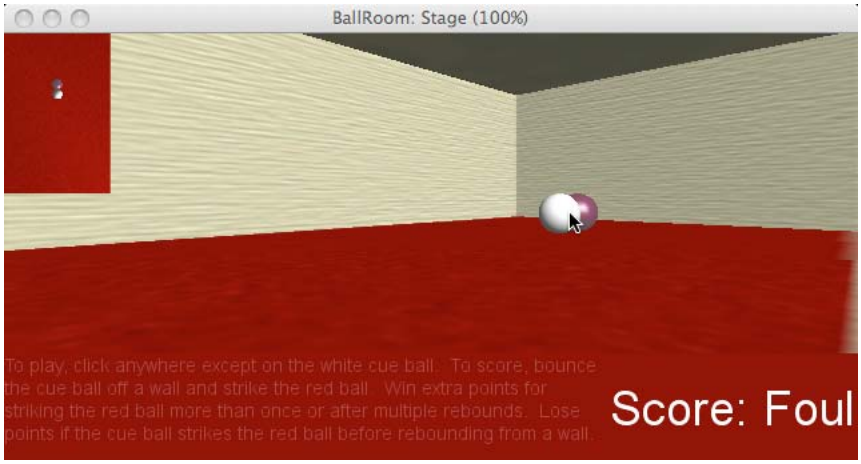
Limiting the number of reported collisions

To see exactly how many collision callbacks are generated, launch the BallRoom.dir movie, hold down the Option key (Macintosh) or Alt key (Windows) and click on the 3D sprite. You will see data filling up your Message window: up to 10 ContactReports per frame. Release the modifier key when you have seen the required messages. Here is a very small sample:

```
 -- "
objectA: rigidBody("room")
objectB: rigidBody("ball2")
points:  [[vector( -1.2625, -28.6313, 129.4819 )]]
normals: [vector( 0.0000, -1.0000, 0.0000 )]
******
objectA: rigidBody("room")
objectB: rigidBody("ball2")
points:  [[vector( -2.5208, -28.6300, 109.0314 )]]
normals: [vector( 0.0000, -1.0000, 0.0000 )]
******
objectA: rigidBody("room")
objectB: rigidBody("ball2")
points:  [[vector( -3.7750, -28.6331, 88.6485 )]]
normals: [vector( 0.0000, -1.0000, 0.0000 )]
****** …"
```

For performance reasons, try to generate only the collision events that are useful to you. In some cases, you can do this by limiting the number of objects that are registered to generate a callback event. Both physicsMember.enableCollisionCallback() and physicsMember.disableCollisionCallback() can be used with a variety of parameters to fine-tune which objects are enabled for callback at any given time.

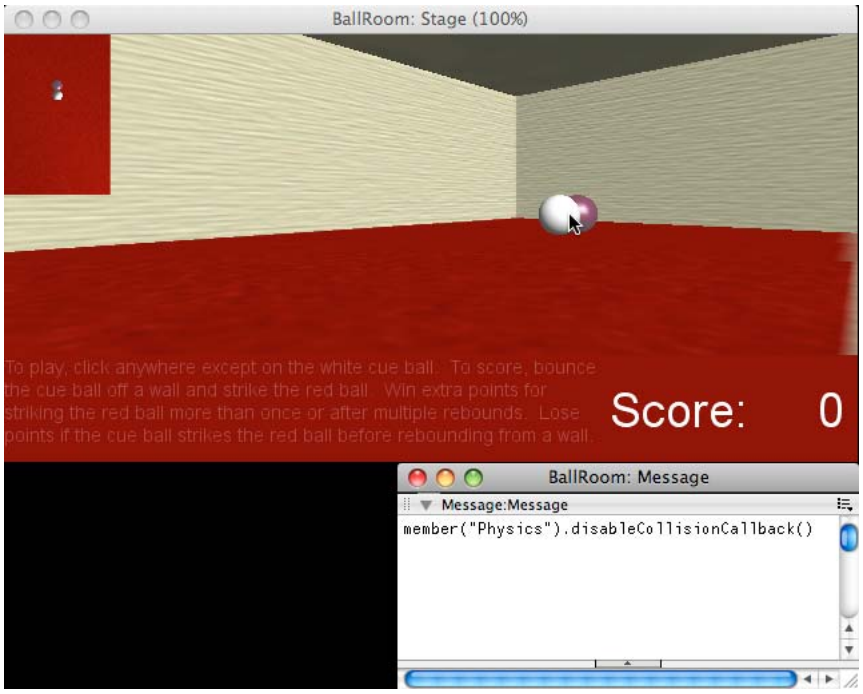
Launch the BallRoom.dir movie and click on the red ball. When the white cue ball hits the red ball without first bouncing off a wall, a 'foul' is given.



Click on the red ball. The collision will result in a Foul being reported

If you disable all callbacks, you can avoid this penalty. Restart the movie and then execute the following command in the Message window:

```
member("Physics").disableCollisionCallback()
```



If you disable all callbacks, no Foul is reported when the cue ball strikes the red ball directly

However, to win any points, you need to detect collisions between the cue ball and both the walls and the red ball. You can enable collision callbacks for a pair of objects with the `enableCollisionCallback()` method.

Relaunch the movie, then execute the following lines in the Message window:

```
member("Physics").disableCollisionCallback()
vBall1 = member("Physics").getRigidBody("Ball1")
vBall2 = member("Physics").getRigidBody("Ball2")
member("Physics").enableCollisionCallback(vBall1, vBall2)
```

Hold the Option/Alt key down and click on the red ball. This time, you will see very few events are reported in the Message window. The `rigidBody("room")` is not registered, so it generates no callbacks, neither for the ground nor for the walls.

Restart the movie and try a different approach. In the Message window, execute the following lines:

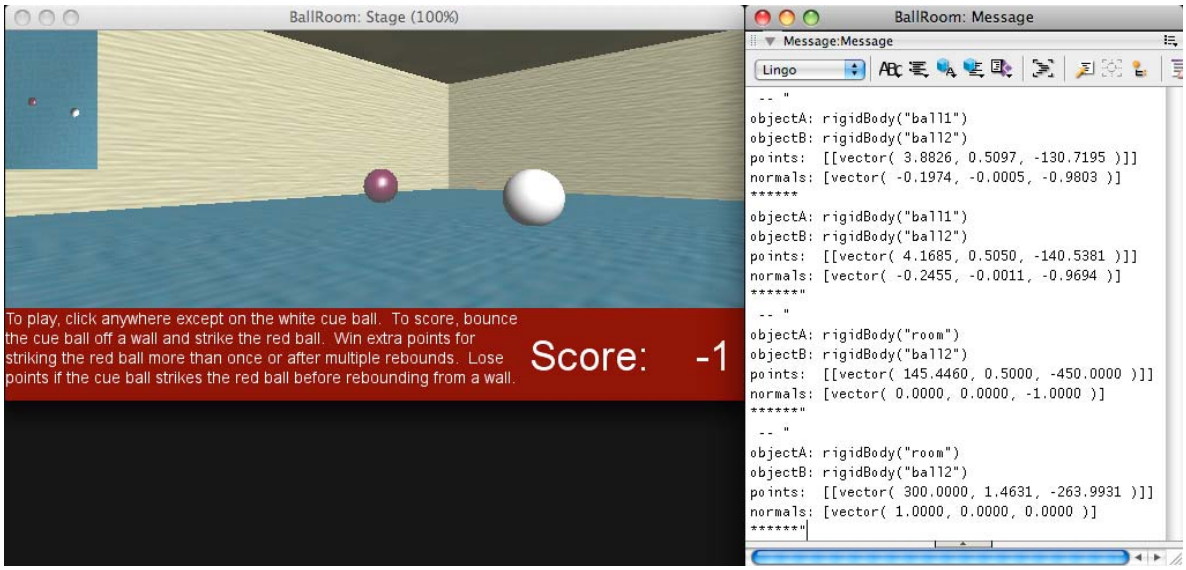
```
member("Physics").disableCollisionCallback()
vBall2 = member("Physics").getRigidBody("Ball2")
member("Physics").enableCollisionCallback(vBall2)
```

This code disables all the callbacks set up by the Collision Demo behavior, then re-enables callbacks generated by the white cue ball, when the ball is in contact with any other object. If you hold down the Option/Alt key, you will again see a flood of events. Collisions between the red ball and the room are now ignored, so that reduces the number by half.

Preventing unwanted collisions

In the BallRoom.dir movie, the problem is that the contact between the ground and the balls produces unwanted collision reports. A simple solution is to use a different rigidBody object for the ground, so that the balls only touch the rigidBody("room") when they strike a wall.

Hold down the Shift key and restart the BallRoom.dir movie. Now, hold down the Option/Alt key and click on the red ball. Watch the output in the Message window. Only useful collisions are now generated.



Using a separate rigidBody for the floor and excluding it from collision callbacks to solve the problem

Note: *Careful design of the models and rigidBody objects in you 3D world can make your job with scripting simpler. If you work with a 3D designer, you may need to explain your collision requirements in detail, so that the designer can suggest the appropriate solutions.*

Holding down the Shift key as you launch the movie makes the following lines of code execute in the `mInitialize()` handler of the Collision Demo behavior:

```
if the shiftDown then
  me.mCreateFalseFloor(vRadius, vSize)
  pPhysics.disableCollisionCallback(pRoomRB, pRedRB)
end if
```

The `mCreateFalseFloor()` handler creates an extra rigidBody called "False Floor" that has a surface just above the base of the Room object. Since it is created *after* the call to `enableCollisionCallback()`, this rigidBody is automatically excluded from the list of objects that generates callback events.

Used with two parameters, the `disableCollisionCallback()` explicitly prevents collisions between those two objects from generating callback events. You can use the getCollisionCallbackDisabledPairs() function to retrieve a list of all the pairs of objects which do not generate a callback event when they collide. (The layout of the output below has been edited for clarity).

```
trace(member("Physics").getCollisionCallbackDisabledPairs())
 -- [[rigidBody("room"),  rigidBody("ball1")],
     [rigidBody("room"),  rigidBody("false floor")],
     [rigidBody("ball1"), rigidBody("false floor")],
     [rigidBody("ball2"), rigidBody("false floor")]]
```

You may notice that this list includes explicit mention of the pair `[rigidBody("room"),  rigidBody("false floor")]`. This mentioning of the pair is in fact unnecessary, since both these objects are static.

Note: *Now that you have experimented with all the methods for controlling collision callback events, you may realize that you can replace the last nine lines of the mInitialize() handler with the following lines:*

```
me.mCreateFalseFloor(vRadius, vSize)
pPhysics.registerCollisionCallback(#FilterCollision, me)
pPhysics.enableCollisionCallback(pCueRB)
```

This code will lead to the most efficient generation of callback events for the movie.

Disabling collisions

In your own projects, there may be situations where you want the Physics simulation to ignore collision detection altogether. You can use physicsMember.disableCollision() for this purpose.

For the experiments below, do not press any modifier keys when starting the BallRoom.dir movie. Relaunch the BallRoom.dir movie, then execute the following command in the Message window:

```
member("Physics").disableCollision()
```

You will see the ball sinking into the floor as collisions are no longer detected between any objects. If the balls do not completely disappear, click the red ball, and you will see the white cue ball pass through it, and then through the wall of the room.

Restart the movie and then execute the following commands in the Message window:

```
vBall1 = member("Physics").getRigidBody("ball1")
member("Physics").disableCollision(vBall1)
```

This code disables collision detection between the red ball and all other objects in the scene. If the red ball does not sink entirely through the floor, click on it. The white cue ball will pass straight through it, and then bounce off the wall.

Restart the movie and then execute the following commands in the Message window:

```
vBall1 = member("Physics").getRigidBody("ball1")
  vRoom = member("Physics").getRigidBody("room")
  member("Physics").disableCollision(vBall1, vRoom)
```

This code disables collision detection only between the red ball and the room. Collisions between the white ball and the red ball will still be detected. What do you predict will happen when you click on the red ball?

Collision disabled pairs

You can use physicsMember.getCollisionDisabledPairs() to retrieve a list of the pairs of objects for which you have explicitly disabled collision management using `disableCollision()`. Try the following command in the Message window.

```
trace (member("Physics").getCollisionDisabledPairs())
-- [[rigidBody("room"), rigidBody("ball1")]]
```

Changing the callback handler

You can only have one registered collision callback handler at a time. You can change the callback handler by registering a new one. For example, launch the BallRoom.dir movie, then execute the following command in the Message window:

```
member("Physics").registerCollisionCallback(#TestCallback)
```

This code will make the `on TestCallback()` in the Test Callback movie script take over the handling of callback events.

If you want to stop all callback event detection completely, you can use physicsMember.removeCollisionCallback(). This function stops triggering callbacks to the handler registered earlier. To test this behavior, execute the following command in the Message window:

```
member("Physics").removeCollisionCallback()
```

Hold the Option/Alt key down and click on the red ball. No collision ContactReports will appear in the Message window, and you will see no Foul warning. (You will still lose a point each time you click, because the scoring system will not be told of any valid interactions before the balls both come to a halt. See the `mStartScoring()`, `mCheckForSleep()` and `mCheckCollisions()` handlers for details on how the scoring is treated.)

Joints and springs

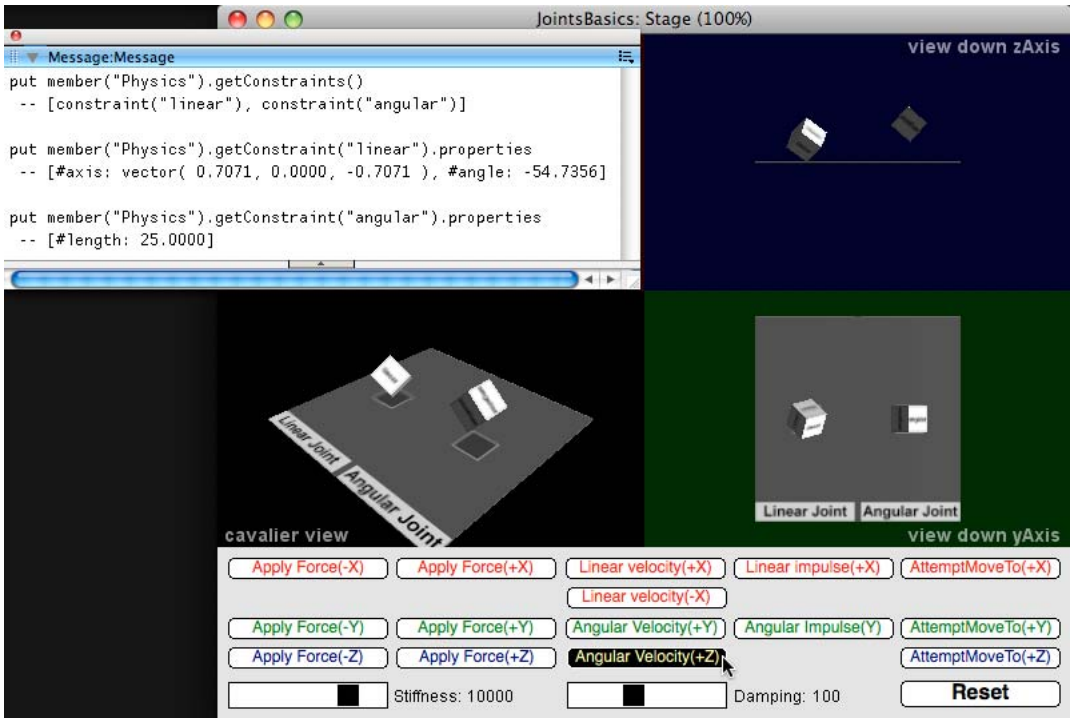
The Dynamiks xtra provides for four different ways to join to objects together in a way that mutually constrains their motion.

- Angular joints allow objects to rotate freely to any angle, but constrain the relative linear movements of two objects. See "Angular joint properties" on page 334 for details on how to modify an angular joint after you have created it.
- Linear joints allow objects to move freely in any linear direction, but constrain the relative rotations of two objects. See "Linear joint properties" on page 335 for details on how to modify a linear joint after you have created it.
- Springs create a compression or an expansion force between two objects which tend to move them until they are at a given resting distance from each other. See "Spring properties" on page 336 for details on how to modify a spring after you have created it.
- "D6Joints" on page 338 allow you to control movement of two different types along 3 different axes giving six degrees of freedom.

Note: *This article deals only with angular joints, linear joints, and springs. For details of D6Joints, see "D6Joints" on page 338.*

Although they share many characteristics, springs are treated as one kind of object (spring) and angular and linear joints are treated as a different kind of object (constraint). The terms "joint" and "constraint" both refer to the angular and linear joint objects.

To compare the behavior of Linear constraints and Angular constraints, download and launch the movie JointsBasics.dir.



The JointsBasics.dir movie allows you to compare the behavior of Linear and Angular Joints

ConstraintDesc

Before you can create any of these three types of controls, you will need to create a ConstraintDesc object using the ConstraintDesc() function. You then use the ConstraintDesc object to create the Linear Joint, Angular Joint, or Spring. The `ConstraintDesc()` function requires seven items of data:

- `name`: a unique string, not used by any other constraint
- `objectA`: a rigidBody object, to connect to one end of the constraint
- `objectB`: a rigidBody or `VOID` or `null` to connect to the other end of the constraint. If `VOID` or `null` are used then the objectA rigidBody will be connected to a fixed point in world space.
- `pointA`: a vector position in the frame of reference of the objectA rigidBody to which the constraint is to be attached.
- `pointB`: a vector position in the frame of reference of the objectB rigidBody, or in world space, to which the constraint is to be attached.
- `stiffness`: a non-negative floating point number to define the *stiffness* of the constraint.
- `damping`: a non-negative floating point number to define the *damping* of the constraint.

Example

```
vRigidBody  = member("Physics").getRigidBodies()[1]
vWorldPoint = vector(0, 0, 0)
vStiffness  = 1234.5
vDamping    = 321.0
vConstraintDescriptor = ConstraintDesc("Test Constraint", vRigidBody, null,
vRigidBody.position, vWorldPoint, vStiffness, vDamping)
trace(vConstraintDescriptor)
 -- (Constraint Descriptor)
```

Creating the constraint

You can use the following functions to create the three different types of constraints:

Note: *Each function requires different parameters in addition to a ConstraintDesc object.*

- physicsMember.createAngularJoint() requires a floating point rest length for the constraint
- physicsMember.createLinearJoint() requires an axisAngle list, containing an axis vector and a scalar angle
- physicsMember.createSpring() requires both a symbol force exertion mode and a floating point rest length for the constraint. The force exertion mode symbol can take one of three values: `#kDuringCompression`, `#kDuringExpansion`, or `#kBoth`.

Examples

In the examples below, it is assumed that a `ConstraintDesc`, using a unique name, has already been created, as described in the previous section. Although a `ConstraintDesc` is an independent object, a script error occurs if you try to use the same ConstraintDesc for two different controls. This is because the name for the control is stored in the ConstraintDesc, and each control must have a unique name.

The first example shows the creation of an angular joint with a rest length of `23.59` world units.

```
vRestLength = 23.59
vAngularJoint = member("Physics").createAngularJoint(vAngularDescriptor, vRestLength)
trace(vAngularJoint)
 -- constraint("angular")
trace(vAngularJoint.constraintType)
 -- #angular
trace(vAngularJoint.properties) -- [#length: 23.5900]
```

The next example shows the creation of a linear joint with a random orientation.

```
vAxis = randomVector()
vAngle = random(360)
vOrientation = [vAxis, vAngle]
vLinearJoint = member("Physics").createLinearJoint(vLinearDescriptor, vOrientation)
trace(vLinearJoint)
 -- constraint("linear")
trace(vLinearJoint.constraintType)
 -- #linear
trace(vLinearJoint.properties)
 -- [#axis: vector( 0.9598, 0.0499, -0.2761 ), #angle: 56.0000]
```

The last example shows the creation of a spring which will resist both expansion and contraction, and which has a rest length of 17.23.

```
vForceExertionMode = #kBoth
vRestLength = 17.23
vSpring = member("Physics").createSpring(vSpringDescriptor, vForceExertionMode, vRestLength)
trace(vSpring)
 -- spring("spring")
trace(vSpring.constraintType)
 -- #spring
trace(vSpring.flags)
 -- #kBoth
trace(vSpring.restLength)
 -- 17.2300
```

Properties: differences and similarities

Because they all require a ContstraintDesc object for their creation, all joints and springs have a shared set of properties: `name`, `objectA`, `objectB`, `pointA`, `pointB`, `stiffness`, and `damping`. They also share a `constraintType` property, which takes the value `#angular`, `#linear`, or `#spring`.

Each `constraintType` has other properties to define its type-specific values. Angular joints and linear joints have a `properties` property, which is a property list.

```
trace(vAngularJoint.properties) -- [#length: 23.5900]
trace(vLinearJoint.properties)
 -- [#axis: vector( 0.9598, 0.0499, -0.2761 ), #angle: 56.0000]
```

Attempting to access the non-existant `properties` property of a spring will lead to a script error. For a spring, you must use the `flags` and `restLength` properties.

```
trace(vSpring.flags)
 -- #kBoth
trace(vSpring.restLength)
 -- 17.2300
```

Configurable properties

For angular joints and linear joints, you can only change the values of the `stiffness` and `damping` properties. All the other properties are get-only. If you try to change the value of one of these other properties, a script error occurs. See "Angular joint properties" on page 334 and "Linear joint properties" on page 335 for more details.

For springs, you can set the values of all the properties except `name` and `constraintType`. See "Spring properties" on page 336 for more details.

Accessing joints and springs

You can use the function physicsMember.getConstraint() to obtain a reference to either a spring or a joint as shown below:

```
trace(member("Physics").getConstraint("angular"))
 -- constraint("angular")
trace(member("Physics").getConstraint("linear"))
 -- constraint("linear")
trace(member("Physics").getConstraint("spring"))
 -- spring("spring")
```

Conversely, the physicsMember.getSpring() function will return a reference to a named spring, but will fail to find any constraint objects.

```
trace(member("Physics").getSpring("test constraint"))
 -- spring("test constraint")
trace(member("Physics").getSpring("linear"))
 -- <Void>
trace(member("Physics").getConstraint("linear"))
 -- constraint("linear")
```

For similar reasons, the physicsMember.getConstraints() function will only include joints in the output list.

```
trace(member("Physics").getConstraints())
 -- [constraint("linear"), constraint("angular")]
```

To obtain references to all the springs created within a Physics member, you need to use the physicsMember.getSprings() function.

```
trace(member("Physics").getSprings())
 -- [spring("spring")]
```

Deleting a joint or spring

To delete a joint or a spring, you can use the physicsMember.deleteConstraint() method. You can provide either the name of the object to delete, or a reference to it.

```
trace(member("Physics").getSprings())
 -- [spring("spring")]
trace( member("Physics").deleteConstraint("spring"))
 -- 1
trace(member("Physics").getSprings())
 -- []
vAngularJoint = member("Physics").getConstraint("angular")
trace(vAngularJoint)
 -- constraint("angular")
trace( member("Physics").deleteConstraint(vAngularJoint))
 -- 1
trace(vAngularJoint)
 -- <Void>
```

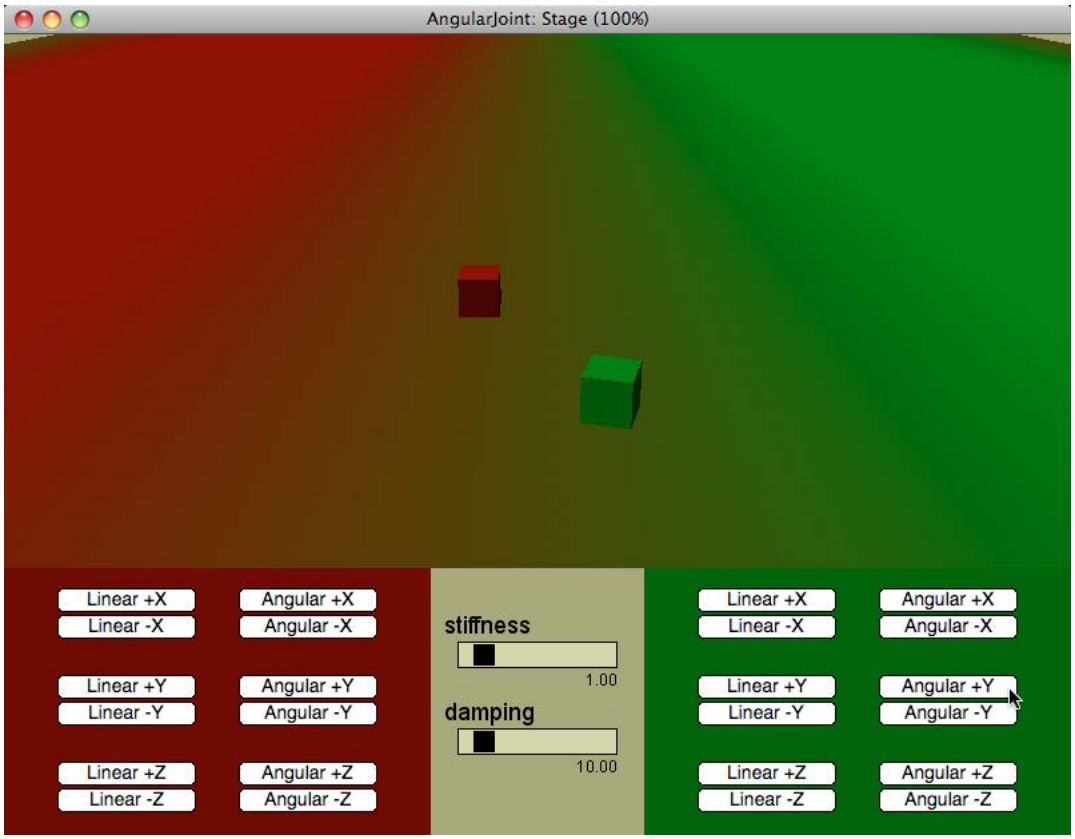
Note: *If you conserved a reference to the ConstraintDesc that you used to create a joint or a spring, the ConstraintDesc will now be available for use with another joint or spring.*

For only springs, you can use the physicsMember.deleteSpring() method. If you attempt to use this method with an object that is not a spring, a script error occurs. It will fail silently if you use it with a string that is not the name of a spring.

Angular joint properties

An angular joint between two objects allows those objects to rotate freely to any angle with respect to each other. However, the distance that separates the two objects is constrained.

For information on how to create, access, and delete an angular joint, see "Joints and springs" on page 330. To experiment with the way an angular joint will link two rigid objects together, download the movie AngularJoint.dir.



The AngularJoint.dir movie shows how the movement of two joined objects is constrained

You can use the buttons in the red zone to move the red box and the buttons in the green zone to move the green box. Note how the distance between the two boxes is maintained, but they are free to rotate independently.

You can alter the stiffness and damping of the angular joint. These values are not limited to `10.0` or `100.0`, despite what the movie shows. These values have been chosen as suitable values for testing in the context of this movie. The most appropriate values for your projects will depend on the mass of the objects, their friction, and other properties.

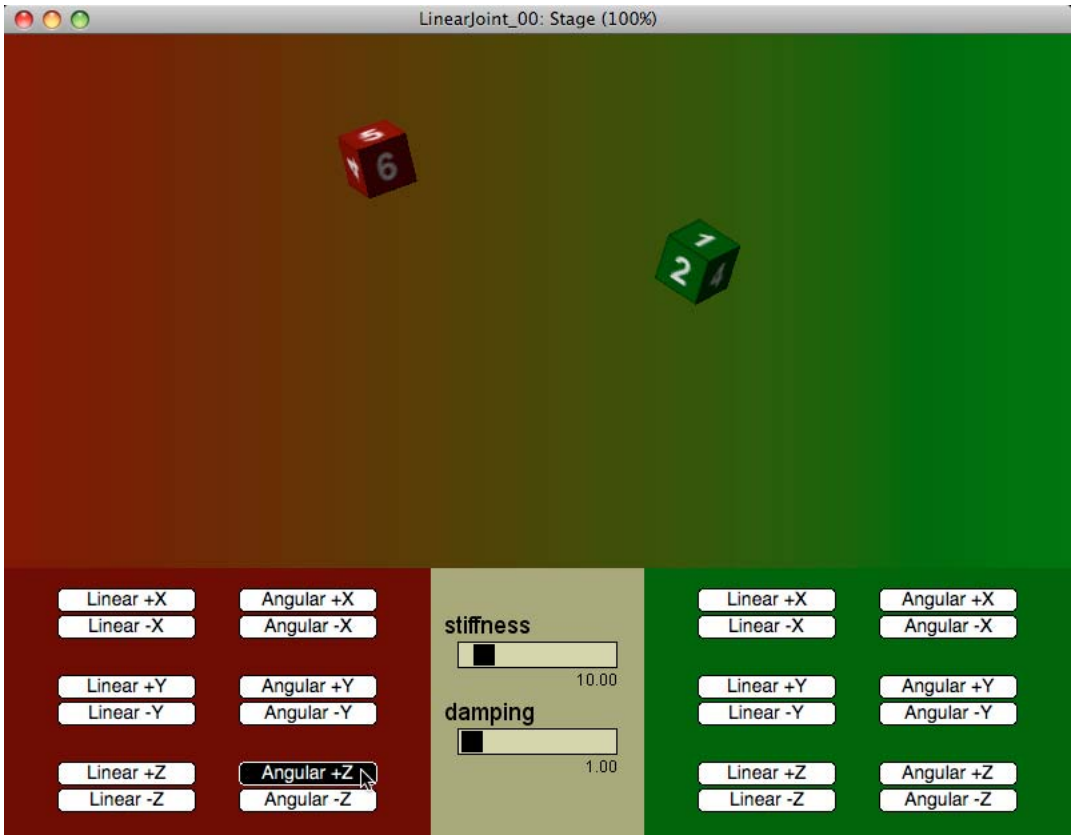
A low value for `stiffness` will allow the objects to move away from their rest-length distance more easily. A low value for `damping` will lead to objects oscillating about their rest-length distance.

Note: *In Director 11.5, if you reduce* `damping`*, you may see no change in the behavior of the angular joint until you modify the* `stiffness`*.*

Linear joint properties

A linear joint between two objects allows those objects to move freely to any spatial position with respect to each other. However, the difference in orientation of the two objects is constrained.

For information on how to create, access, and delete a linear joint, see "Joints and springs" on page 330. To experiment with the way a linear joint will link two rigid objects together, download the movie LinearJoint.dir.



The LinearJoint.dir movie shows how the movement of two joined objects is constrained

You can use the buttons in the red zone to move the red box and the buttons in the green zone to move the green box. Note how the two boxes have a tendency to appear in the same orientation, but they are free to move independently around the surface.

Note: *To test rotation around the xAxis and zAxis, double-click on the appropriate buttons. The first click will make the cube jump into the air, and the second click will make it spin in the air.*

You can alter the stiffness and damping of the linear joint. These values are not limited to `100.0`, despite what the movie shows. This range has been chosen as suitable values for testing in the context of this movie. The most appropriate values for your projects will depend on the mass of the objects, their `angularDamping`, and other properties.

A low value for `stiffness` will allow the objects twist to different orientations more easily. A low value for `damping` will lead to the objects oscillating about their shared orientation.

Note: *In Director 11.5, if setting both `stiffness` and `damping` to zero does not lead to the objects becoming independent.*

Spring properties

Springs create a compression or an expansion force between two objects which tend to move them until they are at a given resting distance from each other.

For information on how to create, access and delete a spring, see "Joints and springs" on page 330. To experiment with the way a spring will link two rigid objects together, download the Springs Basics.dir.

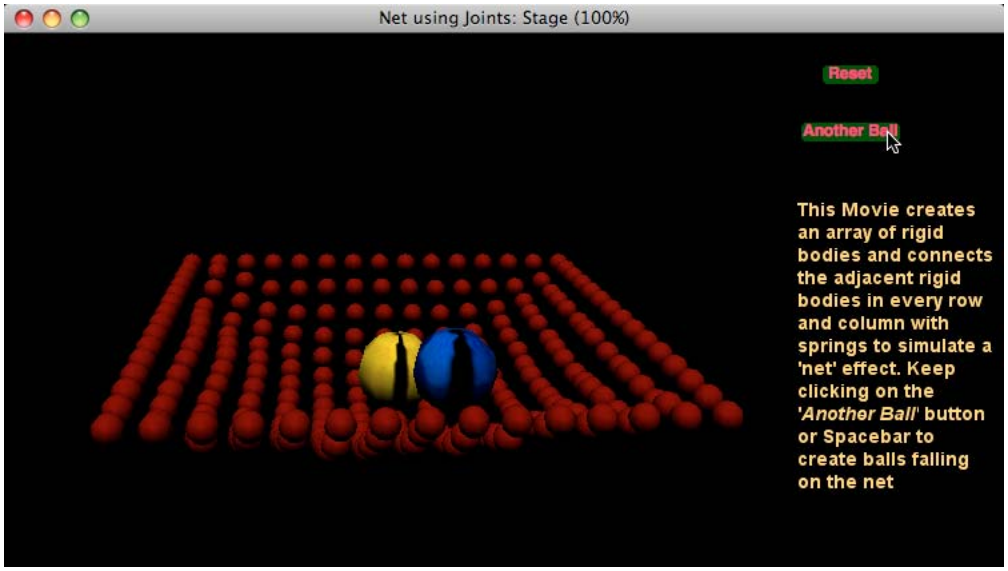
The Springs Basics.dir movie shows how a spring can be controlled

The Springs Basics.dir movie has seven screens where you can test the effect of modifying all eight parameters for a spring.

- **`restLength`** defines the distance between the models at which neither a compression nor an expansion force will act between the two models.
- **`objectA`** and **`objectB`** define the rigidBody objects connected to each end of the constraint.
- **`pointA`** and **`pointB`** are the vector positions in the frame of reference `objectA` and `objectB` to which the constraint is to be attached.
- **`stiffness`** is a non-negative floating point number to define the strength of the spring.
- **`damping`** is a non-negative floating point number to define how much the spring must oscillate before reaching a rest position.
- **`flags`** is a symbol which defines whether the spring must exert a force only when it is compressed (`#kDuringCompression`), only when it is extended (`#kDuringExpansion`), or in both directions (`#kBoth`) .

Additional example

The movie Net using Joints.dir demonstrates how to create a net out of a grid of rigidBody objects held together by springs.



A net created from rigidBody objects and springs

D6Joints

Six-degrees-of-freedom joints (D6Joints) are a very powerful tool for creating mechanical interactions between objects. The six degrees of freedom are:

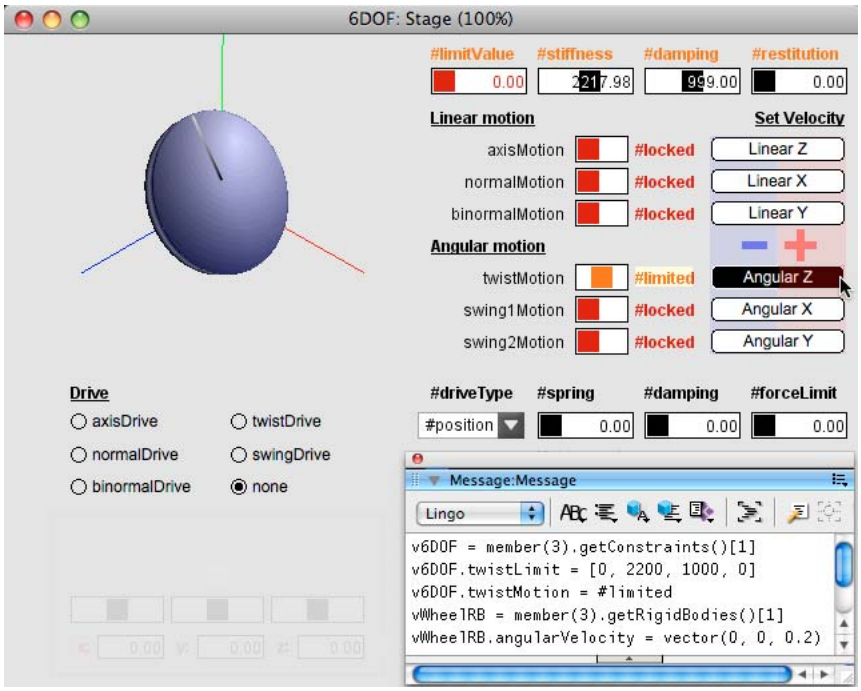
- Linear movement along the xAxis
- Linear movement along the yAxis
- Linear movement along the zAxis
- Rotation about the xAxis
- Rotation about the yAxis
- Rotation about the zAxis

Each of these types of movement can be locked, limited, or left free.

In addition, you can *drive* a joint, as if it were attached to a motor. This means that you can make the joint turn about any of its axes, or force it to advance along any axis. This drive can be controlled and limited in a variety of ways.

Experimenting

The best way to understand all the possibilities of a D6Joint is to experiment with a simple example. Download and launch the movie 6DOF.dir. You can use this to test each of the motions possible with a D6Joint. You can find details of the scripting terms to use in "D6Joint method and properties" on page 343.

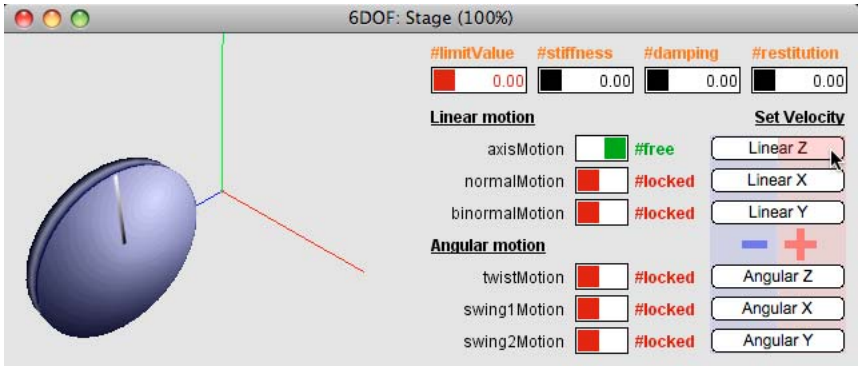


Using the 6DOF.dir movie to test the settings for a six-degrees-of-freedom joint

The 6DOF.dir movie is designed with a simple interface that gives you full control of only one type of motion at a time. Note that many of the sliders in the movie have a logarithmic scale. This means that you can test a whole range of values from very small to very large, but you cannot select precise values.

In your own projects, you can control all six types of motion independently. After you have learned enough to feel comfortable with controlling a D6Joint, you can create custom settings for the test joint by typing commands in the Message window. The custom settings will allow you both to set precise values for each motion, and to control multiple motions simultaneously. Changes that you make in the Message window will not be reflected in the display in the movie itself.

When you launch the movie, all six degrees of freedom are locked. You can start by freeing the motions one at a time, and testing which direction of movement the joint allows for that motion.

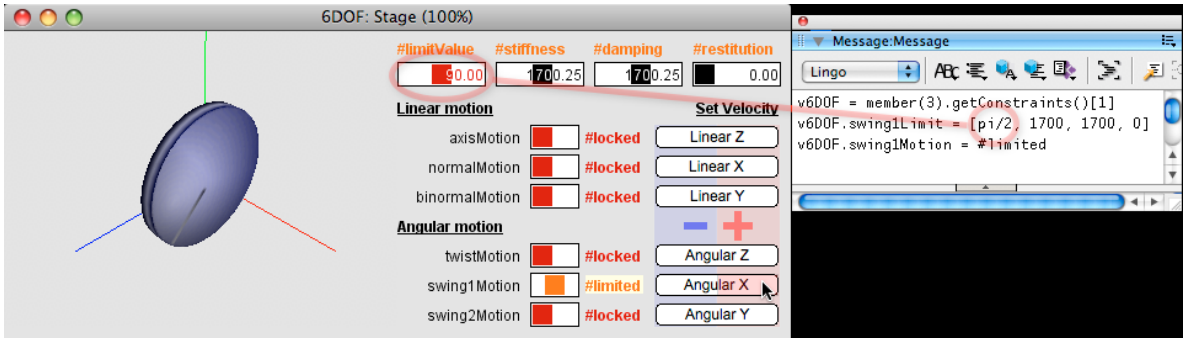


Testing one degree of freedom at a time

Note that the main axis of a joint is its zAxis, so the controls for the zAxis have been placed at the top of each group of controls.

Limited freedom

You must set the limits of a freedom before setting the motion of that freedom to `#limited`. If you set give a particular motion limited freedom when the limits are all set to zero, you will see no change when you modify the value of the limits. If you set any of the limits to a non-zero value, then set the motion of a given freedom to `#limited`, any changes that you make to the limits will be respected. If you set all limits to zero, then the motion will become locked again, regardless of any new limit settings that you apply. You will need to reset the motion to `#limited` for any new changes to be taken into account.



The limitValue for angular motions has to be expressed in radians. The interface shows it in degrees.

You can double-click on any slider to reset it to its initial value.

Note that the `stiffness` is value of `0`, which means that the joint is completely rigid. A stiffness value of `0.00001` is extremely loose, whereas as value of `100000` gives similar behavior to a value of `0.0`. If `stiffness` is zero and `limitValue` is zero, then no movement will occur.

If `stiffness` is low, a motion may exceed its `limitValue` before easing back. If damping is also low, a freedom with `#limited` motion may behave almost as if it were free.

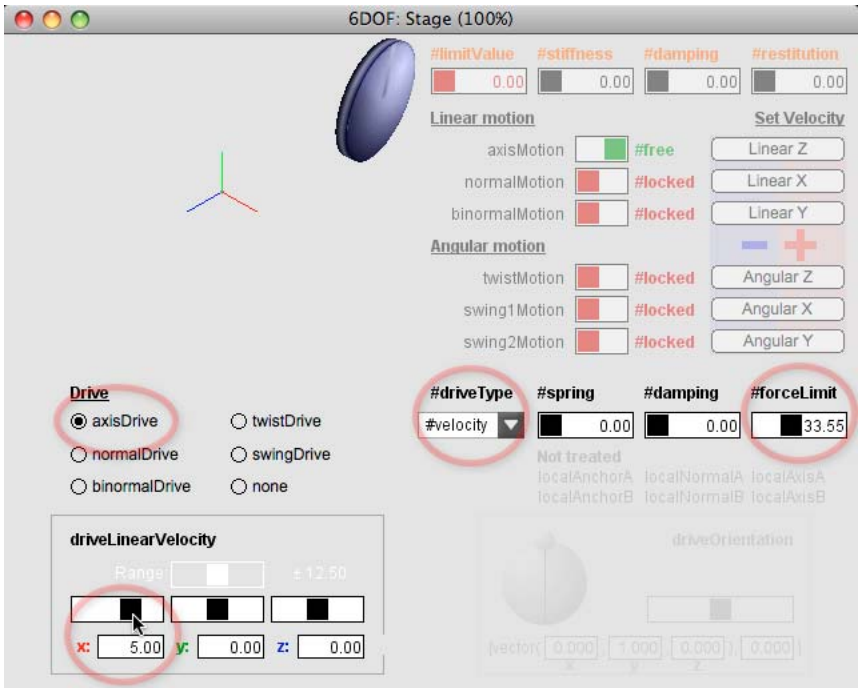
Note: *To simplify the interface, the display for `#limitValue` can vary between `0.0` and `180.0`. This gives a good range for both linear and angular motion, if angular motion were considered in degrees. However, angular limits must be in the range `0.0 - 3.14159` (or `pi`). Angular limits need to be defined in radians, not degrees.*

Drive

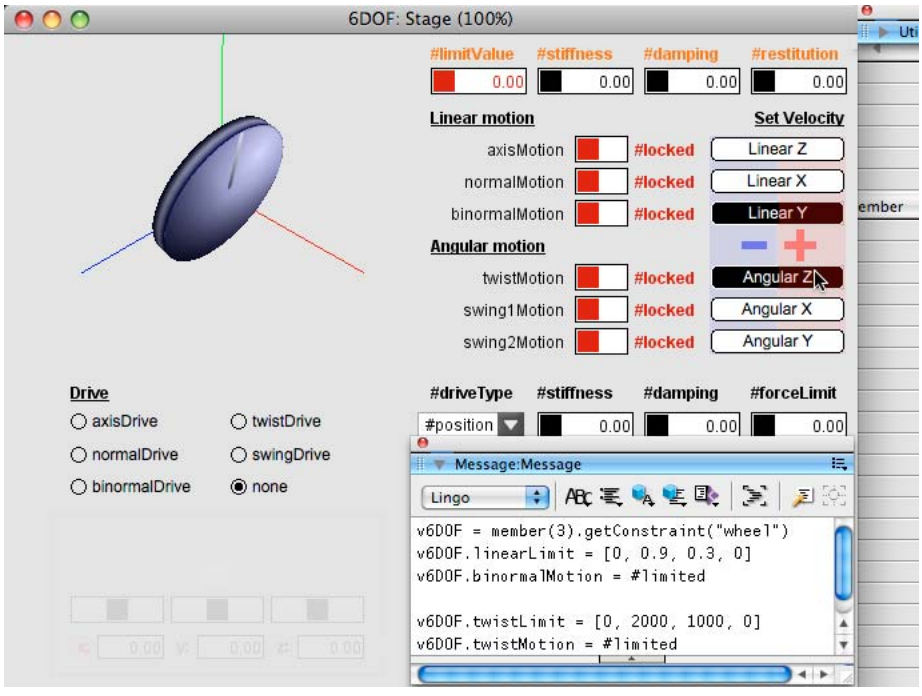
A D6Joint can be driven along each degree of freedom, which means you can provide both a linear and an angular motion along each of the three axes. For each driven motion there are two types of drives. So, there are twelve possibilities of which any six can be active at one time. The values for `driveType` are as follows:

- `#position`: attempts to move the joint to a given position or orientation. An example is a piston, which pushes in a given direction, or a door closer that rotates a door back to its closed position after it has been opened.
- `#velocity`: attempts to move the joint at a constant speed in a given direction. An example is a train wheel. For a wheel rigidBody, like the one in the 6DOF.dir movie, you can set the joint's `axisDrive` to slide the wheel along the track, or you can set its `swingDrive` to rotate it around its zAxis.

These examples are illustrated below:



Using a #position driveType to set D6Joint to a given distance along an axis, or a given rotation around an axis



Using a #velocity driveType to move a joint along an axis or rotate it around an axis, continuously

When a `driveType` of `#velocity` is used, the value of `forceLimit` determines how fast the joint will accelerate until it reaches the required speed. If `forceLimit` is set to zero, the drive will never move. If it is too low, it may take a very long time to get up to speed.

When a `driveType` of `#position` is used, the value of `stiffness` determines how responsive the joint is to the request to move to the desired position. If `stiffness` is set too low, the effect will be minimal, like attempting to pull a train using an infinitely extensible elastic band. The precise value will depend on the mass of the rigidBody.

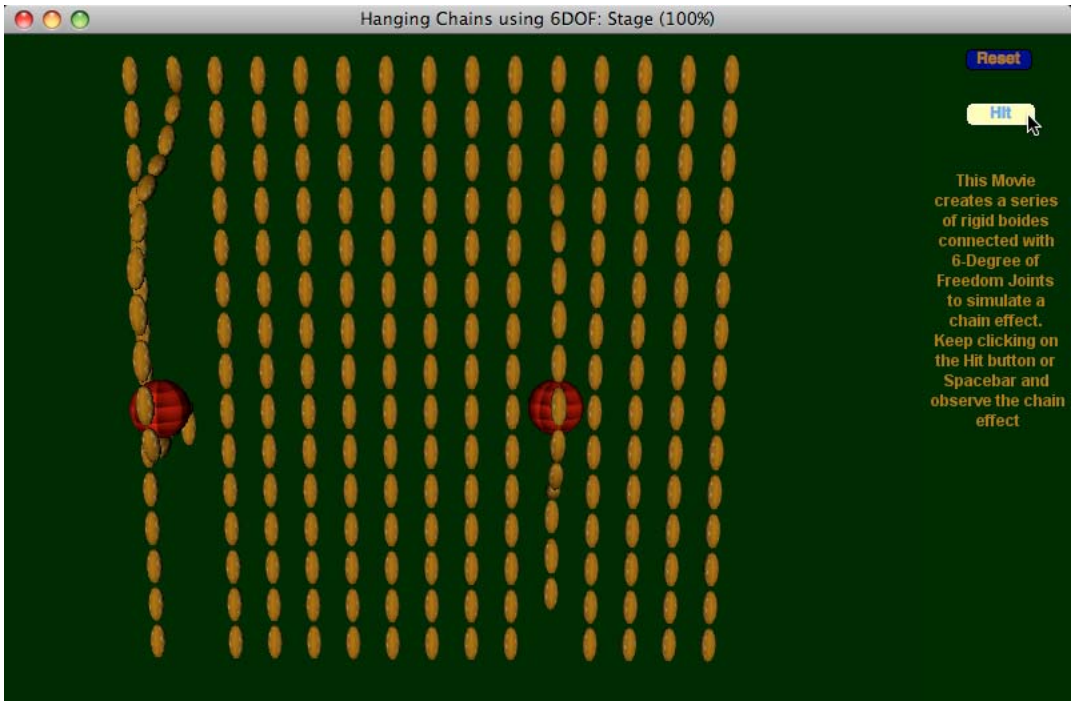
The direction axis that you need to set to move a joint along or around a given axis is not intuitive. For linear movements:

- **axisDrive** uses the x component of the driveLinearVelocity, or the drivePosition vectors to move the joint along the **negative zAxis**
- **normalDrive** uses the **y** component of the driveLinearVelocity or the drivePosition vectors to move the joint along the **negative xAxis**
- **binormalDrive** uses the **z** component of the driveLinearVelocity or the drivePosition vectors to move the joint along the **negative yAxis**
- **twistDrive** uses the **x** component of the driveAngularVelocity vector to rotate the joint around the **positive zAxis**
- **twistDrive** uses rotation around **x** axis of the driveOrientation to rotate the joint around the **negative zAxis**
- **swingDrive** uses the **y** component of the driveAngularVelocity vector to rotate the joint around the **positive xAxis**
- **swingDrive** uses rotation around **y** axis of the driveOrientation to rotate the joint along the **negative xAxis**
- **swingDrive** uses the **z** component of the driveAngularVelocity vector to rotate the joint along the **positive yAxis**
- **swingDrive** uses rotation around **z** axis of the driveOrientation to rotate the joint along the **negative yAxis**

Note also that swingDrive can perform rotation around two different axes at the same time, if both swing1Motion and swing2Motion are not locked. In 6DOF.dir, these rotations create a movement like a coin spinning on a table (free), or like a buoy in a harbour, rocked by waves (limited).

Additional example

You can find another example of using D6Joints in the movie Hanging Chains using 6DOF.dir.



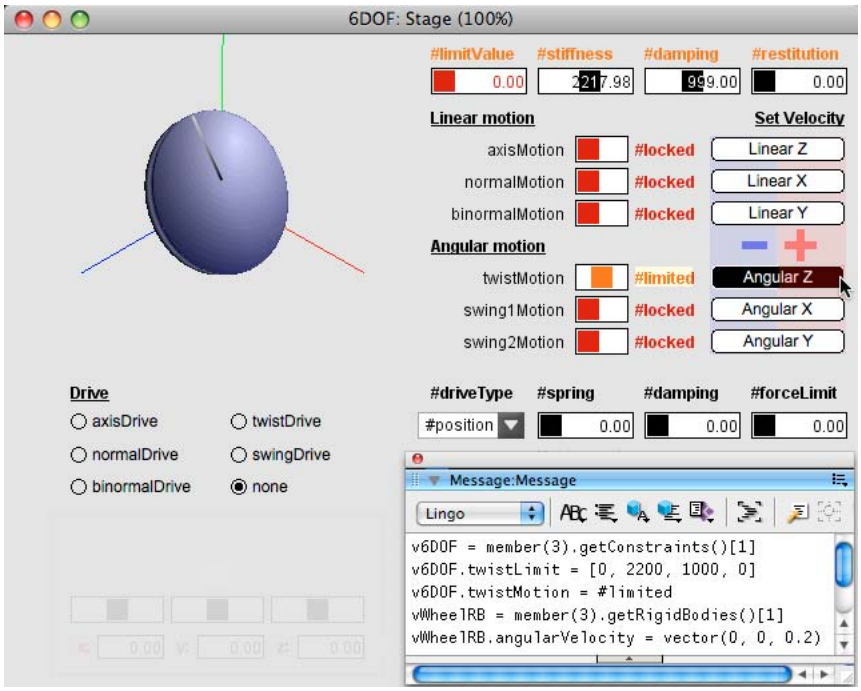
The hanging chains are created by D6Joints locked for any linear motion and free for all angular motion

D6Joint method and properties

Six degrees of freedom joints have one method, and many properties. To make them easier to learn, the properties are grouped into the following categories:

- Get-only properties
- Anchor properties
- Freedom properties
- Limit properties
- Drive properties
- Drive target properties
- Axis properties

To test all these properties, download, and launch the movie 6DOF.dir. See "D6Joints" on page 338 for detailed instructions for using this movie.



The 6DOF.dir movie allows you to experiment with a simple D6Joint

createD6Joint() method

To create a joint with six degrees of freedom, use the physicsMember.createD6Joint() method.

Note: *Before creating the joint, align the two rigidBody objects with the world's axes, and place them in the appropriate position relative to each other. When you do so, controlling the axes, along which the various motions can occur, becomes easier.*

Following is the command with the `createD6Joint()` method:

```
vJoint = vPhysics.createD6Joint(vName, vRB1, vRB2, vAnchor)
```

This method creates a D6Joint between two rigid bodies, or between a point in world and a rigid body, and expects the following values:

- `vPhysics`: a Physics member
- `vName`: a unique string, not yet used for the name of any joint in the Physics member
- `vRB1`: a rigidBody stored within the Physics member
- `vRB2`: a different rigidBody stored within the Physics member, or `VOID`
- `vAnchor`: a position vector defining the point in world space where the joint will be located. The position of this point will be converted into the local frame of reference of each of the rigidBodies. See "Anchor properties" on page 345 for details.

Note: *The order in which rigidbodies are provided as input to a D6Joint is important with respect to the set joint axis and the drive values. For example, if the order is rb1, rb2, and the drive-position is vector(10,0,0), the direction in which the rigid body will move to attain the position is exactly opposite to the direction it will move if the order was rb2, rb1.*

To delete a D6Joint, use physicsMember.deleteConstraint().

Get-only properties

A D6Joint object has five properties whose value is set at the moment that you call createD6Joint(), and which cannot be changed later. Attempting to alter the value of these properties will lead to a script error.

- constraint.name
- d6joint.objectA
- d6joint.objectB
- constraint.constraintType (D6Joints are a type of constraint, just like linear and angular joints. The `constraintType` for D6Joint is `#d6joint`)

Anchor properties

A D6Joint object has three properties that define the mutual frame of reference of the joined rigidBody objects.

- d6joint.globalAnchor
- d6joint.localAnchorA
- d6joint.localAnchorB

The Physics simulation uses world space as a frame of reference for all objects that it controls. When you create a D6Joint, you define a position in world space as the `globalAnchor` point for the joint. The two rigidBody objects now share a mutual point of reference: the `globalAnchor`. Each of the rigidBody objects joined by the joint converts that worldPosition point into its own local frame of reference. These references are stored in the `localAnchorA` (for the first named rigidBody) and localAnchorB (for the second rigidBody).

If the rigidBody objects now move together, with respect to the world, the `worldPosition` of their `globalAnchor` will change. It is "global" only in the sense that it is shared by both rigidBodies in the joint. Each rigidBody will maintain its own position and orientation relative to the shared `globalAnchor` point.

If you alter the values of `localAnchorA` or `localAnchorB` the associated rigidBody will move to a new position with the respect to the mutual `globalAnchor` point.

If you change the value of the `globalAnchor` the values of `localAnchorA` or `localAnchorB` will update to reflect the current positions of the two rigidBody objects with respect to the new `globalAnchor` point.

Note: *If you create a joint with only one rigidBody and `VOID` as the other party to the joint, then changing the `globalAnchor` point will affect the center of rotation of the sole rigidBody.*

Freedom properties

You can set the freedom of each of the six types of motion available to a D6Joint. The following properties can take the values `#locked`, `#limited`, or `#free`:

- d6joint.axisMotion: refers to linear motion along the main zAxis
- d6joint.normalMotion: refers to linear motion along the xAxis
- d6joint.biNormalMotion: refers to linear motion along the yAxis
- d6joint.twistMotion: refers to angular motion around the main zAxis
- d6joint.swing1Motion: refers to angular motion around the xAxis
- d6joint.swing2Motion: refers to angular motion around the yAxis

See "D6Joints" on page 338 for a demonstration of these different motions.

A value of `#locked` means that the joined rigidBodies will move together for that motion. If one rigidBody is joined to a fixed point in the world, a value of `#locked` means that the rigidBody will not move or rotate on the given axis.

A value of `#free` means that there is no constraint for movement on the given axis. Details for the way a value of `#limited` constrains movement are given in the sections below.

Limit properties

A D6Joint object has four properties that can be used to limit the movement of the joint in different ways. Linear movement on all three axes is controlled by a single property. The other three properties deal with angular movement around each of the axes.

- d6joint.linearLimit defines how linear movement is limited on all axes
- d6joint.twistLimit defines how angular movement is limited around the main zAxis
- d6joint.swing1Limit defines how angular movement is limited around the xAxis
- d6joint.swing2Limit defines how angular movement is limited around the yAxis

All of these properties take a value that is a property list with the following structure:

```
[#limitValue: <float>, #stiffness: <float>, #damping: <float>, #restitution: <float>]
```

For the `linearLimit` property, limitValue is defined in world units. For the angular limit properties, it is defined in radians. You can limit angular movement to a maximum of half a turn in either direction, or `±pi` radians. This means that the limitValue for the angular properties must be in the range `0.0 - 3.14159265358979`. If you use a value that is even a small fraction above `pi`, the entire contents of the list will be ignored.

Note: *In the 6DOF.dir movie, the `#limitValue` is displayed in degrees, not radians, to keep the interface simple,. Using degrees will fail to give you the results you expect.*

For stiffness, a value of `0.0` indicates complete rigidity, while a value of `0.00001` indicates almost total flaccidity.

The default values for all these properties is the same, as shown below.

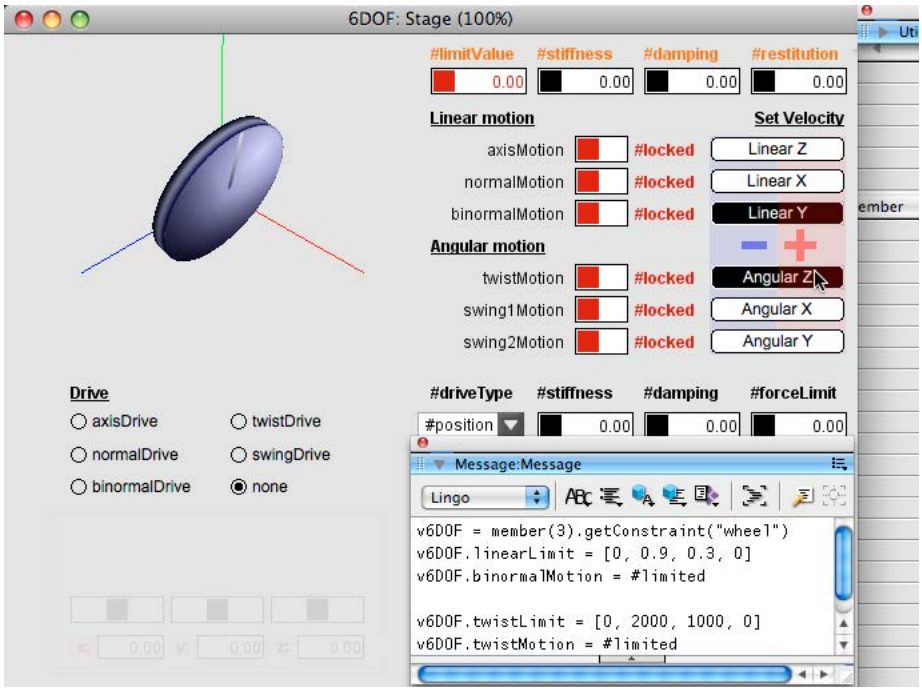
```
[#limitValue:0.0, #stiffness:0.0, #damping:0.0, #restitution:0.0]
```

This means that simply setting (for example) `aD6Joint.axisMotion = #limited` will have the same effect as `aD6Joint.axisMotion = #locked`. Set the appropriate Limit property before setting the required motion to `#limit`.

Relaunch the 6DOF.dir movie, and execute the following commands in the Message window:

```
v6DOF = member(3).getConstraint("wheel")
v6DOF.linearLimit = [0, 0.9, 0.3, 0]
v6DOF.binormalMotion = #limited
v6DOF.twistLimit = [0, 2000, 1000, 0]
v6DOF.twistMotion = #limited
```

Note that the display in the 6DOF.dir movie will not update to reflect these settings.



Using the 6DOF.dir movie to test custom settings

Now click the Linear Y and Angular Z buttons to test these results.

Drive properties

A D6Joint has 5 different properties that can be used to define how the joint will be driven.

- d6joint.axisDrive defines how linear movement along the main zAxis will be driven
- d6joint.normalDrive defines how linear movement along the joint's xAxis will be driven
- d6joint.binormalDrive defines how linear movement along the joint's yAxis will be driven
- d6joint.twistDrive defines how angular movement around the main zAxis will be driven
- d6joint.swingDrive defines how angular movement around both the x- and the yAxis will be driven

All of these properties take a value that is a property list with the following structure:

```
[#driveType: <symbol>, #stiffness: <float>, #damping: <float>, #forceLimit: <float>]
```

You can also use a linear list where the values are arranged in the same order as in the property list above.

driveType

The values for `#driveType` are as follows:

- `#position`: attempts to move the joint to a given position or orientation.
- `#velocity`: attempts to move the joint at a constant speed in a given direction.

See "Drive" on page 340 for examples and illustrations.

forceLimit

When a `driveType` of `#velocity` is used, the value of `forceLimit` determines how fast the joint will accelerate until it reaches the required speed. If `forceLimit` is set to zero, the drive will never move.

stiffness

When a `driveType` of `#position` is used, the value of `stiffness` determines how responsive the joint is to the request to move to the desired position. Be careful not to set stiffness too low.

damping

When a `driveType` of `#position` is used, the value of damping determines how much passive resistance there is to a change in position or orientation. If `stiffness` is low and `damping` is low, the joint will oscillate around the target position after a sudden change in position. If `stiffness` is low and `damping` is high, then the joint may fail to move at all.

Drive target properties

In addition to setting the drive characteristics, you will need to set a target velocity or position. The time it takes to reach these targets will depend on the drive properties, as described above.

- d6joint.driveLinearVelocity: vector used with `axisDrive`, `normalDrive` and `binormalDrive` to define a target speed and direction.
- d6joint.drivePosition: vector used with `axisDrive`, `normalDrive` and `binormalDrive` to define a target position.
- d6joint.driveAngularVelocity: vector used with `twistDrive` and `swingDrive` to define a target rotation velocity
- d6joint.driveOrientation: axisAngle list used with `twistDrive` and `swingDrive` to define a target orientation.

Axis properties

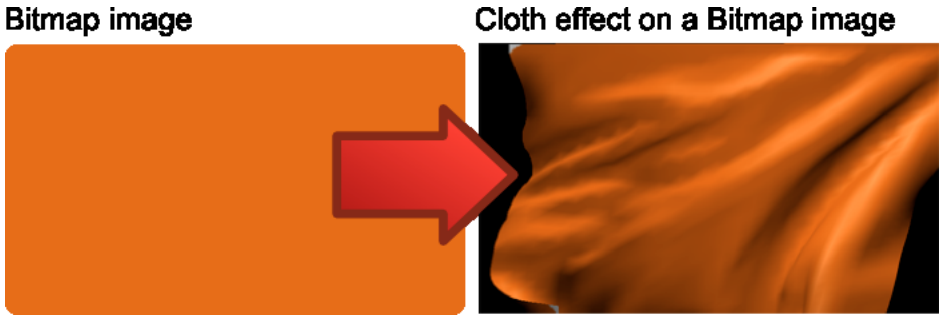
- d6joint.localAxisA
- d6joint.localAxisB
- d6joint.localNormalA
- d6joint.localNormalB

The movie Hanging Chains using 6DOF.dir contains code which alters the values of these properties. However when this code is commented out (as it is now) the movie still behaves in exactly the same way.

Cloth

What if you want to place an animated flag or a banner in your game with a realistic cloth effect? You can easily simulate a cloth effect on low density meshes and bitmap images using Director. Director supports enhanced physics including a complex cloth physics simulation using the AGEIA physics engine.

Your bitmap images can easily be transformed into a cloth in the scene using a few lines of code.



A bitmap image can be used as a cloth

Creating the cloth

A cloth can be created using the `createclothresource` function. This function creates a cloth resource from any existing model reference.

Note: *Only models with single mesh can be used to create a cloth resource.*

Here is how you create a cloth resource using Lingo:

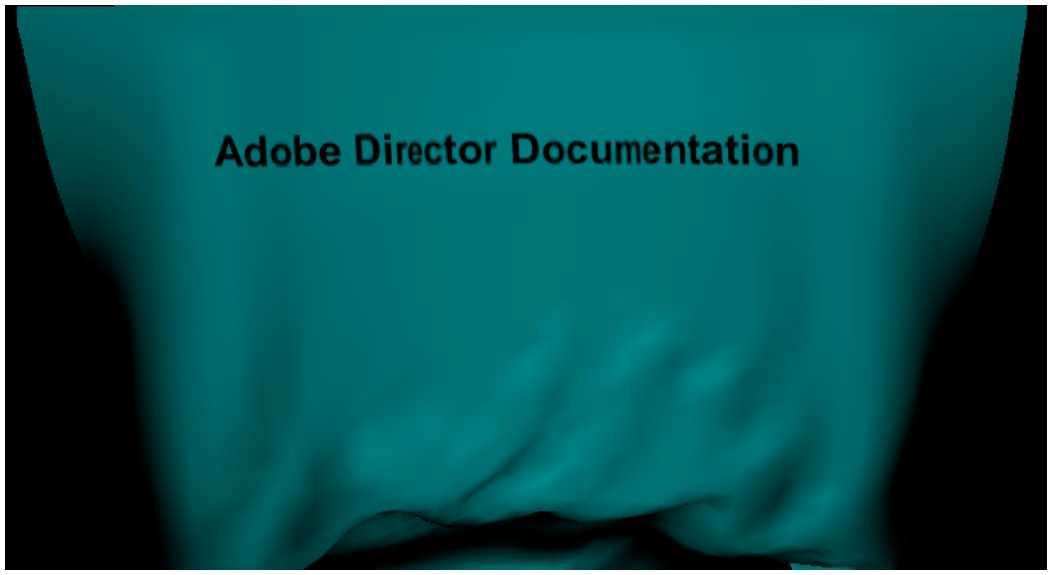
```
member ("physics").createclothresource(modelref, [flipNormals])
```

You can also insert the physics media element by clicking Insert > Media Element > Physics.

The parameters are:

- `modelref`: The reference to the 3D model that will be used as the cloth.
- `flipNormals:` An optional parameter specified to flip the normals in the model. Use the flipNormals flag to invert all the normals after simulation.

During cloth simulation, the normals are calculated according to the order (clockwise/counter-clockwise) in which the triangles in the model are specified. If the calculated normals are not the same as the model's normal list, the model may become invisible.



Cloth simulation

You have successfully created a cloth resource. Now, create a cloth object from that resource. An important point to remember is that only models with a single mesh can be used to create a cloth.

```
member ("physics").createcloth(clothname ,clothres, clothmodel, density,thickness)
```

The parameters are:

- `clothname`: A string that represents the name of the cloth.
- `clothres` - A cloth resource created using the `createclothresource()` method.
- `clothmodel` - The model that was used in creating the cloth resource.
- `density` - The density or the mass per unit area of the cloth. Default value is 1.0. Range is from 0 to infinity.
- `thickness` - The thickness of the cloth. Default value is 0.01. Range is from 0 to infinity. Visual artifacts may appear if thickness is very small or large.

The following Lingo code illustrates the cloth creation example:

```
world = member ("physics")
clothres = world.createclothresource (clothmodel)
clothref = world.createcloth ("banner",clothres,clothmodel, 2.0,1.8)
```

You can also apply a radial force to push or pull the cloth using the applyforceatpos function:

```
cloth.applyforceatpos(positionVector, magnitude, radius, forcemode
```

The parameters are:

- `positionVector`: The position at which the force needs to be applied.
- `magnitude`: The magnitude of the force.
- `radius`: The radius in which all the cloth particles will be affected with a quadratic drop off.
- `forcemode`: Applies either a force or impulse to the `cloth.forcemode` can either be #force or #impulse.

When will you apply a force on the cloth? In you game, if you want the cloth model to wave due to wind coming from a particular direction, you can use the `applyforceatpos` function.

For instance,

```
-- A force of 1000 is applied at the position(0,0,0).
cloth.applyforceatpos(vector(0,0,0),1000, 2, #force)
```

You can also control the direction of the applied force to achieve different effects on the cloth:

```
-- A force of 1000 is applied from the Y-axis at the position(0,0,0).
cloth.applydirectedforceatpos(vector(0,0,0),vector(0,1000,0), 2,#force)
```

To achieve the best effect, you may need to define the pressure element for the cloth. Director allows you to set pressure inside a closed cloth (like a balloon) using the `enablePressure` property. This property has no effect on the open meshes.

To set the pressure for a closed cloth, use this Lingo code:

```
cloth.pressure  = 2.0
```

You can provide a value between 0 to infinity. A value less than 1 will cause the cloth mesh to contract and a value greater than 1 will make the cloth to expand with respect to the rest of the shape. While controlling the pressure inside a closed cloth, you can also set the bending stiffness:

```
cloth.bendingStiffness =  0.5
```

To remove the cloth object from the scene, use the `deletecloth` function:

```
member("physics").deletecloth(clothref)
member("physics").deletecloth(clothname)
```

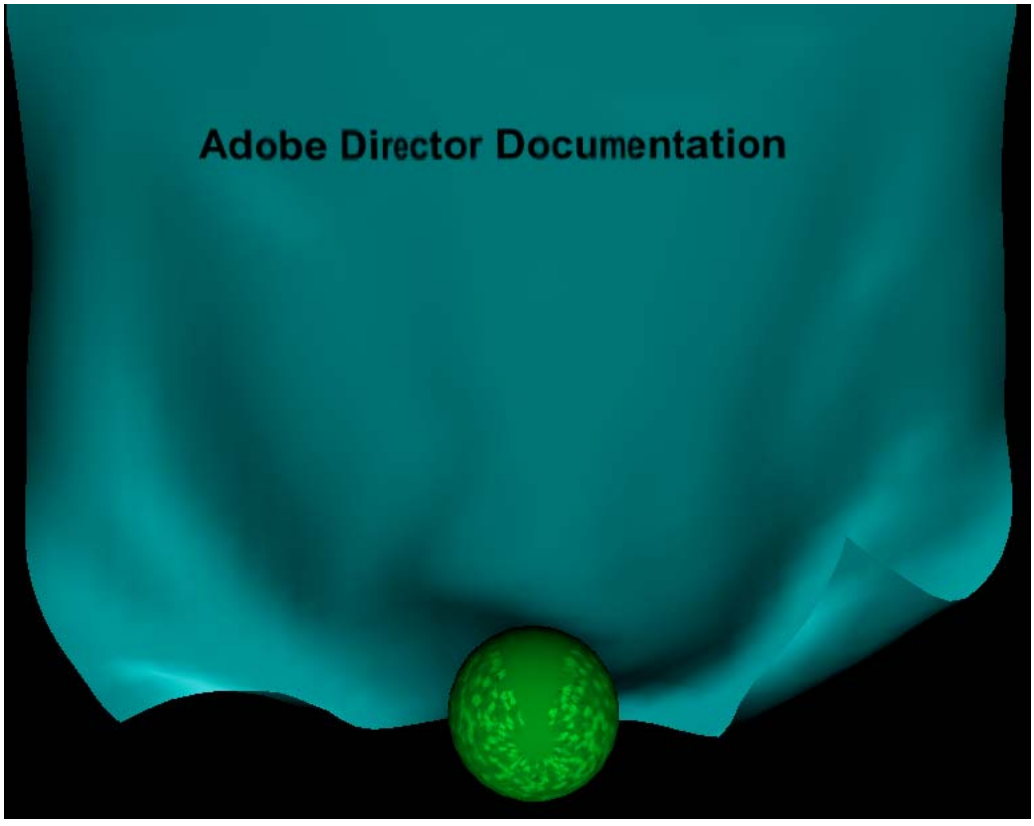
The parameters are:

- `clothref`: The reference to the cloth that needs to be deleted.
- `clothname`: The name of the cloth that needs to be deleted.

Attaching the cloth to a rigid body

The cloth that you create in your game will most likely collide with a rigid body. In that case, how will you make your cloth wrap around the rigid body? A proper collision detection between the cloth that you create and a the rigid body is important.

Director allows you to attach the cloth to a rigid body. However, this method only works with primitive and convex shapes.



Cloth and rigid body collision

For instance, to attach the cloth to a rigid body, "sphere", use the `attachToRigidBody` function:

```
cloth.attachToRigidBody("sphere")
```

Similarly, use the `detachRigidBody` function to detach the cloth from the rigid body.

Sleeping and Waking

When a cloth does not move for a period of time, it is no longer simulated/animated. This state is called the sleeping state of the cloth.

Can you put the cloth to sleep manually? Yes. However, the cloth automatically 'wakes up' when it is either collided with an active object, or when one of its properties is changed by the user.

To put the cloth to the sleeping state, use the Lingo code:

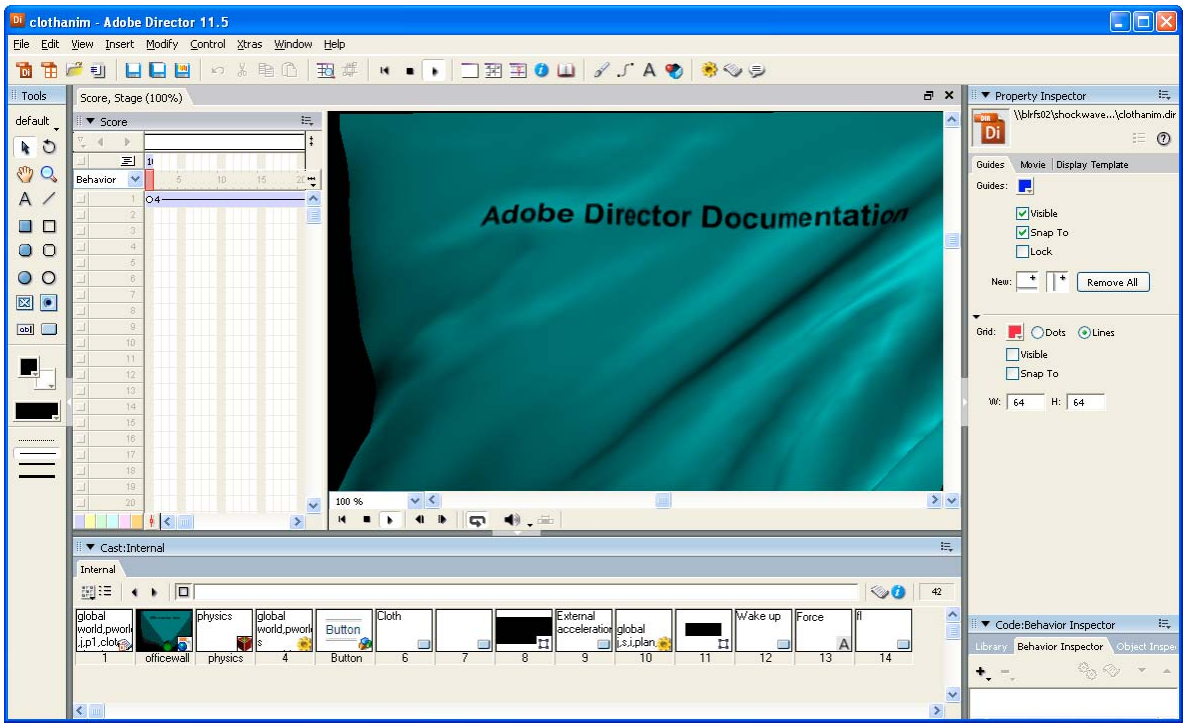
```
cloth.putToSleep()
```

Similarly, you can wake up the cloth using the `wakeup` function:

```
cloth.wakeUp();
```

For more information on working with the Cloth APIs, see:

- Cloth methods
- Cloth properties.



Cloth animation in Director

Download the movie clothanim.dir to understand the cloth animation feature in Director.

Character controller

When you are building a First Person Shooter (FPS) game, you will need the characters to navigate in the 3D world. What should happen if the character encounters a wall? What should happen if the character needs to climb some steps? How will you make the in-game characters interact with the various rigid bodies present in the world? Adobe Director allows you to perform the character controller actions quite effortlessly.

The character controller API in Adobe Director provides the following functionalities:

- You can control the movement of any character.
- You can make your characters interact with rigid bodies.
- You can control the interaction between different characters.

You can easily define a character controller by following these steps:

1 First, identify the model for the character controller.

2 Determine the axis to be used for the movement of the controller (upVector). The axis can be:

   - X-axis, which is vector(1,0,0).
   - Y-axis, which is vector(0,1,0).
   - Z-axis, which is vector(0,0,1).

   A character controller can be created using the `createController` method:

   ```
   controllerRef = createController(controllerName,modelName,controllerType,
   upVector,slopelimit)
   ```

   The controller type can be either `#box` or `#capsule`. If `#capsule` is specified, the bounding sphere of the model is used as the radius for the controller and the bounding box is used for the capsule height. If `#box` is specified, the model's bounding box is used as the controller's extents.

   The `slopelimit`, signifies the maximum slope that the character can walk. This is expressed as the cosine of the desired limit angle. A value of 0 disables this function.

   Note: *The character controller will automatically be treated as a rigid body by the physics engine.*

3 Move the character controller using the `move` method. When the controller is moved using this method, it also interacts with the rigid bodies in the world. If a callback has been registered by the controller, then it gets called during the interaction.

   When you need to move the character, specify the displacement vector and the distance:

   ```
   move(displacement, activegroupsList, minDist, sharpness)
   ```

   The displacement vector is primarily used to move the controller from the current location. The `activegroupList` denotes the list of active collision groups that the controller will interact with while moving. minDist, signifies the minimum distance the controller must travel. If the distance is lesser than this value, the character doesn't move. This is used to stop the recursive motion algorithm when the remaining distance to travel is small.

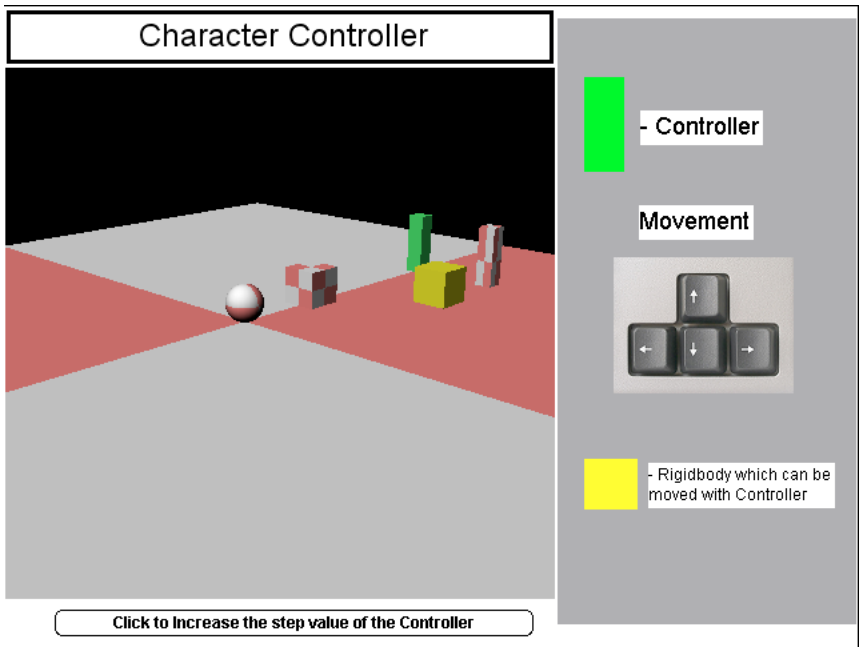
   When the controller is moving on an uneven terrain, to prevent the sudden height changes, the motion can be smoothened using a feedback filter. The `sharpness` coefficient, defines the amount of smoothing. A smaller value promotes better smoothing. (1.0 means no smoothing). Here's an example:

   ```
   rigidbody1.collisionsgroup = 1
   rigidbody2.collisiongroup = 2
   myct.move(vector(1,0,0),[1,2], 0.001,1)
   ```

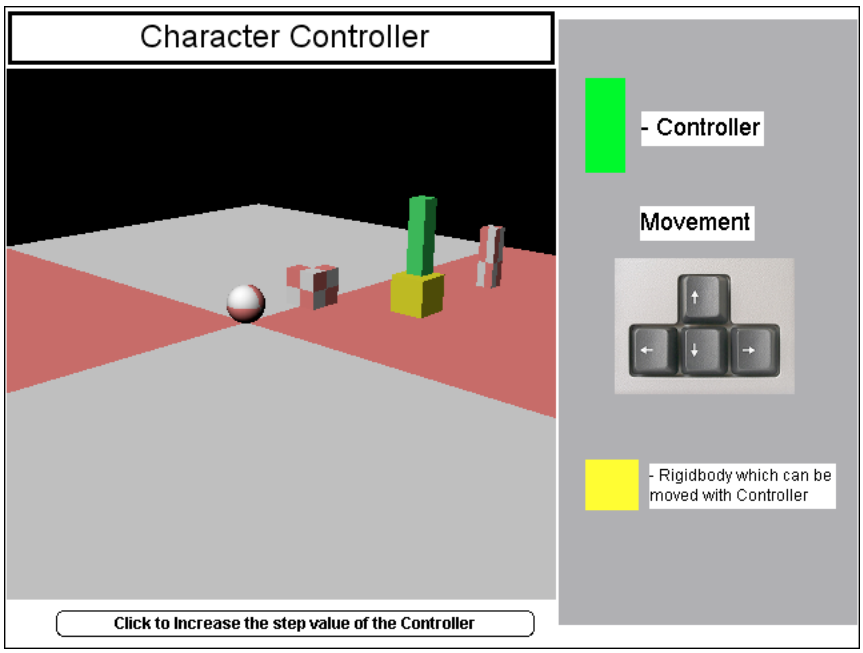
   The controller interacts with `rigidbody1` and `rigidbody2` while moving as they are in the list of active collision groups. If a callback has been registered for a rigid body-controller interaction, it will be invoked.

Download the movie CharacterController.dir to understand the character controller feature in Director.

You can use the arrow keys to move the controller (the green model), Try moving the controller towards the rigid body (the yellow model). The rigid body will get displaced.



Controller and rigid body interaction



You can make the controller 'step' on the rigid body

Also, increase the step value to make the controller climb on top of the rigid body. For more information on the supported character controller methods, see the API documentation.

Events

The collision modifier and the `bonesPlayer` and keyFrame player modifiers generate events automatically in certain situations. Some events may be generated only by particular models, others may be generated by any model when the given situation occurs. You can also use a 3D cast member to generate custom events at specific time intervals for a specific number of times.

To handler these events, you can register specific scripts or script instance. You can send custom events manually to registered scripts and instances.

2D events and 3D events

The way events are treated in Director's 2D environment is different from the way they are treated in a 3D world.

First, consider an example of a **2D event**. When the user clicks on a sprite, Director generates a `#mouseDown` event. The event seeks out an `on mouseDown()` handler in the following places.

- In one of the behaviors attached to the sprite
- In the Member Script of the cast member in the sprite
- In the Frame Script in the current frame
- In the first Movie Script that contains the appropriate handler

If there is a handler in any of these levels, and if the handler does not explicitly use the `pass` command to pass the event on to the next level, the event is not sent any further.

Events in a 3D world work in a different manner.

- **3D events** are never generated by the user. They are generated by changes that occur internally in the world, such as a collision between two models, or a bones animation reaching the end.
- Instead of looking for a pre-defined handler in a pre-defined sequence of locations, a 3D event will send a programmer-defined message to a programmer-defined script or script or instance.

In other words, in order to react to a 3D event, you need to prepare your scripts in a specific way. If you inherit a project from another team of developers, you may find that their method for organizing their scripts is very different from yours.

Registering a script or instance for a 3D event

Imagine that your 3D world contains a character model named “Estragon”, and that your 3D designer has created a number of different motions to play while this character is standing idle. Imagine that these motions are called “TapFoot”, “ScratchHead”, “LookRound”, “CheckPhone”, and “StandIdle”. Each of these motions must start and end with the character in exactly the same position, so that you play the motions seamlessly in any order. To make Estragon's behavior seem natural, you want to play the “StandIdle” motion most of the time, and occasionally play one of the other motions, at random intervals.

To set this up, you can create a handler similar to the following one:

```
on WhatDoIDoNow(aScriptReference, aEvent, aMotionName, aTime)
  vMotions = ["TapFoot","ScratchHead","LookRound","CheckPhone"]
  vRandom  = random(20)
  if vRandom > vMotions.count then
    vMotion = "Stand Idle"
  else
    vMotion = vMotions[vRandom]
  end if

  member("3D").model("Estragon").bonesPlayer.play(vMotion)
end WhatDoIDoNow
```

The idea is that this handler be called automatically each time the current motion comes to an end.

To do this, you need to register the handler as a callback for the Estragon model. If this handler is an a Movie Script, you can use the following lines of code:

```
-- Lingo syntax
vModel = member("3D").model("Estragon")
vModel.registerScript(#animationEnded, #WhatDoIDoNow, 0)
// JavaScript syntax
vModel = member("3D").getPropRef("model", 2) // use model index
vModel.registerScript(symbol("animationEnded"), symbol("WhatDoIDoNow"), 0)
```

The `0` as the third parameter indicates that the `on WhatDoIDoNow()` handler is in a Movie Script. If the handler is in a behavior or the instance of a Parent Script, then you can use a pointer to the script object in the place of the `0`. Here is a Lingo example of a Parent Script:

```
on new(me, aModel)
  aModel.registerScript(#animationEnded, #WhatDoIDoNow, me)
  return me
end new
on WhatDoIDoNow(aScriptReference, aEvent, aMotionName, aTime)
-- See full handler above
end WhatDoIDoNow
```

Custom handler names, custom locations

You will notice that the handler `on WhatDoIDoNow()` does not use a standard handler name, and that it can be placed in any script. You can register many different handlers for the same event if you wish. Each handler will be called in the order in which it was registered.

Events to register for

There are five 3D events that you can register for.

- `#collideAny`: Register a handler for the `#collideAny` event to trigger that handler whenever any two models to which the collision modifier is attached collide.
- `#collideWith`: If a model, to which the collision modifier is attached, registers a handler for the `#collideWith` event, then the handler will be called only when that model collides with another model to which the collision modifier is attached.
- `#animationStarted`: Register a handler for the `#animationStarted` event to trigger that handler whenever a bones or keyFrame motion starts playing on a model.
- `#animationEnded`: Register a handler for the `#animationStarted` event to trigger that handler whenever a bones or keyFrame motion stops playing on a model.

- `#timeMS`: Register a handler for the `#timeMS`event to trigger that handler at a given interval a given number of times.

Ways to register

Use the following methods to register a callback handler:

- Use member3D.registerForEvent() to register a handler to be called whenever the given event is triggered by any model.
- Use node.registerScript() to register a handler to be called only when the given event is triggered by a particular node
- Use model.collision.setCollisionCallback(...) as a shortcut for `model.registerScript(#collideWith, ...)`

For example, if you want information about all collisions in the member "3D" to be channeled to the `on CollisionDetected()` handler in a Movie Script, use the following commands:

```
-- Lingo syntax
member("3D").registerForEvent(#CollideAny, #CollisionDetected, 0)
// JavaScript syntax
member("3D").registerForEvent(symbol("CollideAny"), symbol("CollisionDetected"), 0);
```

If you want information about collisions with model 1 (named "Dark") in the member "3D" to be channeled to the `on DarkImpact()` handler of an instance of the script "Dark Events", use the following commands:

```
-- Lingo syntax
vInstance = new script("Dark Events")
member("3D").model("Dark").collision.setCollisionCallback(#DarkImpact, vInstance)
// JavaScript syntax
vInstance = new script("Dark Events");
member("3D").getPropRef("model", 1).getPropRef("collision",
1).setCollisionCallback(symbol("DarkImpact"), vInstance);
```

To send information on all animations that start playing in member "3D" to an `on ShowMotionDetails()` handler in a behavior on sprite 2, use the following code:

```
-- Lingo syntax
member("3D").RegisterForEvent(#animationStarted, #ShowMotionDetails, sprite 2)
// JavaScript syntax
member("3D").RegisterForEvent(symbol("animationStarted"), symbol("ShowMotionDetails"),
sprite(2));
```

To call the `on SecondThoughts()` handler in a behavior in the script "MetroGnome" every 1000 milliseconds exactly 60 times, starting 42 milliseconds from now, use the following code:

```
-- Lingo syntax
member("3D").RegisterForEvent(#timeMS, #SecondThoughts, script "MetroGnome", 42, 1000, 60)
// JavaScript syntax
member("3D").RegisterForEvent(symbol("timeMS"), symbol("SecondThoughts"), script
"MetroGnome", 42, 1000, 60);
```

Stopping callbacks

You can use member3D.unregisterAllEvents() to stop all callbacks to all handlers in all scripts for all events.

Note: *In Director 11.5, there is no way to unregister just one handler for one event. Design your scripts so that this functionality is not required.*

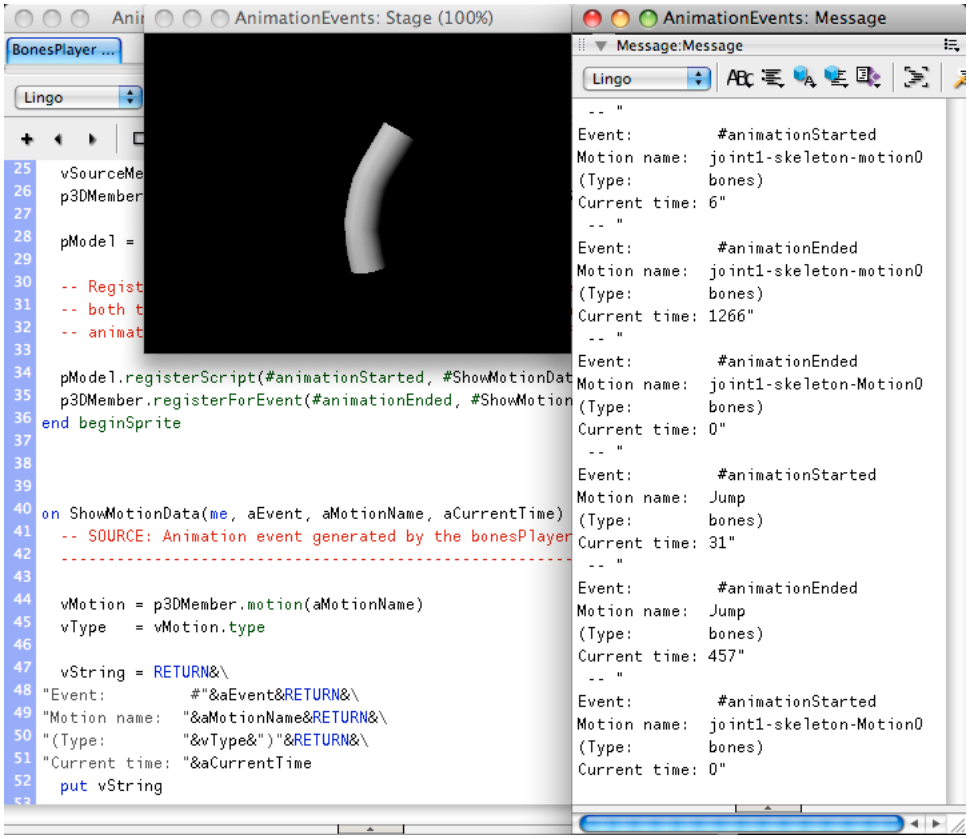
Alternatively, you can add a level of indirection and create a Registered Script Manager object to deal with registering and unregistering scripts. This object can store a look-up table of which scripts and handlers are registered for which events for which models. If you need to unregister just one handler for one event, your Registered Script Manager object can call `unregisterAllEvents()`*, and then restore all the other registrations by referring to its look-up table.*

Callback parameters

For details of the callback events created by the collision modifier, see "Collision modifier" on page 280. For details of the event created by the bonesPlayer and keyframePlayer modifiers see, "Animation event callback" on page 358. For details of the event created by the bonesPlayer and keyframePlayer modifiers, see "timeMS event callback" on page 359.

Animation event callback

To test the `#animationStarted` and `#animationEnded` 3D events, download and launch the movie AnimationEvents.dir. Check the output in the Message window.



Checking the parameters for registered animation events

The AnimationEvents.dir movie contains two 3D members. Both contain a single bones motion. The behavior on the 3D sprite uses `cloneMotionFromCastMember()` to copy the motion from the second 3D member into the member used by the 3D sprite.

It then registers itself for two callbacks, one for `#animationStarted` and one for `#animationEnded` events. Regardless of which technique is used to register the script for the events, the same parameters are sent with the callback.

The callback handler prints information about the event in the Message window. If the original motion has just finished playing for the first time, it queues the motion again to loop forever, and then makes the "Jump" motion jump the queue so that it actually plays first. This allows you to see what events are sent by looping and non-looping motions, and what happens when `play()` is used when the model.motionPlayer.playList already contains one or more motions.

Callback parameters

- `aEvent` will be either `#animationStarted` or `#animationEnded`
- `aMotionName` will be the string name of the motion that generated the event
- `aCurrentTime` will be the current time of the motion of milliseconds. If the motion is playing at the default rate of `1.0`, then this will also be the number of milliseconds since it started playing.

For looping animations, the `#animationStarted` event is issued only for the first loop, not for subsequent loops. During a blend of two animations, this event will be sent when the blending begins. No `#animationEnded` event is ever sent for a looping animation.

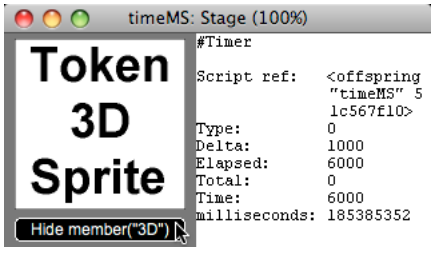
Pausing and stopping the animation callback

You can stop the `#animationStarted` and `#animationEnded` events from calling the registered handler by executing either of the following commands in the Message window. Both have far-reaching effects on other functionalities of the 3D member.

```
member("3D").unregisterAllEvents()
member("3D").resetWorld()
```

timeMS event callback

To test the `#timeMS` 3D event, download and launch the movie timeMS.dir.



#timeMS events are only generated when the 3D member is in a sprite on the Stage

The timerMS.dir movie contains a 3D sprite and a text sprite with the following behavior attached to it:

```
on beginSprite(me)
  member("3D").RegisterForEvent(#timeMS, #Timer, me, 0, 1000, 0)
end beginSprite
on Timer(me, aType, aDelta, aElapsed, aTotal, aTime)
  vString = \
"#Timer"&RETURN&RETURN& \
"Script ref:   "&me&RETURN& \
"Type:         "&aType&RETURN& \
"Delta:        "&aDelta&RETURN& \
"Elapsed:      "&aElapsed&RETURN& \
"Total:        "&aTotal&RETURN& \
"Time:         "&aTime&RETURN& \
"milliseconds: "&the milliseconds

  sprite(me.spriteNum).member.text = vString
end Timer
```

Callback parameters

- `aType` is always `0`.
- `aDelta` is the elapsed time in milliseconds since the last `#timeMS` event.
- `aElapsed` is the number of milliseconds since the first `#timeMS` event occurred. For example, if there are three iterations with a period of `500` ms, the first iteration's time will be `0`, the second iteration will be `500`, and the third will be `1000`.
- `aTotal` is the total number of milliseconds that will elapse between the `registerForEvent()` call and the last `#timeMS` event. For example, if there are five iterations with a period of `500` milliseconds, the duration is `2500` milliseconds. For tasks with unlimited iterations, the duration is `0`.
- `aTime` is the absolute time in milliseconds since the Director movie started.

These values are idealized. They do not take into account any interruptions made to the Director thread. The display in the text sprite will change roughly once per second. If you watch the value of the milliseconds, you will see that the `#timeMS` event is not called exactly every 1000 milliseconds. In fact, it is called soon after 1000 milliseconds have elapsed.

Pausing and stopping the #timeMS event

If you click the Hide Member("3D") button, the movie will jump to a frame where the 3D member is not displayed in a sprite. The updates to the text display will now stop. If you now click the Show Member("3D") button, the updates will continue from the point where they left off.

You can stop the #timeMS event from the Message window by executing either of the following commands:

```
member("3D").unregisterAllEvents()
member("3D").resetWorld()
```

Both have far-reaching effects on other functional it i es of the 3D member.

3D mathematics

Manipulating objects in 3D space requires the use of 3D mathematical concepts. Often, Director can take care of the mathematics for you, without you being aware of it. Many developers will use strategically-placed group nodes as parents for visible models, in order to avoid explicit use of mathematics.

This section provides recipes for certain operations that require mathematical knowledge. So long as you understand the purpose of each recipe, you do not need to understand the underlying mathematics.

This section also provides some insight into the mathematical operations, if you want to explore those further.

In Director, 3D mathematics is based on two objects:

- `vector()`: a three-dimensional representation of a point, a direction, a rotation or a scale. For more information, see "Vectors" on page 361 and "Vector methods and operations" on page 364.
- `transform()`: a three-dimensional representation of a frame of reference, including information on a position in space and the orientation and scale of 3 axes: x, y and z. For more information, see "Transforms" on page 370, "Transform properties" on page 372, and "Transforms methods" on page 372.

Vectors

A 3D vector describes both direction and location in 3D space. Vector objects include floating-point values for position along each of the *x*-, *y*-, and *z*-axes. Vectors can be node- or world-relative. If they are node-relative, their *x*, *y*, and *z* values are relative to the position of the node. If they are world-relative, their *x*, *y*, and *z* directions are relative to the world.

Vector mathematics operations perform calculations using each of the *x*, *y*, and *z* values. These calculations are useful for performing intelligent movement and rotation of models. See "Vector methods and operations" on page 364 for more details.

Creating a vector

You can create a new vector in three different ways

- vector() creates a vector from the given `x`, `y` and `z` values
- randomVector() creates a vector with random `x`, `y` and `z` values, such that the `length` or `magnitude` of the vector is `1.0` world unit.
- vector.duplicate() creates a copy of a source vector at a different location in the computer's memory.

Pointers and duplicates

Vectors are objects. Like other objects, a vector object points to an address in the computer's RAM space where the properties of the object are stored.

Imagine two variables that refer to the same vector object. If you changing the properties of the vector using one variable, the other variable will report exactly the same changes, because it is reading the values from the same address in RAM space.

To test this, try this experiment in the Message window. (For clarity, the output is shown for a Message window set for JavaScript. Identical commands will work in Lingo.)

```
v = randomVector()
<vector( -0.7667, -0.6387, 0.0654 )>
u = v
<vector( -0.7667, -0.6387, 0.0654 )>
u.x = 10
10
trace(u)
// <vector( 10.0000, -0.6387, 0.0654)>
trace(v)
// <vector( 10.0000, -0.6387, 0.0654 )>
```

Notice that the value of vector `v` has changed, even though you did not explicitly modify `v`. If you want to modify one copy of a vector without affecting the value of the original, use `duplicate()`.

```
v = u.duplicate()
v.y = 0
trace(v)
// <vector( 10.0000, 0.0000, 0.0654 )>
trace(u)
// <vector( 10.0000, -0.6387, 0.0654 )>
```

When you retrieve a vector from a transform, you receive a duplicate of the vector. Changing the duplicate will leave the original unchanged.

```
t = transform()
v = t.position
v.x = 100
trace(v)
// <vector( 100.0000, 0.0000, 0.0000 )>
trace(t.position)
// <vector( 0.0000, 0.0000, 0.0000 )>
```

If you want to modify the values in the transform, you need to explicitly set the transform's property.

```
t.position = v
trace(t.position)
// <vector( 100.0000, 0.0000, 0.0000 )>
```

Vector properties

Use the following properties to work with vectors:

Property	Access	Description
`vector.magnitude`	get	The magnitude of the vector. Equivalent to the length of the vector.
`vector.length`	get	The length of the vector. Equivalent to the magnitude of the vector.
`vector[`***`index`***`]`	get and set	Returns the value of a vector at a specified point in an index between 1 and 3.
`vector.x`	get and set	The *x* component of a vector.
`vector.y`	get and set	The *y* component of a vector.
`vector.z`	get and set	The *z* component of a vector.

Here are some examples:

```
v = randomVector()
trace(v.length)
-- 1.0000
trace(v.magnitude)
 -- 1.0000
trace(v.x * v.x + v.y * v.y + v.z * v.z)
 -- 1.0000
v[1] = 3
v[2] = 4
v[3] = 12
trace(v)
 -- vector( 3.0000, 4.0000, 12.0000 )
trace(v.mag
nitude)
 -- 13.0000
```

Position and translation vectors

A vector is essentially used to express a **position** in space or a **translation** through a given distance in a given direction. A position or translation vector is only meaningful within a parent frame of reference (see "Frames of reference" on page 202).

A translation expressed as `vector(x, y, z)` can be thought of as the following movement:

1. A translation of *x* units parallel with the xAxis of the parent frame of reference
2. A translation of *y* units parallel to the yAxis of the parent frame of reference, starting from the point arrived at in step 1
3. A translation of *z* units parallel to the zAxis of the parent frame of reference, starting from the point arrived at in step 2

The actual order of the steps is not in fact important.

A position vector can be thought of as the result of a translation that starts from the origin of the parent frame of reference.

The vector methods and operations described at "Vector methods and operations" on page 364 are only meaningful when vectors are used to express positions or displacement.

Other uses of vectors

The vector format of `(float, float, float)` is also useful for expressing quantities other than positions or translation. For example:

- `transform.`**`rotation`**: the `x`, `y` and `z` properties indicate a sequential rotation around each axis. In other words, a rotation vector says: "Rotate `x` degrees around the `xAxis`, then `y` degrees around the `yAxis`, and then `z` degrees around the `zAxis`, in that order".
- `transform.`**`scale`**: the `x`, `y` and `z` properties provide a shorthand for `transform.xAxis.length`, `transform.yAxis.length` and `transform.zAxis.length`.
- `spotLight.`**`attenuation`**: the `x`, `y` and `z` properties provide a shorthand for the components of a quadratic equation used to calculate how the illumination for a spotlight decreases with distance. This can be expressed as follows:

  ```
  intensity = attenuation.z * attenuation.z * distance + attenuation.y * distance +
  attenuation.x
  ```

- `object3d.gravity`: the `x`, `y` and `z` properties represent the acceleration in world units per second-squared. This is used by Physics members and particle resources.

Vectors are also used in association with a scalar, as part of a list, to define other

- `node.boundingSphere`: expressed as `[<vector position>, <scalar radius>]`
- `object3D.axisAngle`: expressed as `[<rotation vector>, <scalar angle of rotation>]`. This is used for `transform.axisAngle`, for `linearJoint.properties` and for `D6Joint.driveOrientation`.

Note: *Performing vector mathematics on vectors that are not used to express a position or a displacement requires a good understanding of the concepts that these vectors represent. All examples given in this documentation are for operations performed on positions or displacements.*

Vector methods and operations

You can find an alphabetical list of scripting terms to use with vectors at Vector mathematics. Here, the terms are grouped together by usage.

- Unit vectors
- Comparing vectors
- Distance and angle
- Binary operations
- Uses of binary operations
- Vector products
- Dot product
- Cross product

Note: *The vector methods and operations described here are only meaningful when the vectors in question are used to express positions or translations.*

Unit vectors

A unit vector is a vector with a `length` or `magnitude` of `1.0` world unit. Unit vectors are useful for indicating a direction. You can multiply a unit vector by a scalar speed to create a velocity (a speed in a given direction). There are two methods for converting a vector to a unit vector.

- vector.getNormalized() returns a unit vector with the same direction as the original vector. The original vector is not changed.
- vector.normalize() modifies the original vector so that its `length` or `magnitude` now has a value of `1.0`.

Note: *Do not confuse a normalized vector and a normal vector. A normal vector is a direction vector that is at right angles to the surface of a plane or a face. A normal vector need not be a unit vector (but it usually is). A normalized vector is any vector that has been scaled to be one unit in length.*

Before using `normalize()`, you may want to save the `length` or `magnitude` property of the vector in a variable, so that you can restore the original vector later.

```
-- Lingo syntax
vVector = randomVector()
put vVector
 -- vector( -0.7949, 0.2442, 0.5555 )
vLength = random(100)
put vLength
 -- 9
vVector = vVector * vLength
put vVector
 -- vector( -7.1538, 2.1977, 4.9993 )
vMagnitude = vVector.magnitude
put vMagnitude
 -- 9.0000
vDirection = vVector.getNormalized()put vDirection
 -- vector( -0.7949, 0.2442, 0.5555 )
vVector.normalize()put vVector
 -- vector( -0.7949, 0.2442, 0.5555 )
```

Note: *JavaScript does not support mathematical operations for vector objects. In JavaScript, you must write the code to perform the vector math calculations using the vector's* `x`, `y`, *and* `z` *coordinates.*

```
// JavaScript syntax
vVector = randomVector()
<vector( -0.7949, 0.2442, 0.5555 )>
vLength = random(100)
9
vVector.x = vVector.x * vLength
-7.153941750526428
vVector.y = vVector.y * vLength
2.197822719812393
vVector.z = vVector.z * vLength
4.999067902565002
trace(vVector);
// <vector( -7.1539, 2.1978, 4.9991 )>
vMagnitude = vVector.magnitude;
8.999999148200093
vDirection = vVector.getNormalized()<vector( -0.7949, 0.2442, 0.5555 )> )>
vVector.normalize()0
trace(vVector)
// <vector( -0.7949, 0.2442, 0.5555 )>
```

Note: *JavaScript uses a different Math library from Lingo, using 32-bit floating point numbers, where Lingo uses 16-bit floating point numbers for 3D operations. This can lead to slight differences in the least-significant figures.*

Comparing vectors

Vectors use floating point numbers. Comparing the equality of two floating point numbers can lead to unexpected results, if there are small differences between them. Note that JavaScript may be more precise in its evaluations of equality than Lingo is.

```
-- Lingo syntax
put 1.000000000001 = 1.0
 -- 0
put 0.99999999999999 = 1.0
 -- 1
// JavaScript syntax
trace(1.000000000000001 == 1.0);
// false
trace(0.99999999999999999 == 1.0);
// true
```

To compare vectors it is safer to check if the individual `x`, `y` and `z` values fall within a given range.

```
-- Lingo syntax
v = vector( -0.01, -64.0, 0.01 )
v.normalize()
put v.y
 -- -1.0000
put v
 -- vector( -0.0002, -1.0000, 0.0002 )
put v = vector(0, -1, 0)
 -- 0
trace(v.y < -0.9999)
 -- 1
// JavaScript syntax
v = vector( -0.01, -256.0, 0.01 )
<vector( -0.0100, -256.0000, 0.0100 )>
v.normalize()
0
trace(v.y)
// -1
trace(v)
// <vector( 0.0000, -1.0000, 0.0000 )>
trace(v == vector(0, -1, 0))
// false
trace(v.y < -0.9999)
// true
```

Distance and angle

For two vectors that both represent a position in space, it is meaningful to calculate the distance between them. For two vectors which both represent a translation, it is meaningful to compare their lengths. There are three techniques that you can use to do this:

- `vDistance = vector1.distanceTo(vector2)` See `vector.distanceTo()` for details
- `vDistance = (vector1 - vector2).magnitude`
- `vDistance = (vector1 - vector2).length`

Note that it does not matter which order the vectors appear in. The result will be the same.

These operations are not meaningful if both vectors are not position vectors, or if both vectors are not translation vectors. It makes no sense to calculate the distance between a point and a translation.

```
vector1 = vector(3, 4, 5)
vector2 = vector(5, 12, 13)
trace(vector1.distanceTo(vector2))
 -- 11.4891
trace(vector2.distanceTo(vector1))
 -- 11.4891
put (vector1 - vector2).magnitude
 -- 11.4891
put (vector2 - vector1).length
 -- 11.4891
```

Note: *Subtracting one vector for another is not permitted in JavaScript.*

For two vectors, both of which represent a translation or a direction, it is meaningful to calculate the angle between them. See vector.angleBetween() for more information.

The angle between two direction vectors will be in the range `0.0` (if the vectors are parallel) to `180.0` degrees (if they are pointing in opposite directions).

If one vector is pointing North and the second vector is at 135° to it, you cannot tell whether the second vector is pointing South-East or South-West. To distinguish between the two cases, you will also need to know which direction is Up. The vector.crossProduct() (see below) of the two vectors will point either up or down, depending on whether the second component points more or less East, or not.

Binary operations

A *scalar* is a one-dimensional number, such as `42`, `-0.7071`, `pi` or `_movie.frameTempo`. The table below shows you what addition, substraction, multiplication, and division operations are permitted between scalars and vectors, vectors and vectors, and transforms and vectors. Note that some operations are meaningless, and fail silently. Other meaningless operations lead to a script error.

Operation	Description	Output
`scalar + vector1` *or* `vector1 + scalar`	Returns a new vector equaling `vector(vector1.x+scalar, vector1.y+scalar, vector1.z+scalar)`	A new vector object
`scalar - vector1`	Returns a new vector equaling `vector(scalar-vector1.x, scalar-vector1.y, scalar-vector1.z)`	A new vector object
`vector1 - scalar`	Returns a new vector equaling `vector(vector1.x-scalar, vector1.y-scalar, vector1.z-scalar)`	A new vector object
`vector * scalar` *or* `scalar * vector`	Returns a new vector with the same direction as `vector` and a `length` equal to `vector.length * scalar`	A new vector object
`vector / scalar`	Returns a new vector with the same direction as `vector` and a `length` equal to `vector.length / scalar`	A new vector object
`scalar / vector`	**Operation not supported**	***** Script error *****
`vector1 + vector2`	Returns a new vector equal to `vector(vector1.x+vector2.x, vector1.y+vector2.y, vector1.z+vector2.z)`	A new vector object

Operation	Description	Output
`vector1 - vector2`	Returns a new vector representing the displacement from the position of vector 2 to the position of vector1.	A new vector object
`vector1 * vector2`	Returns the `dotProduct()` of the two vectors. See below.	A floating-point value
`vector1 / vector2`	Meaningless operation.	0
`transform * vector`	Returns a new vector that translates, rotates and scales the `vector` by the amounts defined in the `transform`.	A new vector object

`transform + vector`
`transform - vector`
`transform / vector`
	Meaningless operations.	0

`vector + transform`
`vector - transform`
`vector / transform`
	Meaningless operations.	0

`vector * transform`	**Operation not supported.**	***** Script error *****

Uses of binary operations

Operation	Uses

`-vector`	If the vector is a translation, reverses the direction of the translation.
	`vServerToReceiver = vReceiver.worldPosition - vServer.worldPosition`
	`-- (see vector1 - vector2 below)`
	`vReceiverToServer = -vServerToReceiver`

`vector * scalar`	Scales a vector If the vector is a unit direction vector, and the scalar is the distance to travel, the result is a translation vector.the distance to travel, the result is a translation vector. `vSpaceShip.translate(vDirectionVector * vDistance)`
`vector / scalar`	Divides a vector into smaller steps. If the vector is a translation vector and the scalar is a number of frames, the result will be the translation to move on each frame. `vMovementPerFrame = vTotalMovement / vNumberOfFrames`

`vector1 + vector2`	1) If `vector1` is a position and `vector2` is a translation, the result is a new position.
	`vNewPosition = vOldPosition + vTranslation`
	2) If `vector1` and `vector2` are both translations, then the result is a new translation that combines both movements
	`vDiagonalMotion = vForwardMotion + vSidewaysMotion`

Operation	Uses

`vector1 - vector2`	1) If `vector1` and `vector2` are both position vectors, then the result is a translation vector that travels from `vector2` to `vector1`.
	`vPathFromPoint2ToPoint1 = vPoint1 - vPoint2`
	2) If `vector1` is a position and `vector2` is a translation, the result is a new position, resulting from traveling backwards along the translation vector
	`vPreviousPosition = vCurrentPosition + vTranslationFromPreviousPosition`
	3) If `vector1` and `vector2` are both translations, then the result is a new translation equivalent to traveling forwards along `vector1` and then backwards along `vector2`.

`vector1 * vector2`	See "Dot product" on page 369.
`transform * vector`	Allows you to change the frame of reference of a vector. See "Transforms methods" on page 372.

Vector products

The table above shows that division by a vector is a mathematically meaningless operation. However, there are two different ways to multiply two vectors together: *dot product* and *cross product*. For both operations, Director provides three different approaches. You can choose one approach for each operation, and use that at all times. For clarity, all examples will use the most verbose syntax.

Dot product

In mathematics, the dot product of vector u by vector v is written: $u.v$

The dot product of two vectors creates a scalar that is equivalent to the cosine of the angle between the vectors, multiplied by the magnitude of each vector.

Note: *To visualize a cosine, imagine a ladder propped against a wall. If the ladder is 1 unit long, and the angle between the ladder and the ground is Angle, then the distance between the ladder and the wall will be* `cos(Angle)`*. Click here for more information.*

All three of the techniques below will give the same result.

- `vDotProduct = vector1 * vector2`
- `vDotProduct = vector1.dot(vector2)`
- `vDotProduct = vector1.dotProduct(vector2)`

Note that the order of the vectors is not important: `vector1.dotProduct(vector2)` gives the same result as `vector2.dotProduct(vector1)`.

The `dotProduct()` has a number of interesting characteristics:

- For two unit vectors, the `dotProduct()` is equivalent to the cosine of the angle between the vectors.
- For two vectors at exactly right angles to each other, the `dotProduct()` is zero.
- If you have a vector position on a plane P, a unit vector normal to the plane N, and a vector point in space Q, then the distance from Q to the nearest point on the plane is (Q - P).dotProduct(N).
- Imagine that you have an equation with vectors on both sides. You can simplify the equation by taking the `dotProduct()` of both sides by a vector that is perpendicular to one of the vectors on one side.

You find examples that use these characteristics at "3D mathematics recipes" on page 378.

Cross product

In mathematics, the cross product of vector `u` by vector `v` is written: `u x v`

The cross product of two vectors creates a new vector that is at right angles to both vectors. Its length will depend on both the angle between the vectors and their magnitude. In most cases, you will be interested in the direction of the crossProduct and not its length.

All three of the techniques below will give the same result.

- `vCrossProduct = vector1.cross(vector2)`
- `vCrossProduct = vector1.crossProduct(vector2)`
- `vCrossProduct = vector1.perpendicularTo(vector2)`

To visualize the direction of the output vector, use the thumb, index finger and middle finger of your right hand. (If you are left-handed, you must still use your right hand). Point your thumb in the direction of `vector1`, your index finger in the direction of the `vector2`, and turn your middle finger at right angles to both of the input vectors.

Note: *The order in which you use the vectors is important. Imagine that the input vectors point upwards and to the right. If you point your thumb upwards and your index finger to the right, your middle finger will point away from you. If you point your thumb to the right and your index finger upwards, your middle finger will point towards you.*

The crossProduct is useful for determining the axis of a rotation. Imagine that you have a spaceship pointing at planetA and you want to turn it to point at planetB, using the least amount of fuel. You can define two vectors:

```
vToPlanetA = vPlanetA.worldPosition - vSpaceShip.worldPosition
vToPlanetB = vPlanetM.worldPosition - vSpaceShip.worldPosition
```

You can now obtain the axis of rotation using `crossProduct()`:

```
vAxisOfRotation = vToPlanetA.crossProduct(vToPlanetB)
```

When you turn around an axis, you use a different right-hand rule. Point your thumb in the direction of the axis and curl your fingers: your fingers will point in the direction of positive rotation. Imagine that the spaceship is pointing at planetA straight ahead of you, and that planetB is off to your right and slightly downwards. With your right hand, point your thumb forwards and your index finger to the right and downwards. Your middle finger will probably now be pointing to the ground just beneath you. That is the direction of the axis of rotation.

Point your right thumb in the direction of the axis of rotation and point your fingers forward. curl your fingers round. Now curl your fingers round: they will turn in the direction your imaginary spaceship needs to turn in order to point at the imaginary planetB.

It takes just two more lines to put your spaceship on target:

```
vAngleOfRotation = vToPlanetA.angleBetween(vToPlanetB)
vSpaceShip.rotate(vSpaceShip.worldPosition, vAxisOfRotation, vAngleOfRotation)
```

Transforms

In this section and in the following sections, the terms are grouped together by usage and by type.

To experiment with transforms, download and open the movie 3DMathematics.dir. The movie contains a number of scripts that you may find useful. In particular, the OS 3DeBug script contains a `showTransform()` handler you can use to display transform data neatly in the Message window.

Creating a transform

There are three ways to create a new transform object.

- `vNewTransform = transform()`
- `vNewTransform = vExistingTransform.duplicate()`
- `vNewTransform = node.getWorldTransform()`

See transform(), transform.duplicate() and node.getWorldTransform() for details.

Like vectors, transforms are objects. See the "Pointers and duplicates" on page 361 for details of the implications of this.

Visualizing a transform

A transform represents the position, rotation and scale of an object in 3D space. With the 3DMathematics.dir movie open, try the following in the Message window:

```
t = transform()
t.position = vector(2, 3, 5)
t.rotation = vector(0, 60, 0)
t.scale = vector(0.1, 1, 10)
put t
 -- transform(0.05000,0.00000,-0.08660,0.00000, 0.00000,1.00000,0.00000,0.00000,
8.66025,0.00000,5.00000,0.00000, 2.00000,3.00000,5.00000,1.00000)
put showTransform(t)
 -- "
  0.05000  0.00000  -0.08660  0.00000
  0.00000  1.00000   0.00000  0.00000
  8.66025  0.00000   5.00000  0.00000
  2.00000  3.00000   5.00000  1.00000
"
```

The position and axes of a transform

Here is the same transform arranged in a table with explanatory headers:

	world x-axis	**world y-axis**	**world z-axis**	**direction (0) or position (1)?**
transform x-axis	0.05	0.0	-0.0866	0
transform y-axis	0.0	1.0	0.0	0
transform z-axis	8.66	0.0	0.5	0
position	2.0	3.0	5.0	1

You can recognize that the last row of numbers in the transform represents the `position` vector that you set. The other lines represent the direction and scale of the x, y and z axes of the transform in world coordinates.

The transform has been rotated around its yAxis, and scaled by 1.0 along its yAxis, so the transform's yAxis is pointing upwards 1.0 units on the world's yAxis. If you compare the figures in the first and third lines, you can see that the xAxis has been scaled by 0.1, and the zAxis has been scaled by 10, and that these axes are now at an angle to the world's axes.

Transform properties

Transforms have four properties that you can both get and set:

- transform.position: a vector that defines the position of the transform with respect to its frame of reference.
- transform.rotation: a vector that defines the rotation of the transform with respect to its frame of reference.
- transform.scale: a vector that defines the scale of the transform with respect to its frame of reference.
- transform.axisAngle: a list with the format `[<axis direction vector>, <float angle of rotation about axis>]`. This provides the same information as `transform.rotation`, but in a different format. Changing the `rotation` will update the `axisAngle` and vice versa.

Transforms also have three properties that you can get but not set:

- transform.xAxis
- transform.yAxis
- transform.zAxis

See the other articles in this section for a hands-on explanation of using these properties.

Note: *The camera looks back down zAxis of its world transform. In an activity with a first person camera, you can use* `sprite("3D").camera.getWorldTransform().zAxis` *to tell in which direction the main character is facing. (Note the negative sign).*

The view displayed in the sprite will be on a plane whose normal is identical to the zAxis of the camera.

Transforms methods

The scripting terms are grouped together by usage in this section.

- Positioning a transform
- Operations with transforms
- Using a transform to set a frame of reference
- Parent and child relationships with no nodes
- Applying one transform to another
- Interpolation

Positioning a transform

- transform.translate()
- transform.rotate()
- transform.scale()

All these methods accept two types of parameters:

- A vector
- Three separate x, y, and z numbers

The `rotate()` method also allows a third set of parameters:

```
vectorPosition, vectorAxis, scalarAngle
```

For example, the three `rotate()` commands below have an identical effect:

```
vTransform = member("3D").model(1).transform
vTransform.rotate(0, 45, 0)
vTransform.rotate(vector(0, 45, 0))
vTransform.rotate(vTransform.position, vTransform.yAxis, 45)
```

Nodes have similar methods. However, there are two main differences.

Nodes have a parent property. Transforms do not. When you apply the `translate()`, `rotate()`, or `scale()` methods to a transform, you cannot use a additional frame of reference parameter such as #self, #parent or #world. If you do so, a script error will occur.

The node.scale() method will also accept a single scalar value, to scale the node uniformly on all three axes. In Director 11.5, if you attempt to use a single scalar value as the parameter for the `transform.scale()` method, you will get a script error.

See "Translation" on page 206 and "Rotate()" on page 209 for examples of using the `translate()` and `rotate()` methods.

Operations with transforms

As the table below shows, there are many operations that will fail silently when used with transforms, and two that are useful.

`scalar + transform`
`scalar - transform`
`scalar / transform`
	Meaningless operations.	0

`scalar * transform`	**Operation not supported.**	***** Script error *****
`transform * vector`	Returns a new vector that is equivalent to the worldPosition of `vector` when `vector` is applied to the frame of reference defined by `transform` (see "Using a transform to set a frame of reference" on page 374)	A new vector object

`transform + vector`
`transform - vector`
`transform / vector`
	Meaningless operations.	0

`vector + transform`
`vector - transform`
`vector / transform`
	Meaningless operations.	0

`vector * transform`	**Operation not supported.**	***** Script error *****
`transform1 * transform2`	Returns a new transform equivalent to `transform2.multiply (transform1)` or `transform1.preMultiply (transform2)` (see "Parent and child relationships with no nodes" on page 374)	A new transform

Using a transform to set a frame of reference

Imagine that you have a bead, a small box, and a desk. You place the box at the corner of your desk, and align it with the edges of the desk. The box and the desk now have the same frame of reference. You glue the bead inside the box at a particular position relative to the desk. You can use a vector to represent the position of the bead. Now you pick up the box, twist it around, and place it on top of one of the other objects on your desk. The bead is still in the same position relative to the box, but the box is in a new position relative to the desk. You can represent the position of the box by a transform.

To determine the position of the bead relative to the desk, you can multiply the box's transform by the bead's vector.

```
vBeadPositionRelativeToDesk = vBoxTransform * vBeadVector
```

To test this in Lingo, download, and open the movie 3DMathematics.dir, then execute the following commands in the Message window. This simulates gluing a bead inside a box at the position `vector(13, 17, 19)`, and then turning the box through 90° and placing it in a new position and a new height, represented by `vector(5, 7, 11)`.

```
vBeadVector = vector(13, 17, 19)
vBoxTransform = transform()
vBoxTransform.position = vector(5, 7, 11)
vBoxTransform.rotate(0, 90, 0)
put ShowTransform(vTransform1)
 -- "
  -0.00000   0.00000  -1.00000   0.00000
   0.00000   1.00000   0.00000   0.00000
   1.00000   0.00000  -0.00000   0.00000
  11.00000   7.00000  -5.00000   1.00000
"
put vBoxTransform * vBeadVector
 -- vector( 30.0000, 24.0000, -18.0000 )
```

The box's transform has turned 90° around the world's yAxis, so the box's xAxis is now pointing down the desk's negative zAxis, and the box's zAxis is pointing along the desk's xAxis. The bead inside the box has turned in the same way. Its `x` value of `13` is subtracted from the box's z value of -5 to place the bead -18 units along the desk's zAxis. The bead's z value of 19 has been added to the box's x value of 11, to place the bead 30 units along the desk's xAxis. The box's yAxis is still facing upwards, so the bead's new vertical position has increased by the elevation of the box: 17 + 7 = 24.

Parent and child relationships with no nodes

Multiplying one transform by another has the same effect as setting one node to be the child of another. To put it another way: when one node is made the child of another, the transforms of the two nodes are multiplied together.

To test this, execute the following commands in the Message window. Your commands will create a second transform, and apply the two transforms to two groups. You will then make one group the parent of the other, and compare the value of `getWorldTransform()` for the child node with the result of multiplying the two transforms together.

```
vBeadTransform = transform()
vBeadTransform.position = vBeadVector
put ShowTransform(vBeadTransform)
 -- "
   1.00000   0.00000   0.00000  0.00000
   0.00000   1.00000   0.00000  0.00000
   0.00000   0.00000   1.00000  0.00000
  13.00000  17.00000  19.00000  1.00000
"
member("3D").resetWorld()
vBoxGroup  = member("3D").newGroup("Box")
vBeadGroup = member("3D").newGroup("Bead")
vBoxGroup.transform  = vBoxTransform
vBeadGroup.transform = vBeadTransform
vBoxGroup.addChild(vBeadGroup, #preserveParent)
put ShowTransform(vBeadGroup.getWorldTransform())
 -- "
  -0.00000   0.00000  -1.00000  0.00000
   0.00000   1.00000   0.00000  0.00000
   1.00000   0.00000  -0.00000  0.00000
  30.00000  24.00000 -18.00000  1.00000
"
put ShowTransform(vBoxTransform * vBeadTransform)
 -- "
  -0.00000   0.00000  -1.00000  0.00000
   0.00000   1.00000   0.00000  0.00000
   1.00000   0.00000  -0.00000  0.00000
  30.00000  24.00000 -18.00000  1.00000
"
```

Identity and inverse

When you first create a transform, it is placed at the center of its frame of reference, with its axes aligned to the frame of reference, and with a scale of 1.0. This is the "identity" position for a transform. When you first create a model, it will be given an identity transform.

To return a transform to its original state, you can use this command: transform.identity()

You can use `transform.inverse()` to generate a new transform that will undo the effect of a given transform. If a transform and its inverse are multiplied together, the result is the identity transform.

To convert a transform into its inverse, you can use `transform.invert()`.

Here are some experiments that you can try in the Message window:

```
t = transform()
t.position = vector(2,3,5)
t.rotation = vector(30, 45, 60)
t.scale = vector(0.5, 1.0, 2.0)
trace(showTransform(t))
 -- "
   0.17678  0.30619  -0.35355  0.00000
  -0.57322  0.73920   0.35355  0.00000
   1.47840  0.56066   1.22474  0.00000
   2.00000  3.00000   5.00000  1.00000
"
i = t.inverse()
trace(showTransform(i))
 -- "
   0.70711  -0.57322   0.36960  0.00000
   1.22474   0.73920   0.14017  0.00000
  -1.41421   0.35355   0.30619  0.00000
   1.98262  -2.83892  -2.69063  1.00000
"
put showTransform(t*i) -- this creates an identity transform
 -- "
  1.00000  -0.00000  -0.00000  0.00000
  0.00000   1.00000  -0.00000  0.00000
  0.00000   0.00000   1.00000  0.00000
  0.00000   0.00000   0.00000  1.00000
"
```

Note: *JavaScript does not support the multiplication of transforms.*

```
i.invert() -- i will now be identical to t
trace(showTransform(i))
 -- "
   0.17678  0.30619  -0.35355  0.00000
  -0.57322  0.73920   0.35355  0.00000
   1.47840  0.56066   1.22474  0.00000
   2.00000  3.00000   5.00000  1.00000
"
t.identity()
trace(showTransform(t))
 -- "
  1.00000  0.00000  0.00000  0.00000
  0.00000  1.00000  0.00000  0.00000
  0.00000  0.00000  1.00000  0.00000
  0.00000  0.00000  0.00000  1.00000
"
```

Applying a preliminary manipulation to a transform

The following commands reset the transform to its identity state, apply a modification to the identity transform, and then re-apply the original transform to the new state.

- transform.preTranslate()
- transform.preRotate()
- transform.preScale()

Applying one transform to another

As illustrated in the "Parent and child relationships with no nodes" on page 374 section above, the * (multiply) operator can generate a new transform by applying one transform to another. If you prefer to modify one of the original transforms you can use one of the following methods:

- transform.multiply()
- transform.preMultiply()

To test this in the Message window, create a transform `t` that represents a translation and a transform `r` that represents a rotation:

```
t = transform()
t.translate(13, 17, 19)
r = transform()
r.rotate(90, 0, 0)
```

Generate a transform that represents a rotation followed by a translation, without changing the original transforms. Note that the y and the z axes change as the initial rotation occurs around the `vector(0, 0, 0)` and then the translation `t` is applied.

```
put showTransform(t*r)
 -- "
   1.00000   0.00000   0.00000  0.00000
   0.00000  -0.00000   1.00000  0.00000
   0.00000  -1.00000  -0.00000  0.00000
  13.00000  17.00000  19.00000  1.00000
"
```

Create a duplicate of the rotation transform, and apply the translation to it using `multiply()`. This will modify the duplicate rotation transform.

```
r2 = r.duplicate()
r2.multiply(t)
put showTransform(r2)
 -- "
   1.00000   0.00000   0.00000  0.00000
   0.00000  -0.00000   1.00000  0.00000
   0.00000  -1.00000  -0.00000  0.00000
  13.00000  17.00000  19.00000  1.00000
"
```

Generate a transform that represents a translation followed by a rotation, without modifying the original transforms.

```
put showTransform(r*t)
 -- "
   1.00000    0.00000   0.00000  0.00000
   0.00000   -0.00000   1.00000  0.00000
   0.00000   -1.00000  -0.00000  0.00000
  13.00000  -19.0000017.00000  1.00000
"
```

Use `r.preMultiply()` to apply the translation first, followed by the rotation, modifying the `r` transform in the process.

```
r.preMultiply(t)
put showTransform(r)
 -- "
   1.00000    0.00000   0.00000  0.00000
   0.00000   -0.00000   1.00000  0.00000
   0.00000   -1.00000  -0.00000  0.00000
  13.00000  -19.00000  17.00000  1.00000
"
```

Interpolation

The following methods allow you to calculate a transform that is partway between one transform and another.

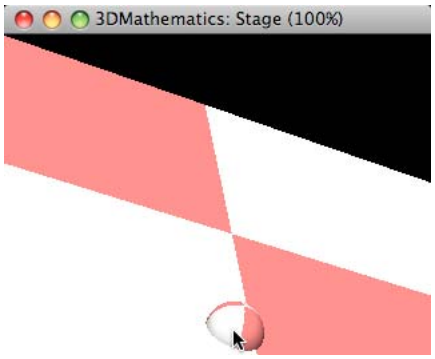
- transform.interpolate()
- transform.interpolateTo()

These methods are illustrated with example movies at "Interpolation" on page 264.

3D mathematics recipes

This section provides handlers for some common 3D manipulations. You can find all these handlers and more in the scripts in the movie 3DMathematics.dir.

- Calculating the normal to a plane
- Getting the shortest distance to plane
- Getting the closest point in a plane
- Mapping a vector to a plane
- Finding where a ray intersects a plane
- Getting a bearing
- Reflecting a vector in a plane
- Rotating a vector position around an axis
- Converting a world transform to a local transform
- Get bounding box



Using RayCutsPlane() to drag a model across an arbitrary plane

Defining a plane

To define a plane, you need two vectors:

- A position on the surface of the plane
- A direction vector normal to the surface of the plane.

A normal is a vector that is at right angles to the surface of a plane or a face in all directions. If you know three points on the surface of the plane that form a triangle, you can calculate the normal. Each face of the mesh of a model has three vertex points.

Calculating the normal to a plane

Here is a handler that calculates the normal to a plane, given three points on the plane. This technique is used in the movie MeshDeform.dir.

```
on GetNormal(aVertex1, aVertex2, aVertex3) ----------------------
  -- INPUT: <aVertex1>, <aVertex2>, <aVertex3> must all be vector
  --         positions that together define a triangle. (They
  --         must not all lie on the same straight line). They
  --         are considered to be listed in a counter-clockwise
  --         order, when looked at from the front.
  -- OUTPUT: Returns a unit vector that is at right angles to all
  --         vectors between the three vertex points. Together
  --         with any one of these vertex points, the normal
  --         defines a plane.
  ---------------------------------------------------------------

  vVectorA = aVertex2 - aVertex1
  vVectorB = aVertex3 - aVertex1
  vNormal  = vVectorA.crossProduct(vVectorB)
  vNormal.normalize()

  return vNormal
end GetNormal
```

Getting the shortest distance to a plane

Suppose you have a Fridge Magnet activity. You know that the magnet is strong enough to jump to the fridge if it is less than a given distance away. How do you know when the magnet is near enough to the surface of the fridge to jump?

Suppose you want to set off an alarm when the user's avatar gets within a certain distance of a wall. How do you know when to set off the alarm?

The following handler can help you solve these questions.

```
on GetDistanceToPlane(aPointInSpace,aPointOnPlane,aNormalToPlane)
  -- INPUT:  All parameters must be vectors
  --        <aPointInSpace> and <aPointOnPlane> must be position
  --         vectors
  --        <aNormalToPlane> must not have a length of zero
  -- OUTPUT: Returns the distance from aPointInSpace to the
  --         closest point in the plane
  ------------------------------------------------------------------

  vNormal       = aNormalToPlane.getNormalized()
  vPlaneToPoint = aPointInSpace - aPointOnPlane
  vDistance     = vPlaneToPoint.dotProduct(vNormal)

  return vDistance
end GetDistanceToPlane
```

Getting the closest point in a plane

Where will the fridge magnet jump to when the user releases it? The following handler uses the `GetDistanceToPlane()` to find the closest position rather than the distance:

```
on GetClosestPointInPlane(aPosition,aPointOnPlane,aNormalToPlane)
  -- INPUT:  All parameters must be vectors
  --        <aPosition> and <aPointOnPlane> must be position
  --         vectors
  --        <aNormalToPlane> must not have a length of zero
  -- OUTPUT: Returns the closest point in the plane to aPosition
  ------------------------------------------------------------------

  vNormal   = aNormalToPlane.getNormalized()

  vDistance = GetDistanceToPlane( \
aPosition, \
aPointOnPlane, \
vNormal)

  vPointInPlane = aPosition - vNormal * vDistance

  return vPointInPlane
end GetClosestPointInPlane
```

Mapping a vector to a plane

Imagine that you have an inclined plane with a ball on it. The force of gravity acts vertically on the ball. What proportion of that force acts parallel to the plane to make the ball roll?

Imagine that you have a pole in the ground at an arbitrary angle, and that the sun is directly overhead. What will be the length and direction of the pole's shadow?

You can use following handler to solve both these questions:

```
on GetVectorComponentInPlane(aDirectionVector, aPlaneNormal) ----
  -- INPUT: <aVector> and <aPlaneNormal> must both be direction
  --         vectors
  -- OUTPUT: Returns a vector which represents the component of
  --         aVector at right angles to aNormal
  ---------------------------------------------------------------

  vMagnitude       = aDirectionVector.magnitude
  aDirectionVector = aDirectionVector.getNormalized()
  aPlaneNormal     = aPlaneNormal.getNormalized()

  vCosine          = aDirectionVector.dotProduct(aPlaneNormal)
  vSine            = sqrt(1 - vCosine * vCosine)
  vCrossInPlane    = aDirectionVector.crossProduct(aPlaneNormal)
  vPerpendicular   = aPlaneNormal.crossProduct(vCrossInPlane)
  vPerpendicular.normalize()
  vComponent       = vPerpendicular * vSine * vMagnitude

  return vComponent
end GetVectorComponentInPlane
```

Finding where a ray intersects a plane

The following handler is demonstrated in the movie 3DMathematics.dir. It is useful for many dragging operations.

Note: *Click here for the mathematical explanation.*

```
on RayCutsPlane(aRayOrigin,aRayDirection,aPointOnPlane,aNormal)--
  -- INPUT:  All parameters must be vectors
  --        <aRayOrigin> and <aPointOnPlane> must be position
  --         vectors
  --        <aRayDirection> and <aNormal> must be unit
  --         direction vectors
  -- ACTION: Calculates where a ray starting at aRayOrigin and
  --         travelling in aRayDirection passes through a plane
  --         defined by aPointOnPlane and aNormal
  -- OUTPUT: Returns a list. This may be empty if aRayDirection
  --         is parallel to the plane. If not, it will contain
  --         one position vector representing the intersection
  -- SEE:   <http://softsurfer.com/Archive/algorithm_0104/
  --         algorithm_0104B.htm#Line-Plane%20Intersection>
  ----------------------------------------------------------------

  vIntersection = []

  vRayDotNormal = aRayDirection.dotProduct(aNormal)
  if not vRayDotNormal then
    -- The dot product of two perpendicular vectors is zero. If
    -- the ray and the normal to the plane are at right angles to
    -- each other, they do not intersect (or they intersect
    -- everywhere)
    return vIntersection
  end if

  vPointToOrigin          = aRayOrigin - aPointOnPlane
  vNormalDotPointToOrigin = aNormal.dotProduct(vPointToOrigin)
  vDistanceAlongRay = -vNormalDotPointToOrigin / vRayDotNormal

  vPlanePoint = aRayOrigin + aRayDirection * vDistanceAlongRay

  vIntersection.append(vPlanePoint)

  return vIntersection
end RayCutsPlane
```

Getting a bearing

In a flight simulator, you may wish to display the current compass bearing of the plane. The handler below will provide the appropriate value.

```
on GetMappedBearing(aDirection, aNorth, aUp) --------------------
  -- INPUT: <aDirection> and <aNorth> must both be non-zero-
  --        length direction vectors
  --        <aUp> may be a direction vector. If not, a vertical
  --         yAxis is assumed
  -- ACTION: Projects aNorth and aDirection onto a plane
  --         orthogonal to aUp, then determines the angle between
  --         these two projected directions. Adjusts the angle
  --         if aDirection has a positive component towards the
  --         West
  -- OUTPUT: Returns an angle between 0.0 and 360.0, representing
  --         the angle about the aUp axis that an object facing
  --         in aDirection will have to turn in order to face
  --         North.
  ----------------------------------------------------------------

  if ilk(aUp) <> #vector then
    -- Set aUp to vector(0,0,1) for a world with a vertical zAxis
    aUp = vector(0, 1, 0)
  end if

  -- Map aNorth onto the plane orthogonal to aUp
  vCross  = aNorth.cross(aUp) -- East
  if not vCross.magnitude then
    -- Gimbal lock: aNorth is parallel to aUp. The bearing is
    -- undefined
    return 0.0
  end if

  aNorth = aUp.cross(vCross)

  -- Map aDirection onto the plane orthogonol to aUp
  vCross = aDirection.cross(aUp)
  if not vCross.magnitude then
    -- Gimbal lock: aNorth is parallel to aUp. The bearing is
    -- undefined
    return 0.0
  end if

  aDirection = aUp.cross(vCross)

  -- Find the angle to turn from aDirection around the aUp axis,
  -- in order to face North
  vAngle = aNorth.angleBetween(aDirection)
  vCross = aNorth.crossProduct(aDirection)
  if aUp.angleBetween(vCross) < 90.0 then
    vAngle = 360.0 - vAngle
  end if

  return vAngle
end GetMappedBearing
```

Reflecting a vector in a plane

If you want to bounce a ray or a moving ball of a plane, you can use the following handler:

```
on ReflectVectorInPlane(aVector, aPlaneNormal) ------------------
  -- INPUT: <aVector> and <aPlaneNormal> must both be direction
  --         vectors
  -- OUTPUT: Returns a vector which represents aVector after it
  --         is reflected in a fully elastic collision with a
  --         plane whose orientation is defined by aPlaneNormal
  ----------------------------------------------------------------

  vMagnitude       = aVector.magnitude
  aDirectionVector = aVector.getNormalized()
  aPlaneNormal     = aPlaneNormal.getNormalized()

  vCosine        = -(aDirectionVector * aPlaneNormal)
  vSine          = sqrt(1 - vCosine * vCosine)
  vCrossInPlane  = aVector.cross(aPlaneNormal)
  vPerpendicular = aPlaneNormal.cross(vCrossInPlane)
  vPerpendicular.normalize()
  vComponent     = vPerpendicular * vSine * vMagnitude

  vReflection    = aPlaneNormal * vCosine + vComponent

  return vReflection
end ReflectVectorInPlane
```

Rotating a vector position around an axis

There is no built-in command to rotate a vector. However, you can use a vector as the position property of a transform, and then rotate the transform, as demonstrated in the following handler:

```
on RotateVector(aVector, aCenter, aAxis, aAngle)  ---------------
  -- INPUT: <aVector>, <aCenter> and <aAxis> must be vectors.
  --         aAxis should have a non-zero length
  --        <aAngle> must be a scalar
  -- OUTPUT: Returns a vector representing the position of
  --         aVector after it has been rotated bay aAngle around
  --         an axis parallel to aAxis
  ----------------------------------------------------------------

  vTransform = transform()
  vTransform.position = aVector
  vTransform.rotate(aCenter, aAxis, aAngle)

  return vTransform.position
end RotateVector
```

It can be useful if you want to rotate a face around one of its edges, or to find a point at a given distance from an axis in a given direction. This technique is used in the movie MeshDeform.dir.

Converting a world transform to a local transform

Imagine that you wish to set the world transform of one model to that of another node that has a different parent. Imagine also that you do not want to change the parent of the model that you are moving. You can use the handler below to calculate the transform to apply to the moving model, within its own frame of reference.

This feature is demonstrated in the movie ParentChain.dir.

```
on GetLocalTransform(aNode, aWorldTransform) --------------------
  -- INPUT: <aNode> must be a node object
  --        <aWorldTransform> must be a transform within the
  --         world frame of reference
  -- OUTPUT: Returns a transform in aNode's frame of reference
  --         that will set its world transform to aWorldTransform
  ----------------------------------------------------------------

  vFrameOfReference  = aNode.parent.getWorldTransform()
  vInverse           = vFrameOfReference.inverse()

  vLocalTransform =  vInverse * aWorldTransform

  return vLocalTransform
end GetLocalTransform
```

Get bounding box

To obtain the coordinates of the smallest axis-aligned box that a given model will fit into, use the following handler:

```
on GetBoundingBox(aModel, aFrameOfReference) --------------------
  -- INPUT: <aModel> must be a model
  --        <aFrameOfReference> may be a transform, VOID or
  --         #parent. Any other value will use
  --         aModel.getWorldTransform()
  -- ACTION: Calculates the maximum and minimum positions of the
  --         vertices in aModel, along each of the axes.
  -- OUTPUT: Returns a property list with the format:
  --         [#minX: <float>,
  --          #maxX: <float>,
  --          #minY: -float>,
  --          #maxY: <float>,
  --          #minZ: <float>,
  --          #maxZ: <float>]
  ----------------------------------------------------------------

  vBoundingBox = [:]

  vMinX = the maxInteger
  vMaxX = -vMinX
  vMinY = vMinX
  vMaxY = vMaxX
  vMinZ = vMinX
  vMaxZ = vMaxX

  case ilk(aFrameOfReference) of
    #void: -- use transform()
    #transform:
      vTransform = aFrameOfReference
    #symbol:
      case aFrameOfReference of
        #world:
          vTransform = aModel.getWorldTransform()
        #parent:
          vTransform = aModel.transform
        #self: -- use transform()
      end case
```

```
    otherwise
      vTransform = aModel.getWorldTransform()
  end case

  if not vTransform then
    vTransform = transform()
  end if

  vModifiers = aModel.modifier
  if not vModifiers.getPos(#meshDeform) then
    aModel.addModifier(#meshDeform)
    vRemove = TRUE

  else
    vRemove = FALSE
  end if

  -- Iterate through the meshes
  vMeshCount = aModel.shaderList.count
  repeat with ii = 1 to vMeshCount
    vVertexList  = aModel.meshDeform.mesh[ii].vertexList
    vVertexCount = vVertexList.count

    repeat with jj = 1 to vVertexCount
      vVertex        = vVertexList.getAt(jj)
      vWorldPosition = vTransform * vVertex
      vWorldPosition = vVertexList.getAt(jj)

      vX = vWorldPosition.x
      if vMaxX < vX then
        vMaxX = vX
      end if

      if vMinX > vX then
        vMinX = vX
      end if

      vY = vWorldPosition.y
      if vMaxY < vY then
        vMaxY = vY
      end if

      if vMinY > vY then
        vMinY = vY
      end if

      vZ = vWorldPosition.z
      if vMaxZ < vZ then
        vMaxZ = vZ
      end if
```

```
      if vMinZ > vZ then
        vMinZ = vZ
      end if
    end repeat
  end repeat

  if vRemove then
    aModel.removeModifier(#meshDeform)
  end if

  vBoundingBox[#minX] = vMinX
  vBoundingBox[#maxX] = vMaxX
  vBoundingBox[#minY] = vMinY
  vBoundingBox[#maxY] = vMaxY
  vBoundingBox[#minZ] = vMinZ
  vBoundingBox[#maxZ] = vMaxZ

  return vBoundingBox
end GetBoundingBox
```

Performance

Showing interactive 3D content requires many thousands of calculations for every frame. Every new generation of computers is faster than the previous one, but expectations of what applications can do increases at the same rate.

Animated feature films do not need to be rendered in real-time by a single personal computer. End-users expectations may be influenced by the quality of pre-rendered footage. Nonetheless, they will always prefer an application that responds rapidly to their input to one which looks beautiful, but which fails to react. Knowing what shortcuts you can take to improve the responsiveness of your application can help extend the popularity of your work to users who do not have the most modern hardware.

Optimizing performance

You can use different ways to reduce the number of 3Dcalculations performed. Some techniques may be applicable to all your projects. Others may be useful only in specific cases. Here are some examples.

- Each face that appears in a scene requires its own personal treatment. The fewer faces you use, the faster the scene will display. See "Low-polygon modelling" on page 388 for more details.
- Two scenes that use the same total number of faces can have different performance characteristics depending on the number of models and shaders that they use. See "Shader count and model count" on page 389 for more details.
- Lighting effects, like specular highlights increase the complexity of the calculations required to render each face. See "Specular light" on page 389 for more details.
- The 3D playback engine does not know which models in your scene are invisible. It will spend time rendering all models, even those that are completely hidden by others. If you manually remove undetectable models from the scene, you can reduce rendering time. See "Culling" on page 389 for more details.
- In an animated scene, the user may not have time to focus on all the visual details. In I static scene, the details are important. Switching antialiasing on and off at the appropriate times can improve the end user's overall appreciation of your work. See "Antialiasing" on page 390 for more details.

- Displaying changes on the screen is very time consuming, and it can be distracting. When initializing your world, you may want to prevent any changes from being made visible. See "suspendUpdates" on page 394 for more details.
- Collision detection requires intensive use of the computer processor, especially when there are many moving objects that are close together. Limiting the use of Physics simulations, and deactivating any low-priority Physics elements can improve performance. See "Physics simulations" on page 395 for more details.

  Both JavaScript and Lingo provide multiple ways for achieving the same ends.
- If you have a lengthy operation that needs to be repeated on every frame, it helps to that uses few lines of code is not necessarily faster than one that uses more precise commands. This is especially true when using repeat loops. See "CPU-friendly code" on page 396. It also helps to execute as much of your code as possible while the Director playback engine is not busy doing something else. See "Using frame events wisely" on page 397.

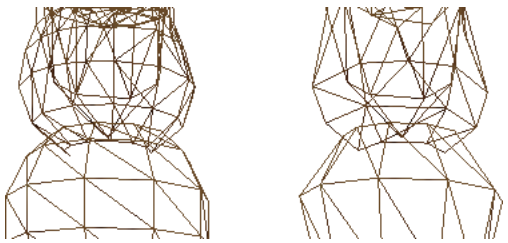
Low-polygon modelling

Each face that appears in a scene requires its own personal treatment. The fewer faces you use, the faster the scene will display.

Creating realistic low-polygon models is a specialized skill. Careful use of textures and normal mapping can lead to very impressive effects. Click here for more information.

Reducing the number of faces at runtime

- If you prefer to work with models that have a higher polygon count, you can use two techniques to reduce the number of polygons that are displayed at run-time. You can use the LOD (Level of Detail) modifier to simplify the geometry of models when they appear small in the distance. See "Level of Detail (LOD)" on page 51. This gives you model-by-model control over appearance.
- You can set a targetFrameRate and set useTargetFrameRate to `TRUE`. The 3D playback will use these settings to reduce the number of faces used to display all models in an attempt to achieve the specified target frame rate. Depending on the scene that you wish to display, you may find that you can increase the frame rate by 5 - 20% using this technique.



useTargetFrameRate merges adjacent faces which have very similar orientations

In certain situations, such as when you use a narrow beam spot light, you may want to increase the polygon count on a low-polygon model. You can use the SDS (SubDivision Surfaces) modifier to do this. See "Subdivision Surfaces (SDS)" on page 51.

Polygons and particle emitters

A single particle emitter can produce many thousands of 2-faced polygons, each of which will be turned towards the camera to be rendered. If you use particle emitters, you may find that you can significantly increase the frame rate of your movie by reducing the number of particles produced. Using non-textured particles can also result in significant improvements in performance.

Shader count and model count

Two scenes that use the same total number of faces can have different performance characteristics depending on the number of shaders and model that they use.

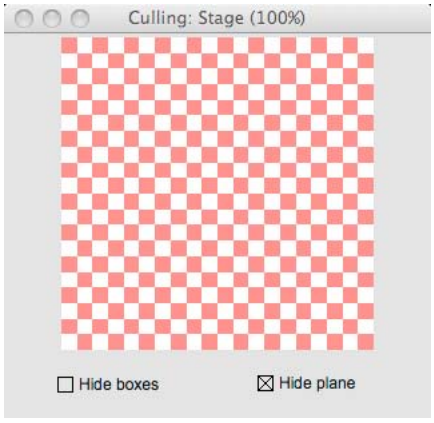
Specular light

Lighting effects, like specular highlights increase the complexity of the calculations required to render each face.

Culling

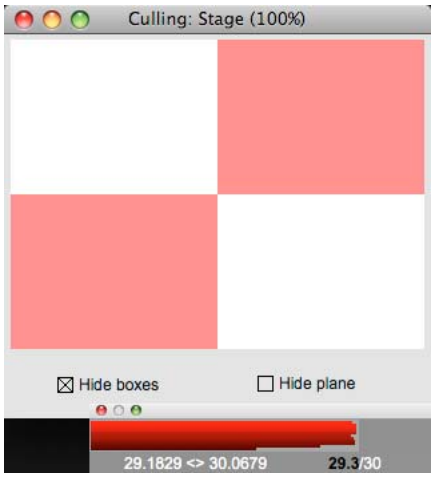
The 3D playback engine does not know which models in your scene are invisible. It will spend time rendering all models, even those that are completely hidden by others. If you manually remove undetectable models from the scene, you can reduce rendering time.

To see a demonstration of this, download and launch the movie Culling.dir. This creates 100 Box models in the distance.



Culling.dir creates a wall of 100 boxes in the distance

It then places a Plane model in the foreground, all but hiding the boxes. Two lines 1 pixel high have deliberately been left visible, so that you will be able to see that the boxes are there.



A plane model prevents the user from seeing the box models

The 3D rendering engine first renders each of the 100 boxes, and then renders the Plane model. The time taken to render the boxes is wasted. If you select the Hide Boxes check box, all 100 boxes are removed from the world. Apart from the two lines of pixels where the boxes have been deliberately left visible, this changes nothing in the final rendering. However, the frame rate is greatly improved.

Removing the hidden boxes from the world improves the frame rate by 50%

Conclusion: If you know that certain models will not be visible from the current camera viewpoint, manually remove them from the world to improve performance. You can return them to the world when the camera moves to a position from where they models will be visible.

Antialiasing

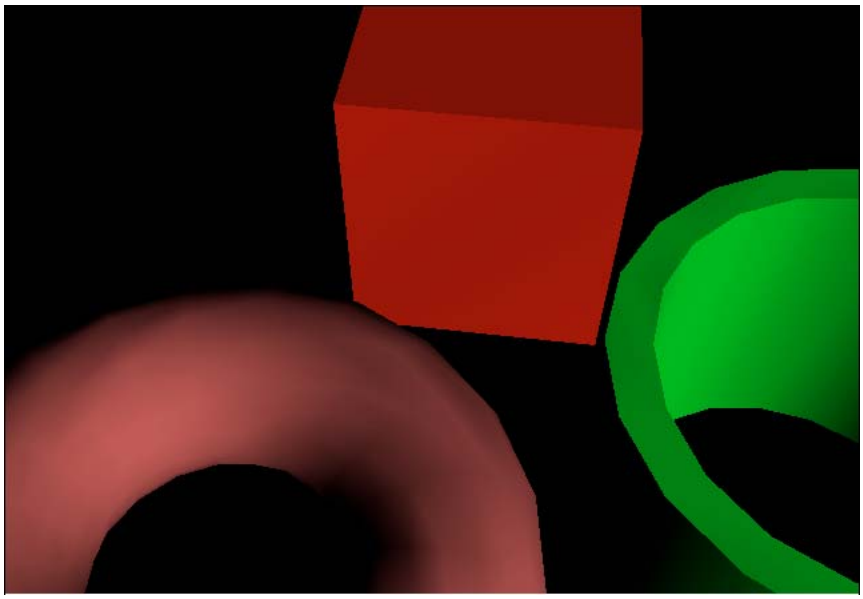
In an animated scene, the user may not have time to focus on all the visual details. In I static scene, the details are important. Switching antialiasing on and off at the appropriate times can improve the end user's overall appreciation of your work.

Using 3D anti-aliasing

In an animated scene, the user may not have time to focus on all the visual details. However, in a static scene, the details are important. Switching antialiasing on and off at the appropriate times can improve the end user's overall appreciation of your work.

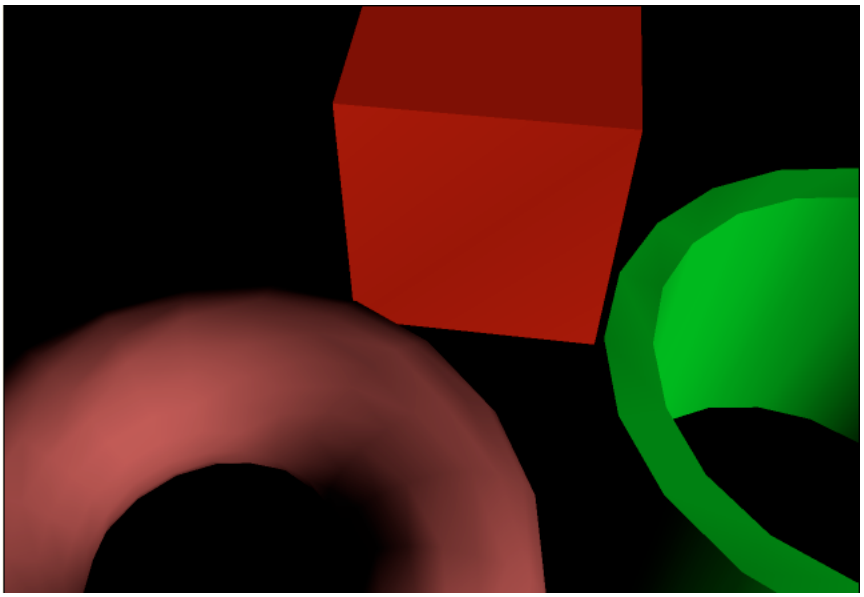
Director gives you the ability to use anti-aliasing with 3D cast members in your movies. Anti-aliasing improves the appearance of graphics by smoothing the lines between shapes or areas of different color so that the lines do not appear jagged. When you use anti-aliasing with a 3D sprite, the edges of each model in the sprite appear smoother against each other and against the background. Anti-aliasing of 3D sprites is particularly well-suited for merchandise demos and other e-commerce applications because its image quality is high and it can be turned on and off, as needed, in real time.

The following scene shows the 3D cast members with anti-aliasing effect turned off.



Anti-aliasing is off

The following scene shows the 3D cast members with anti-aliasing effect turned on.



Anti-aliasing is on

To see a demonstration of this, download and launch the movie Antialiasmode.dir.

Effects of anti-aliasing

An anti-aliased 3D sprite uses more processor power and memory than one that is not anti-aliased, resulting in lower frame rates. Because of this, it is recommended that you turn off anti-aliasing for 3D sprites while any part of the sprite is being moved or animated and turn it back on when the animation is complete. Movies that are designed to animate quickly and games might work better with anti-aliasing turned off. During authoring, movies that use anti-aliasing continue to draw heavily on the processor, even after the movie is stopped. You might want to turn off anti-aliasing each time you stop your movie to ensure that the performance of Director is not affected.

Determining whether anti-aliasing is supported

Not all 3D renderers can perform the additional calculations that anti-aliasing requires. If you have a 3D sprite that you want to anti-alias, check first that the 3D renderer supports anti-aliasing.The renderers that currently support anti-aliasing include the Director software renderer, and DirectX® 5.2, DirectX 7.0, and DirectX 9.

If the 3D sprite is in channel 1 of the Score, test the `antiAliasingSupported` property of sprite 1, as shown in the following example:

```
if sprite(1).antiAliasingSupported = TRUE then
```

Turning on anti-aliasing

If the `antiAliasingSupported` property is `TRUE`, turn on anti-aliasing for the 3D sprite by setting the sprite's `antiAliasingEnabled` property to `TRUE`.

```
sprite(1).antiAliasingEnabled = TRUE
```

For example, if you have a 3D sprite in channel 5 and you want to turn on anti-aliasing for the sprite when it first appears on the Stage, write a `beginSprite` script and attach it to the sprite. Your script should contain code as shown:

```
-- Lingo syntax
on beginSprite
    -- check whether anti-aliasing is supported by the current 3D renderer
    if sprite(5).antiAliasingSupported = TRUE then
        -- if it is, turn on anti-aliasing for the sprite
        sprite(5).antiAliasingEnabled = TRUE
    end if
end beginSprite
// JavaScript syntax
function beginSprite() {
    // check whether anti-aliasing is supported by the current 3D renderer
    if (sprite(5).antiAliasingSupported) {
        // if it is, turn on anti-aliasing for the sprite
        sprite(5).antiAliasingEnabled = true;
    }
}
```

Turning off anti-aliasing

If you plan to animate any part of a 3D sprite, you might want to turn anti-aliasing off temporarily to improve the animation performance. To do this, set the `antiAliasingEnabled` property for the sprite to `FALSE`. Set it back to `TRUE` when the animation is complete.

It is a good idea to turn anti-aliasing on and off on separate handlers. For example, you might want to animate a model, camera, or light while the mouse button is held down and stop the animation when the mouse button is released. In that case, you would turn off anti-aliasing in a `mouseDown` handler and turn it back on in a `mouseUp` handler, as shown in the following example:

```
-- Lingo syntax
on mouseDown
-- user interaction/animation is about to start so turn
-- anti-aliasing OFF
sprite(1).antiAliasingEnabled = FALSE

-- start animation
end

on mouseUp
    -- stop animation

-- the interaction/animation has ended so turn
-- anti-aliasing ON
sprite(1).antiAliasingEnabled = TRUE
end
// JavaScript syntax
function mouseDown() {
    // user interaction/animation is about to start so turn
    // anti-aliasing OFF
    sprite(1).antiAliasingEnabled = false;

    //start animation
}

function mouseUp() {
    // stop animation

    // the interaction/animation has ended so turn
    // anti-aliasing ON
    sprite(1).antiAliasingEnabled = true;
}
```

suspendUpdates

Displaying changes on the screen is very time consuming, and it can be distracting. When initializing your world, you may want to prevent any changes from being made visible. To do this, set the 3D sprite property sprite3D.suspendUpdates to `TRUE`. After you have finished your initialization, set it back to `FALSE`.

During a lengthy initialization process, you will want to provide feedback to the end-user about what is happening. One way to do this is to use perform different initialization steps in an `on enterFrame()` handler, and to update a progress bar at each step.

To see an example of a (faked) initialization sequence that uses `suspendUpdates` to freeze the 3D sprite, download and launch the movie Suspend.dir.

Using suspendUpdates during an asynchronous initialization process

Physics simulations

Collision detection requires intensive use of the computer processor, especially when there are many moving objects that are close together. Limiting the use of Physics simulations, and deactivating any low-priority Physics elements can improve performance.

sleepThreshold

By default, the `sleepThreshold` of all dynamic rigidBody objects in a 3D world is 0.0. Depending on your settings for gravity and restitution, certain objects may never reduce their momentum or `linearVelocity` to zero. Even if you cannot see them move, these objects will be consuming processor time on each frame, testing to see if they have moved relative to the objects closest to them.

You may be able to improve performance by raising the default `sleepThreshold` value for the Physics simulation, and by customizing the `sleepThreshold` and `sleepMode` settings for each individual dynamic rigidBody.

You can also set the `isSleeping` property of low priority rigidBody objects to `TRUE`, to free up processor time for other interactions.

Generating collision callback events

Setting a collision callback handler and enabling collision callbacks requires two lines of code. The result may lead to the creation of up to 5 ContactReport objects for every moving rigidBody on every frame. You may find that many of the collision callbacks that are generated are unnecessary. See Suspend.dir for an example treatment of this issue.

Proxy geometry

The simpler the proxy geometry that you use for rigidBodies and for terrains, the faster the Physics simulation can process them. In some scenes, you may have many complex objects, but not all of them will be the focus of the user's attention all the time. You may find that you can improve performance by using low-detail proxy objects for most parts of the scene.

When the user's attention turns to a particular object, you can remove the low-detail proxy and replace it with a high-detail proxy for the same model.

Terrains

Where possible, use several small terrain objects, placed side by side. Using a single large terrain can lead to performance issues. For your own projects, you may wish to find the optimal balance between multiple terrain objects, the terrain dimensions and the polygon count for each individual terrain.

Ray casting

Using `rayCastAll()` and `rayCastClosest()` are both significantly faster than the native 3D ray-casting methods. However, even with the simplified geometry used by rigidBody proxies, ray casting can be an expensive operation if the scene is full of rigidBody objects. Unlike the native 3D ray-casting methods, the Dynamiks xtra does not allow you to limit the number of objects that the ray will consider, nor does it allow you to indicate a maximum distance at which to stop searching.

If you are sending multiple ray casts on every frame into a crowded scene, you may like to check whether this is affecting the rate of playback. If so, you may wish to explore alternative techniques. Here are a couple of examples.

- Create a parallel scene with 3D models with very simple geometry and remove it from the world. Use the native member3D.modelsUnderRay() on a subset of these simple models.
- Use a path-finding algorithm, such as A* (see "Finding a path" on page 229) to find a pre-determined path through a maze of obstacles.

CPU-friendly code

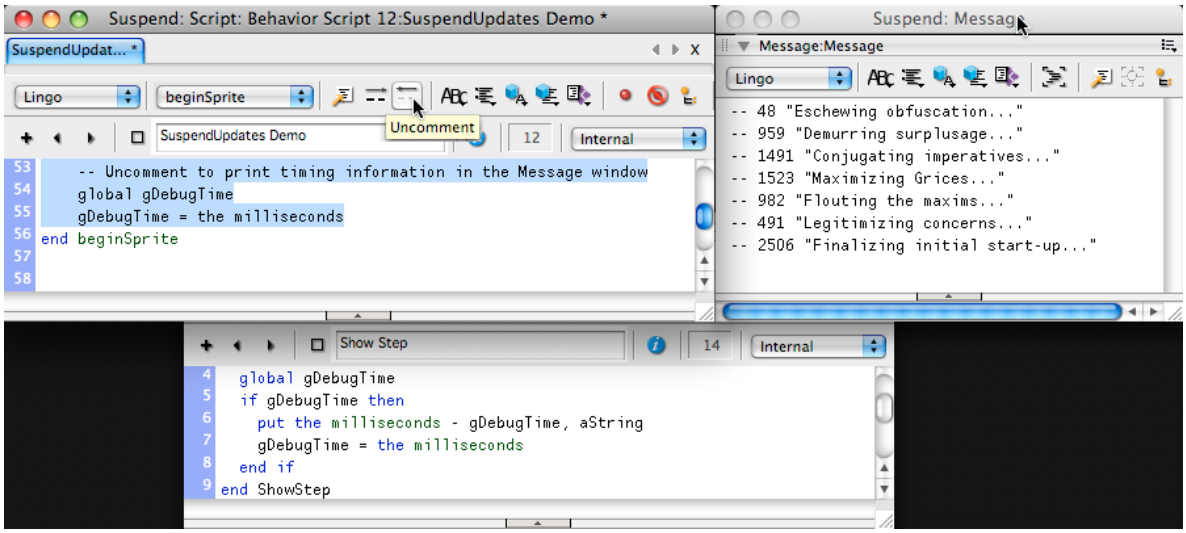
3D operations may not be the only bottleneck in your project. You may unwittingly be creating scripts that use inefficient techniques.

Both JavaScript and Lingo provide multiple ways for achieving the same ends. A script that uses few lines of code is not necessarily faster than one that uses more precise commands. This is especially true when using repeat loops.

Placing timer points in your scripts

In any complex process, there may be certain operations that take longer than others. The most likely places to improve performance are in those operations that take the longest. To determine where these are, you can place timer points in your script.

To test this concept, download the movie Suspend.dir. At the end of the `on beginSprite()` handler in the SuspendUpdates Demo behavior, you will find three lines commented out. Uncomment these lines and launch the movie. Watch the output in the Message window.



Using a debug timer to indicate the elapsed time at the beginning of each operation

You can use a similar technique in your own projects to identify the areas where it would be useful to try to save time.

Comparing potential solutions

If you are aware that there is more than one way to achieve the same goal, it is worthwhile testing which alternative is the most efficient. If you download the movie Suspend.dir, you can find a Movie Script named Test. This is designed to compare two different approaches and print out information in the Message window about which one is faster.

The test() handler is currently written to compare these two operations:

- `x = aList.getAt(aList.count)`
- `x = aList.getLast()`

You can test this by typing the following command in the Message window.

```
test(1000000)
```

This command will run each of the two lines of code above one million times on a list with one million items. Here is a typical result:

```
test(1000000)
 -- 195 #lastItem
 -- 104 #lastItem
```

The test shows that `aList.getLast()` is almost twice as fast as the alternative. This case is rather trivial; it tells you how to shave fractions of a nanosecond off an operation on a list. However, you can edit the script to test cases that are of importance for your own projects.

Using frame events wisely

If you have a lengthy operation that needs to be repeated on every frame, it helps to execute this while the Director playback engine is not busy doing something else.

Every frame, Director generates a series of events that are sent to all sprites and to the first Movie Script that contains a handler for the event.

- `#enterFrame`: sent after the image of the frame has been displayed on the screen. All sprites and 3D nodes will report their new positions accurately.
- `#exitFrame`: sent while the current frame is still on the screen, after the Director has registered the new position of the mouse, but before any adjustments have been made to the position of sprites or 3D nodes
- `#stepFrame`: sent only to scripts and script instances placed on the actorList. See "actorList and #stepFrame events" on page 426.
- `#prepareFrame`: sent while the current frame is still on the screen but before modifying the positions of sprites and 3D nodes in preparation for displaying the image of the next frame.

  Immediately after the `#prepareFrame` event, Director plays sounds, updates the position of 3D nodes, draws sprites, and performs any transitions or palette effects. When this is complete `#enterFrame` is then sent again.

Between the `#prepareFrame` and the following `#enterFrame` event, the Director playback engine is calculating the value of each pixel to display on the screen. Any code that executes during this period increases the time it takes to display the next screen.

If the movie is playing at 50 frames per second, each frame will be visible on the screen for 20 milliseconds (= 1000 ms / 50 frames per second). The Director playback engine will aim to trigger a new `#exitFrame` event 20 milliseconds after the previous one.

Between `#enterFrame` and `#exitFrame`, the Director playback engine is "idle". It has time to wait until the next frame is ready to be drawn. During this time, the playback engine can process a large quantity of code without slowing down the process of redrawing the screen. It is in the period between `#enterFrame` and `#exitFrame` that many other events are generated. These include:

- `#mouseEnter, #mouseWithin, #mouseDown, #mouseUp, #mouseLeave, #mouseUpOutside`
- `#keyDown, #keyUp`
- Callbacks from `timeOut` objects
- Callbacks from 3D `#timeMS` events

enterFrame

If performance is critical to your project, place any lengthy code that needs to be executed on every frame in an `on enterFrame` Lingo handler. This is especially true if the code involves updating the appearance or position of sprites on the screen.

However, an `on enterFrame()` handler is not good location for code that deals with moving an item in synchronization with the mouse. The lag between the moment the operating system updates the position of the mouse pointer and the moment when Director updates the Stage is most obvious if the Stage updates are made in an `on enterFrame()` handler.

exitFrame

The `#exitFrame` event is sent just before the process for calculating the new Stage image starts. Any code executed as the result of an `#exitFrame` event will delay the update process, in the same way that looking for you keys just before you leave the house delays your departure. Limit your use of your `on exitFrame()` handlers to navigational calls to _movie.go() or _movie.play().

stepFrame and prepareFrame

If you do not make any changes to the sprites on the Stage, you may see a very slight difference in performance if you use `#stepFrame` or `#enterFrame` instead of `#enterFrame`. The difference may be so small that it is difficult to detect over the natural variations in playback speed.

However, if you change the position or appearance of any sprites in an `on stepFrame()` or `on prepareFrame()` handler, then the effect on performance can be even worse than using `on exitFrame()`.

Nonetheless, an `on stepFrame()` or `on prepareFrame()` handler is a good location for code which synchronizes the movement of an item with the mouse pointer. Director asks the operating system for the current position of the mouse immediately before sending out the `#stepFrame` event, so the lag between the position of the mouse and the position of a synchronized sprite will be least if the position of the sprite is updated in an `on stepFrame()` or `on prepareFrame()` handler.

Chapter 5: Audio mixers and sound objects

The Audio Mixer Inspector window provides a visual interface for managing the Mixer members in your movies. The Audio Mixer Inspector allows you to work with three different types of object:

- A *mixer* is a container that mixes sounds and applies filters to the resulting mix to create a variety of effects. It provides an efficient way to play back multiple sounds simultaneously. See "Audio mixers" on page 400.
- A *sound object* defines a sound source. This can be a cast member, a file on the end-user's hard disk, or a sound streaming over a network connection. It can also be the sound track of an MP4 video file. Each individual sound object can have its own settings and filters. See "Sound objects" on page 402.
- A *filter* is a plug-in audio editor that applies an effect such as echo or distortion to audio samples as they are played back.

Audio mixers and sound objects can use a panMatrix to control the volume of playback on up to six speaker channels. You can use mixers and sound objects to manage output for multiple sound playback systems, including mono, stereo, 2.1, 3.1, 4.1, and 5.1 sound.

This section describes how to work with the Audio Mixer Inspector. It also gives a glimpse at other features of audio features that are only accessible through Lingo or JavaScript syntax.

Audio mixers

A mixer is container that mixes the sound objects that it contains and plays the resulting output. Because multiple audio sources are merged into a single audio source, mixers save resources by reducing the amount of data transferred to the sound card.

You can use a mixer to mix audio sources with the same or different sampling rates, bit depth, or number of channels. The audio sources for a mixer can be from different musical instruments, vocalists, members of an orchestra, announcers, journalists, crowd noises, and so on. You can apply audio filters to the output of a mixer to create a range of effects.

You can find two types of mixer in a Director movie.

- Mixers can be stored as a cast member with the type `#Mixer`. You can use the Audio Mixer Inspector to modify Mixer members. See "The Audio Mixer Inspector" on page 405 for details. When you save your movie, the current state of all Mixer members is saved.
- Every `#mp4` member has a mixer property, and so do sprites which display `#mp4` members. You cannot modify the mixer of an `#mp4` member, but you can use Lingo and JavaScript syntax to make run-time changes to the mixer of an `#mp4` sprite. Run-time changes made to `#mp4` sprites are not saved when the movie is saved. In Director 11.5.8, the Audio Mixer Inspector can not be used to display the properties of an `#mp4` mixer. See "Mixing MP4 movie sound with other sounds" on page 419 for more details.

Creating a new mixer

You can create a new Mixer member in three different ways:

- Select the menu item Insert > Media Element > Mixer
- Use the [+] button in the Audio Mixer Inspector. See "The Audio Mixer Inspector" on page 405 for details

- Use the `new()` or _movie.newMember() methods with the symbol `#Mixer`

Mixer methods

You can use all the standard cast member methods on a Mixer member. See the method summary table at Member. Mixer members also have 13 type-specific methods, as listed below.

Use the following methods to manage the sound objects that are played back by a mixer:

- mixer.createSoundObject()
- mixer.deleteSoundObject()
- mixer.getSoundObject()
- mixer.getSoundObjectList()

Use the following methods to control playback of the sound mix:

- mixer.mute()
- mixer.play()
- mixer.pause()
- mixer.stop()
- mixer.unmute()

Use the following methods to save the mixed sound to a file on a local drive:

- mixer.save()
- mixer.startSave()
- mixer.stopSave()

Use the following method to revert the mixer to its last saved state:

- mixer.reset()

Mixer properties

You can use all the standard cast member properties with a Mixer member. See the property summary table at Member. Mixer members also have 15 type-specific methods, as listed below.

The following properties are read-only:

- mixer.channel
- mixer.elapsedTime
- mixer.filterList
- mixer.isSaving
- mixer.soundObjectList
- mixer.status

In Director 11.5.8, when you attempt to set one of the above properties, a script error occurs.

Use the following properties to arrange the output sound spatially on systems with multiple speakers.

- mixer.channelCount
- mixer.panMatrix

- mixer.toChannels
- mixer.useMatrix

Currently, `panMatrix`, `toChannels` and `useMatrix` share a single entry in the Scripting Dictionary. In the alphabetical Scripting Dictionary, these will need to be divided into separate entries.

The following properties can only be set while the mixer has a `status` of `#stopped`. Attempts to change the value of these properties will fail silently if the mixer is not stopped.

- mixer.bufferSize
- mixer.numBuffersToPreload
- mixer.sampleRate
- mixer.bitDepth

You can set the `volume` of a mixer at any time using mixer.volume.

The Audio Mixer Inspector does not give you access to the following properties; You need to use Lingo or JavaScript syntax with these properties:

Read-write properties:

- mixer.panMatrix
- mixer.useMatrix

Read-only properties:

- mixer.channel
- mixer.elapsedTime
- mixer.isSaving
- mixer.status

Sound objects

A sound object defines the audio data that is added to a mixer. Sound objects contain the actual sound data that is to be played. Sound data may come from a variety of sources:

- An audio file (either stored locally or retrieved over a network connection)
- An audio cast member
- A streaming cast member
- A byteArray

You can add audio filters to a sound object to create a range of effects. These effects are applied before the sound data is transferred to the mixer for playback. Additional audio filters can be added to the mixer.

A sound object cannot exist independently of a mixer. If you delete a sound object from its parent mixer, any variables that refer to the sound object will be set to `VOID`. You can, however, transfer a sound object from one mixer to another. When you save your movie, the current settings for all the sound objects will be saved in their parent mixers.

Creating a new sound object

You can create a new sound object in two different ways:

- Use the [+] button in the Audio Mixer Inspector. See "Adding a sound object to a mixer" on page 407 for details
- Use the mixer.createSoundObject() method

Sound object methods

Use the following methods to modify the sound object:

- soundObject.moveTo(): moves the sound object to a different mixer
- soundObject.replaceMember(): changes the cast member audio source

Note: *In Director 11.5.8, `soundObject.replaceMember()` fails unless a second parameter with the value `[#loopCount: 0]` is used. This will lead to the sound repeating continuously. There are also issues with setting `soundObject.loopCount = 1`. If you want to change the sound member without looping the sound forever, the simplest workaround is to use a handler like the one below.*

```
on ReplaceSoundObjectMember(aSoundObject, aMember)
  vName = aSoundObject.name
  vMixer = aSoundObject.mixer
  vFilterList = aSoundObject.filterList
  vMixer.deleteSoundObject(aSoundObject)
  vSoundObject = vMixer.createSoundObject(vName, aMember)
  vCount = vFilterList.count
  repeat with ii = 1 to vCount
    vFilter = vFilterList[ii]
    vSoundObject.filterList.append(vFilter)
  end repeat
  return vSoundObject
end ReplaceSoundObjectMember
```

Note that this code will replace any customized values for the properties of the sound object with default values. You may need to use a similar technique to the one used for audio filters to transfer the custom property values from the old sound object to the new one.

Use the following methods to control playback of a sound object:

- `soundObject.breakLoop()`
- `soundObject.mute()`
- `soundObject.pause()`
- `soundObject.play()`
- `soundObject.seek()`
- `soundObject.stop()`
- `soundObject.unmute()`

Use the following methods to save the sound object to a file on a local drive:

- `soundObject.save()`
- `soundObject.startSave()`
- `soundObject.stopSave()`

Use the following methods to work with callbacks:

- `soundObject.registerByteArrayCallback()`
- `soundObject.registerCuePointCallback()`
- `soundObject.unregisterByteArrayCallback()`
- `soundObject.unregisterCuePointCallback()`
- `soundObject.unregisterEndOfSpoolCallback()`

Sound object properties

The following properties are read-only. In Director 11.5.8, attempting to set one of these properties may provoke a script error.

- `soundObject.bitDepth`
- `soundObject.channelCount`
- `soundObject.connectionStatus`
- `soundObject.currentTime` (see `soundObject.seek()`)
- `soundObject.duration`
- `soundObject.elapsedTime`
- `soundObject.isSaving`
- `soundObject.filterList`
- `soundObject.loopsRemaining`
- `soundObject.member`
- `soundObject.mixer`
- `soundObject.mostRecentCuePoint`
- `soundObject.percentStreamed`
- `soundObject.sampleCount`
- `soundObject.sampleRate`
- `soundObject.status`

You can set the following properties of a sound object.

- `soundObject.name`
- `soundObject.preloadTime`

Use the following properties to set how the sound source is played back:

- `soundObject.playRate`
- `soundObject.startTime`
- `soundObject.endTime`
- `soundObject.loopCount`
- `soundObject.loopStartTime`
- `soundObject.loopEndTime`
- `soundObject.volume`

Use the following properties to set how the different channels in the sound object are played through the available speakers:

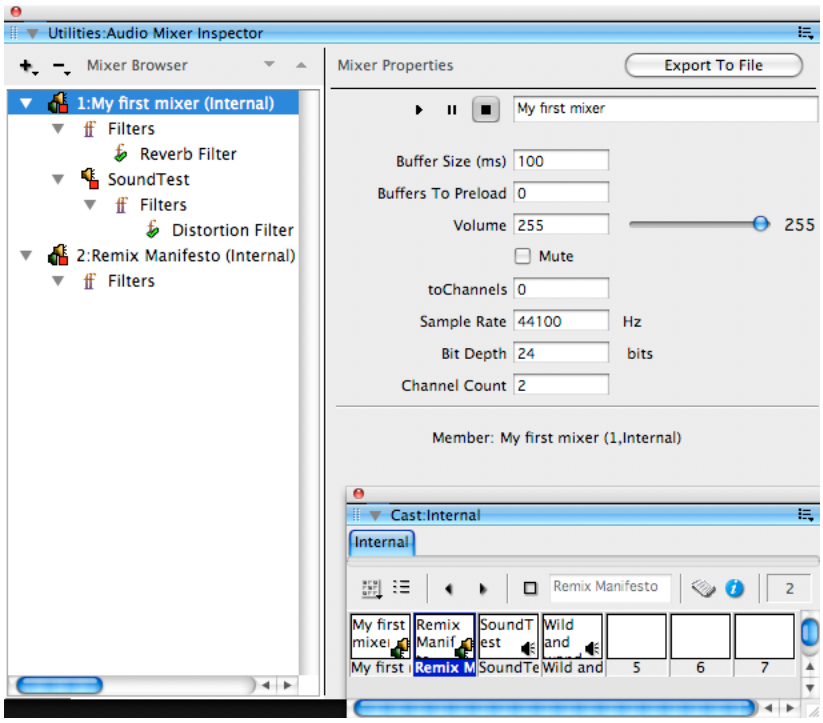
- `soundObject.panMatrix`
- `soundObject.toChannels`
- `soundObject.useMatrix`

The Audio Mixer Inspector

You can open the Audio Mixer Inspector in the following ways:

- Select the menu item Window > Audio Mixer.
- Press Ctrl+Shift+X (Microsoft® Windows XP, Windows Vista, or Windows 7) or Command-Shift-X (Mac OS X).
- Double-click on a Mixer member in the Cast window.
- Select a Mixer member in the Cast window and press the Return key.

If you use either of the last two methods when the Audio Mixer Inspector window is already open, the entry associated with the Mixer member will be selected in the Mixer Browser pane.



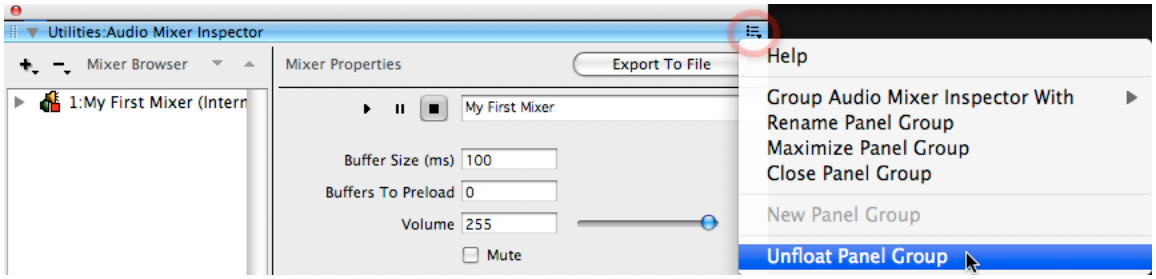
The Audio Mixer Inspector window showing the hierarchy of objects inside a mixer

The Audio Mixer Inspector is divided into two panes. The Mixer Browser in the left pane lets you select one of the various Mixer members in your movie, and browse its hierarchy of sound objects and filters. The Mixer Browser is where you select the object that you wish to inspect.

In the Properties pane on the right, you can view most of the properties of the selected object. You can set the values of any read-write properties that are displayed. The Export To File button will be disabled if the currently selected object is not a mixer or a sound object, or if the selected object or its parent object is currently playing.

Floating or document window

By default, the Audio mixer Inspector will be set to float above other windows. If you prefer to view other document windows in front of it, click on the contextual menu button at the top right corner of the window, and select the Unfloat Panel Group option.

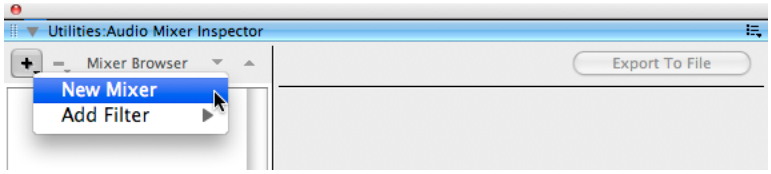


Making the Audio Mixer Inspector behave like a document window

Creating a new mixer

If you currently have no Mixer members in your movie, both panes will be blank.

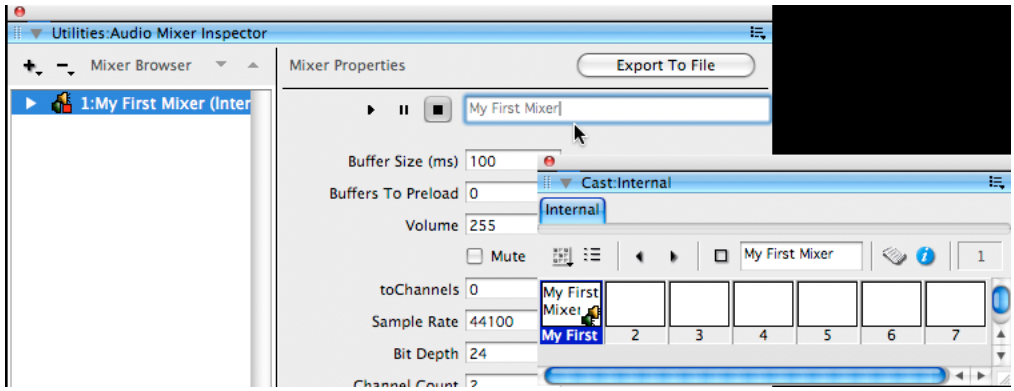
To create a new Mixer member, click the [+] button, and select New Mixer. A new Mixer member will be created and added to the active Cast library. The new mixer will be listed in the Mixer Browser area along with its cast member number.



Creating a new mixer in the Audio Mixer Inspector

Naming a mixer

The new Mixer member will have an empty name. It will not be selected automatically. To name the mixer, select it in the left Pane, and then enter a name in the text box in the right pane. This will automatically set the name for the new member. Alternatively, you can name the member itself in the Cast window.



Naming a mixer

Mixer properties

With a mixer selected, the right pane will allow you to set the following properties:

- `mixer.name`
- `mixer.bufferSize`
- `mixer.numBuffersToPreload`
- `mixer.volume`
- `mixer.toChannels`
- `mixer.sampleRate`
- `mixer.bitDepth`
- `mixer.channelCount`

Note: *In Director 11.5.8, you will not be able to set the values for mixer.panMatrix or mixer.useMatrix using the Audio Mixer Inspector. You will need to use Lingo or JavaScript syntax to set these properties.*

You will also be able to play, pause, stop and mute the mixer.

Removing a mixer

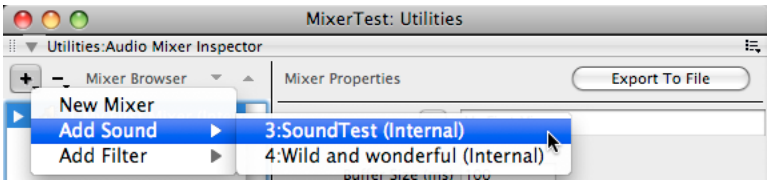
You cannot remove a mixer from within the Audio Mixer Inspector. To remove a Mixer member, you need to select the Mixer member in the Cast window and select the menu item Edit > Clear Cast Members, or simply press the Delete or Backspace key.

Adding a sound object to a mixer

You can add audio cast members as sound objects to a mixer. If the current movie does not contain any audio cast members, import one or more audio cast members by selecting the menu item File > Import.

You can use one of the following ways to add an audio cast member as a sound object to a mixer:

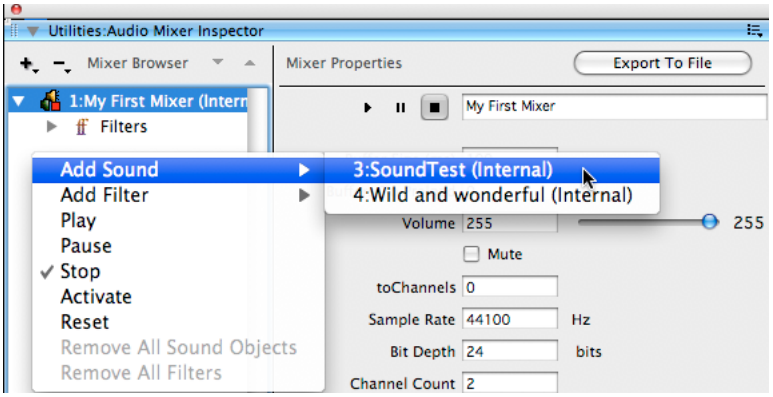
- Select a mixer in the Mixer Browser area and click [+].
- From the Add Sound submenu, select the audio cast member that you want to add as a sound object.



Selecting an audio cast member to use as a sound object using the [+] button

The alternative method is:

1. Select a mixer in the Mixer browser area.
2. Right-click (Windows) or Ctrl-click (Macintosh) anywhere in the left pane.
3. In the contextual menu, click Add Sound, and then click the audio cast member that you want to add as a sound object.



Selecting an audio cast member to use as a sound object using the Mixer Browser contextual menu

Sound object properties

With a sound object selected, the right pane will allow you to set the following properties:

- `soundObject.startTime`
- `soundObject.endTime`
- `soundObject.preloadTime`
- `soundObject.loopCount`
- `soundObject.loopStartTime`
- `soundObject.loopEndTime`
- `soundObject.volume`
- `soundObject.toChannels`
- `soundObject.playRate`

You can also see the values for the following read-only properties:

- `soundObject.member`
- `soundObject.duration`
- `soundObject.sampleRate`

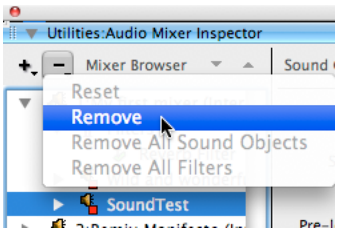
- `soundObject.bitDepth`
- `soundObject.channelCount`

Note: *In Director 11.5.8, you will not be able to set the values for soundObject.panMatrix or soundObject.useMatrix using the Audio Mixer Inspector. You will need to use Lingo or JavaScript syntax to set these properties.*

Removing a sound object from a mixer

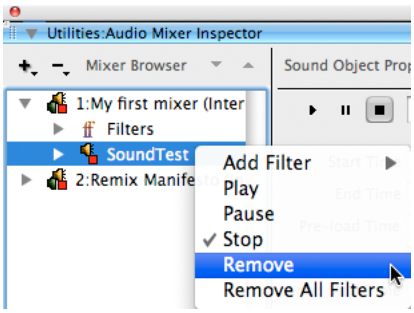
To remove a sound object from a mixer, you can choose one of the following techniques:

- Select the sound object, click the [-] button, and choose Remove Item from the pop-up menu.



Using the [-] button to remove a sound object from a mixer

- Right-click (Windows) or Ctrl-click (Macintosh) on the sound object. In the contextual menu, select Remove.



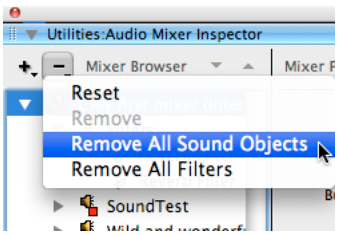
Using the contextual menu to remove a sound object from a mixer

- Alternatively, select the sound object, and then press the Delete or Backspace key.

Removing all sound objects

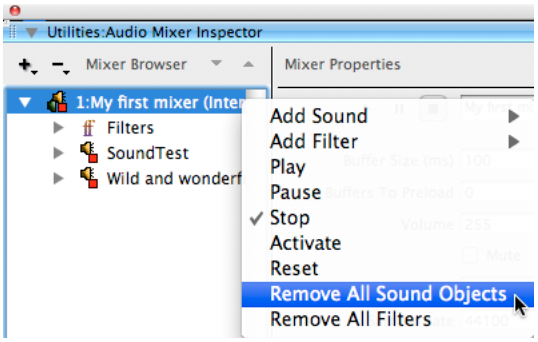
If you want to remove all the sound objects from a given mixer, you can use the following additional methods:

- Select the mixer in the Mixer Browser pane. Click the [-] button, and then select Remove All Sound Objects from the popup menu.



Using the [-] button to remove all sound objects from a mixer

- Right click (Windows) or Ctrl-click (Macintosh) on the mixer in the left pane. Select Remove All Sound Objects from the contextual menu.



Using the contextual menu to remove all sound objects from a mixer

Applying filters to a sound object or mixer

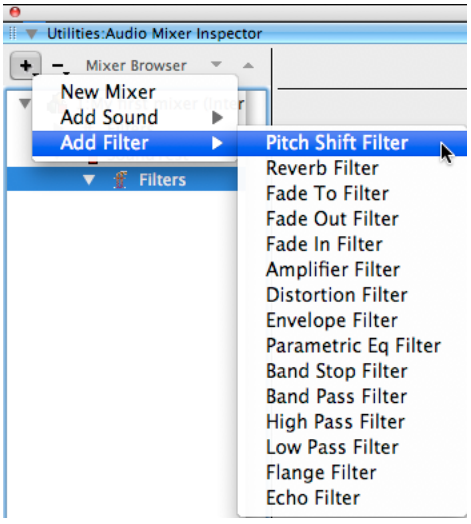
You can apply filters both to mixers and to individual sound objects within a mixer. When you apply filters to a mixer, the filters are applied to all the contained sound objects, in addition to any filters applied directly to the individual sound objects.

You can add a filter to a mixer or sound object in two different ways:

1 Select a mixer or sound object

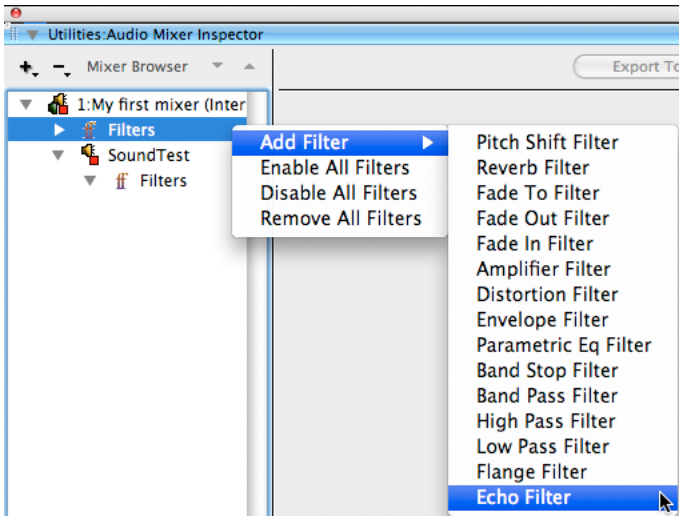
2 Click [+].

3 From the Add Filter submenu, select the filter that you want to apply to the mixer or sound object.



Using the [+] button to add a filter

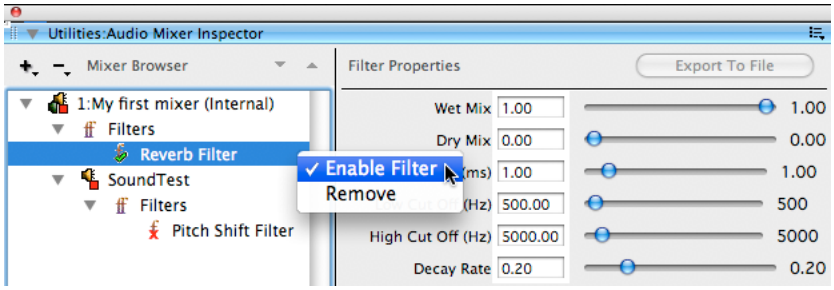
Alternatively, right-click (Windows) or Ctrl-click (Macintosh) on a mixer or Sound object, or on the filters node of a mixer or sound object. Choose a filter from the Add Filter submenu.



Using the contextual menu to add a filter

Enabling and disabling filters

You can enable and disable individual filters. To do so, right-click (Windows) or Ctrl-click (Macintosh) on the filter and select or deselect the Enable Filter menu item.



Using the contextual men to enable or disable filters individually

Note that the icon of a disabled filter changes, to display a red x.

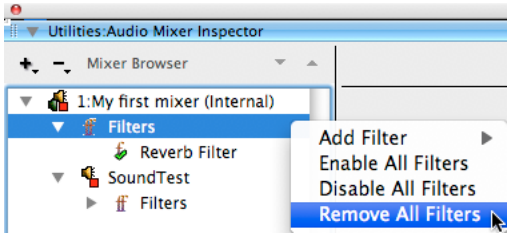
Applying an action to multiple filters

To apply an action to all the filters of an object, right-click (Windows) or Ctrl-click (Macintosh) on the filters node of a mixer or sound object in the Mixer Browser area, and select one of the following:

Enable All Filters Enables all filters applied to the mixer or sound object

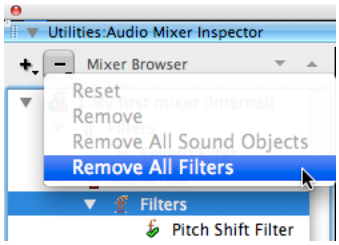
Disable All Filters Disables all filters applied to the mixer or sound object

Remove All Filters Removes all filter effects applied to the mixer or sound object



Applying an action to all filters for a given object

You can also remove all filters for a given object by selecting the object or its filter node, clicking the [-] button, and selecting Remove All Filters.



Using the [-] button to remove all filters

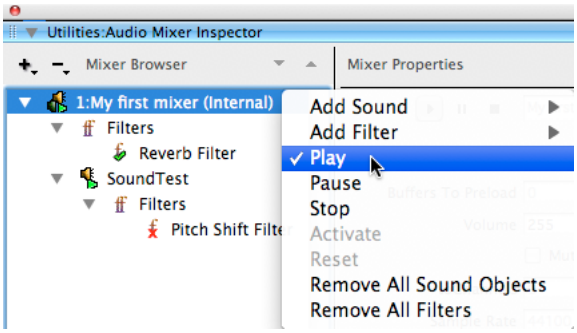
***Note:** Filters enabled for the mixer or sound object are applied to it before playback. You can also apply filters to a mixer or its contained sound objects at run time, using Lingo or JavaScript syntax.*

Playing a mixer or sound object

To hear a mixer or sound object play back, you can do one of the following:

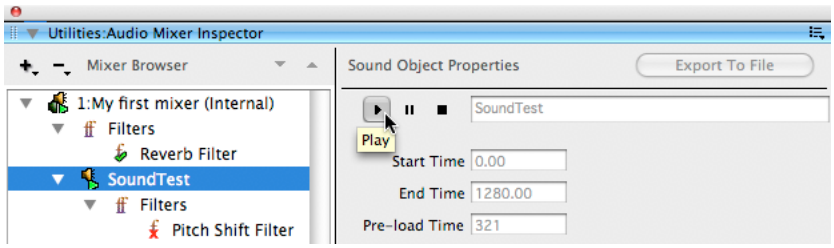
1 Right-click (Windows) or Ctrl-click (Macintosh) on a mixer or sound object in the Mixer Browser area.

2 Select Play, Pause, or Stop from the pop-up menu.



Using the contextual menu to control playback

Alternatively, select a mixer or sound object. Then use the controls (Play, Pause, and Stop) in the upper-left corner of the mixer/sound object



Using the playback controls for a mixer or sound object

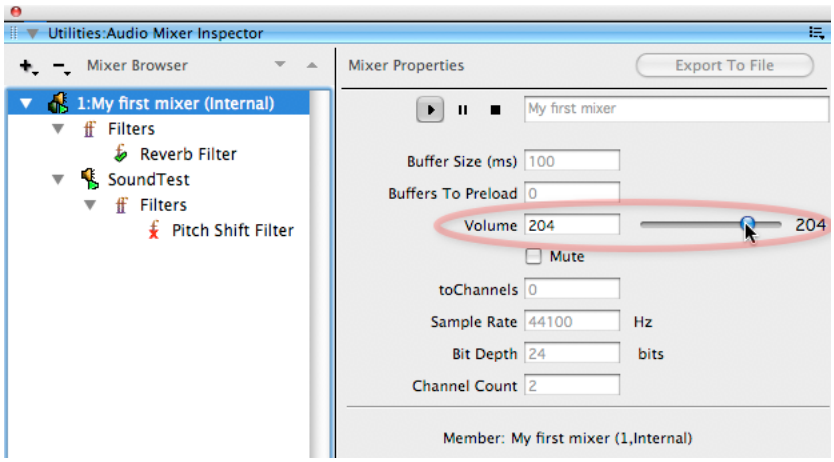
Playing a mixer is not identical to playing the sound objects that it contains. Any sound objects that are not set to loop indefinitely will eventually stop. The mixer will continue playing, even if all its component sound objects have stopped.

When you start playing back a sound object, its parent mixer will also start playing.

Note: *This is not the same behavior as when you use the script method soundObject.play() at runtime. This method will have no effect unless the parent mixer is already playing. See "Activating a mixer" on page 416 for more information.*

Disabled properties

During playback, all configurable properties except for volume will appear disabled in the Audio Mixer Interface window. This is the `status` of both the object needs to be `#stopped` before setting these properties will have any effect.

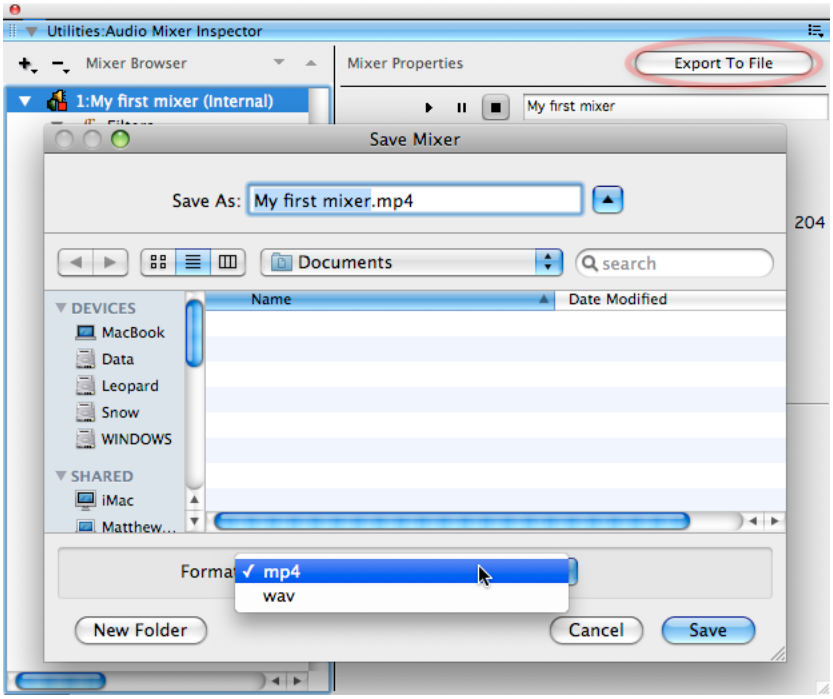


During playback, only the volume property can be set

Exporting a mixer or a sound object

You can export a mixer or sound object as a WAV or MP4 audio file, along with filter effects.

1. Select a mixer or sound object in the Mixer Browser area.
2. Click Export To File.
3. Specify a filename and a file type.
4. Click Save.

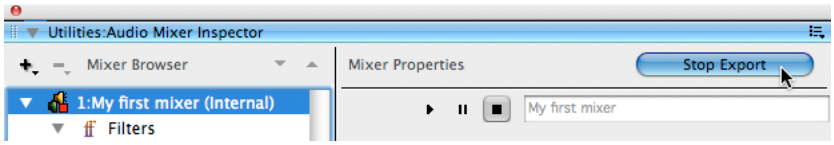


Exporting a mixer or sound object as a WAV or MP4 file

The time required to save a file depends on its size. In any case, it is faster than playing the sound in real time.

If the `loopCount` property of any of the sound objects is set to 0, then the export will continue until the target disk is full or until you press the Stop Export button, whichever is sooner.

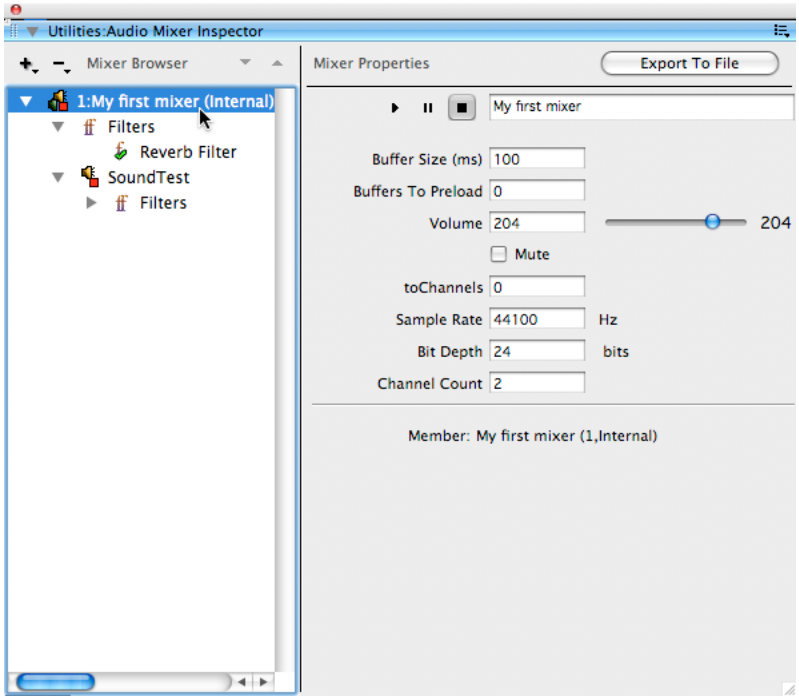
Functionalities like play, pause, and stop are disabled while a file is being saved. While a file is being saved, the Export To File button changes to Stop Export.

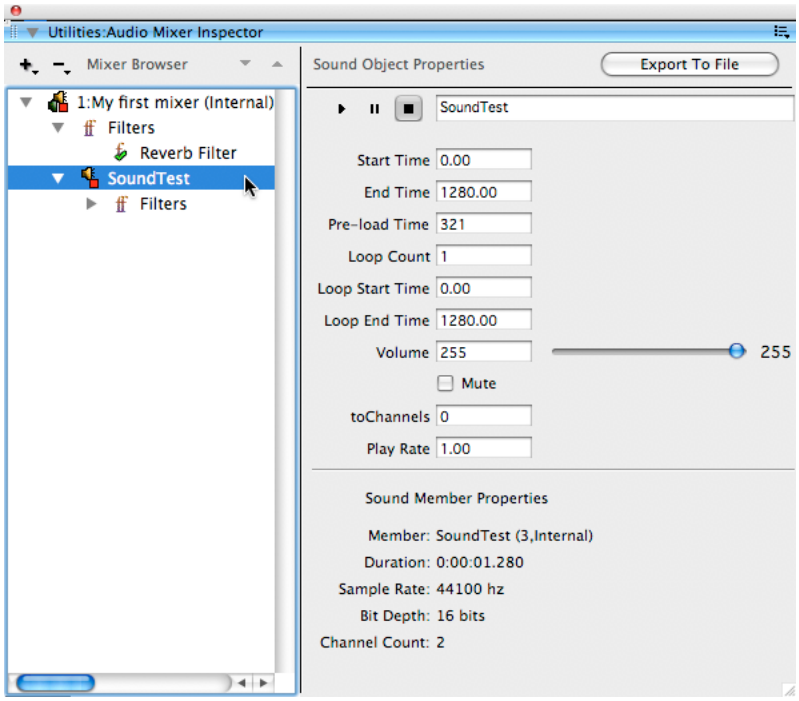


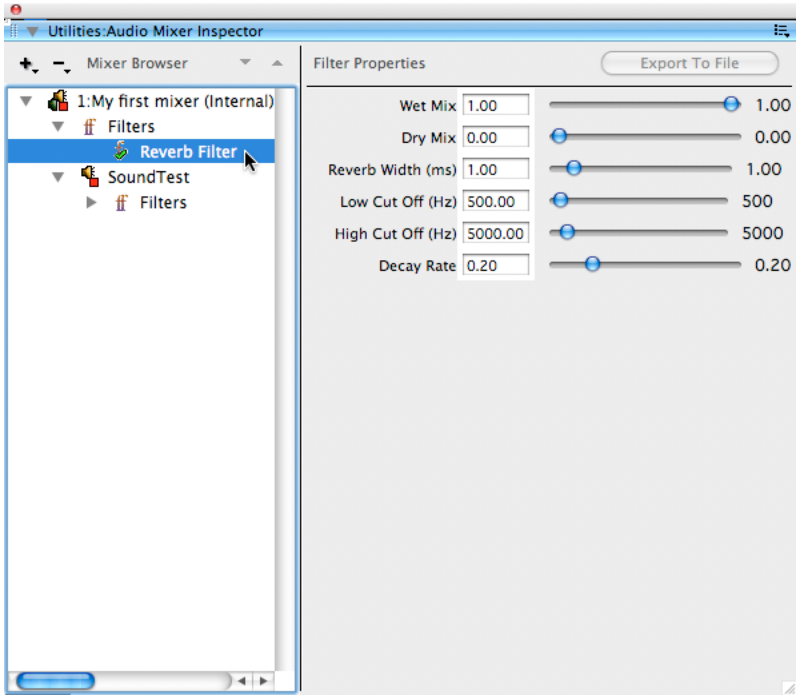
Click the Stop Export button to cancel the export operation.

Modifying mixer, sound object, or Filter properties

Select a mixer, sound object, or a filter in the Mixer Browser area.







Modify the properties displayed in the right pane. For example, you can change the `loopCount` of a sound object from 1 to 3.

Note: *Changes to filter properties apply at run time. Such changes are applicable only for the mixer or sound object to which the filter is directly applied.*

To rename a mixer or sound object, enter a new name in the box to the right of playback controls.

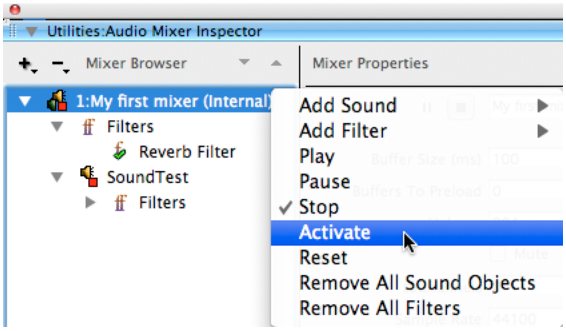
Activating a mixer

Activating a mixer sets the `state` of the mixer to `#playing`, but it does not start playing the sound objects contained in the mixer automatically. You can activate a mixer only if it is in the `#paused` or `#stopped` state.

Activating a mixer is similar to calling `mixer.play([])` from a script. Do the following to activate a mixer:

1 Right-click (Windows) or Ctrl-click (Macintosh) on the mixer in the Mixer Browser pane.

2 Select Activate from the pop-up menu.



Using the contextual menu to activate a mixer

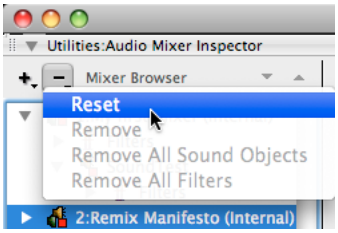
See mixer.play() for more details on this feature.

Resetting a mixer

When you reset a mixer, any changes made to the Mixer member or its contained sound objects since the last save operation are discarded. You will not see any confirmation dialog. You cannot undo the Reset operation.

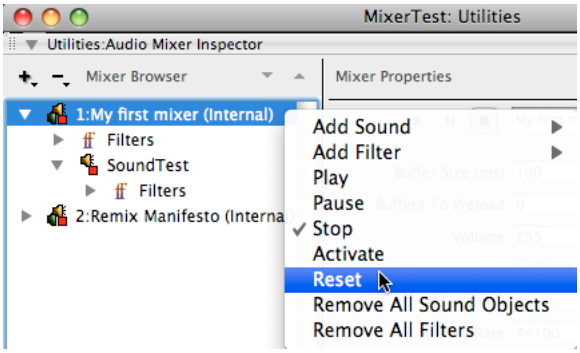
You can reset a mixer in two different ways:

- Select a mixer in the Mixer Browser area. Click [-] and select Reset Mixer from the pop-up menu.



Using the [-] menu to reset a mixer

- Right-click (Windows) or Ctrl-click (Macintosh) on a mixer. In the contextual menu that appears, select Reset



Using the contextual menu to reset a mixer

***Note:** In Director 11.5.8, the Reset menu option will appear enabled even if you have not made any changes to the mixer.*

Creating a mixer asset reference

When using Lingo or JavaScript syntax with mixers and soundObjects, you will need to create a reference to a mixer. You can do so in one of the following ways, depending on your needs:

- Method 1 assumes that you have already created a Mixer member.

```
vMixer = member("MixerName")
```

- Method 2 creates a new Mixer member in an empty slot a castLib. There are several possible variants. See `new()` and _movie.newMember() for more details.

```
-- Lingo syntax
vMixer = new(#Mixer)
vMixer = _movie.newMember(#Mixer)
// JavaScript syntax
vMixer = _movie.newMember(symbol("Mixer"))
```

- Method 3 accesses the `mixer` property of an MP4 member or sprite. See "Mixing MP4 movie sound with other sounds" on page 419 for more details.

```
vSpriteMixer = sprite("MP4").mixer
vMemberMixer = sprite("MP4").member.mixer
vMemberMixer = member("MP4").mixer
```

Examples

The following code creates a new Mixer member in cast slot 23 of castLib 1, and names it "Mixer23". It then creates a `soundObject` using the sound member "PinkNoise", and plays the sound.

```
-- Lingo syntax
vMixer = _movie.newMember(#Mixer, member(23))
vMixer.name = "Mixer23"
vMixer.createSoundObject("Noise", member("PinkNoise"))
vMixer.play()
// JavaScript syntax
vMixer = _movie.newMember(symbol("Mixer"), member(23));
vMixer.name = "Mixer23";
vMixer.createSoundObject("Noise", member("PinkNoise"));
vMixer.play();
```

Note: *You will be able to use the Audio Mixer Inspector to visualize the new Mixer member and its soundObject, and to control playback.*

The Lingo behavior below creates a reference to the `mixer` property of the MP4 sprite that it is attached to. It then creates a new `soundObject` using the sound member "VoiceOver". As a result, the MP4 video plays the voiceover at the same time as the original soundtrack.

Note: *You will only be able to control the mixer of the MP4 sprite using Lingo or JavaScript syntax. The Audio Mixer Inspector cannot display the properties of an MP4 mixer.*

```
on beginSprite(me)
  vSprite = sprite(me.spriteNum)
  vMixer  = vSprite.mixer
  vName = "VoiceOver"
  vSoundObject = vMixer.getSoundObject(vName)
  if not vSoundObject then
    vSource = member(vName)
    vSoundObject = vMixer.createSoundObject(vName, vSource)
  end if
end beginSprite
```

Note: *In Director 11.5.8, the command below will cause Director to crash if it is executed as JavaScript code. The same line executed as Lingo code works correctly.*

```
sprite("MP4").mixer.createSoundObject(vName, vSource)
```

Mixing MP4 movie sound with other sounds

Director 11.5 introduced a new #mp4 member type. An #mp4 member allows you to play an external video or audio file in MP4, F4V or FLV format. Both #mp4 members have an mp4Member.mixer and the sprites that contain them have an mp4Sprite.mixer property.

Note: *In Director 11.5.8, there are issues with using JavaScript in connection with MP4 member and sprite mixers. The examples below are designed for Lingo implementation only.*

The member mixer and the sprite mixer are two separate objects. You can modify the `soundObjectList` of the sprite mixer and control each soundObject in the sprite mixer separately.

Note: *If the external video file does not contain an audio channel, then its mixer will be set to `0`. You will not be able to use it as an audio mixer. If you want to synchronize sounds with a silent video, you will need to use a Mixer cast member.*

You can access the different soundtracks in the video file through the sound objects list of the sprite mixer. You can add new sound objects to the sprite mixer.

Playing just the soundtrack of a video

To play the sound track of an MP4 member without playing the video, you can control its mixer directly. The following lines of code play back the sound track of member("MP4"), and play the sounds recorded in member("SoundEffects") at the same time.

```
vMixer = member("MP4").mixer
vSoundMember = member("SoundEffects")
vMixer.createSoundObject("SFX", vSoundMember)
vMixer.play()
```

Playing the video with additional soundtracks

The Lingo behavior is designed to be attached to an MP4 video sprite. It will play the video, the original soundtrack and the sounds recorded in member("SoundEffects") at the same time.

```
on beginSprite(me)
  vSprite = sprite(me.spriteNum)
  vMixer = vSprite .mixer
  vSoundMember = member("SoundEffects")
  vMixer.createSoundObject("SFX", vSoundMember)
end beginSprite
```

Chapter 6: Asynchronous programming

Today, features that are common to multitude of applications require asynchronous programming. Some examples are transitions, sliding menus, dragging and dropping, countdown timers, file downloads, and queries to a remote server.

The most simplest example of an asynchronous process is email. When you send an email, you may have to wait for some time before you get a reply. That means, you can spend time doing something else while waiting for the reply, and you have to check back occasionally to see if the reply has arrived yet. Even if the reply has already arrived, you are not obliged to deal with it immediately.

The following examples provide a clear understanding of the difference between synchronous and asynchoronous processes:

Example for a synchronous process

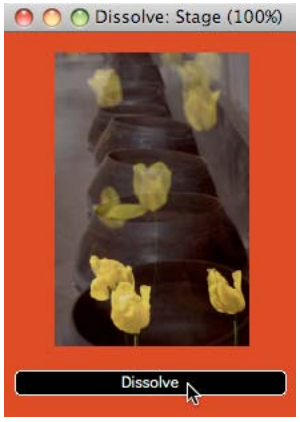
The code for a synchronous process works like a telephone conversation, where everything is treated in sequence without a break. Here is an example:

```
on mouseUp(me)
    vSprite = sprite(me.spriteNum)
    vSprite.member = member("Mouse Up")
    puppetSound("Button Click")
    go #next
end mouseUp
```

Everything inside the `on mouseUp()` handler happens in sequence, with no pauses to wait for a reply. The bitmap image for the “Mouse Up” member and the audio data from the “Button Click” sound member and the images to display at the next marker are retrieved from the computer's RAM space, or read from the Director file on the local hard disk in a fraction of a second. You expect the click to have an immediate effect.

Example for an asynchronous process

An asynchronous process, on the other hand, occurs little by little over a length of time, like an exchange of email messages. Below is a very simple behavior as an example. To see it in action, download and launch the movie Dissolve.dir.



The dissolve transition occurs asynchronously over several frames

```
-- DISSOLVE TRANSITION --
property pSprite
property pFrame
on beginSprite(me)
pSprite = sprite(me.spriteNum)
-- Grab the image of the previous frame
vStageImage = (the stage).image
-- Crop the image to the size of this sprite
vStageImage = vStageImage.crop(pSprite.rect)
-- Show image of previous frame in this sprite
pSprite.member.image = vStageImage
-- Start countdown
pFrame = 50
end beginSprite
on enterFrame(me)
if pFrame then
-- Reduce the opacity of this sprite
pFrame = pFrame - 1
pSprite.blend = 2 * pFrame
end if
end enterFrame
```

Basics of asynchronous programming

Asynchronous code relies on three principles:

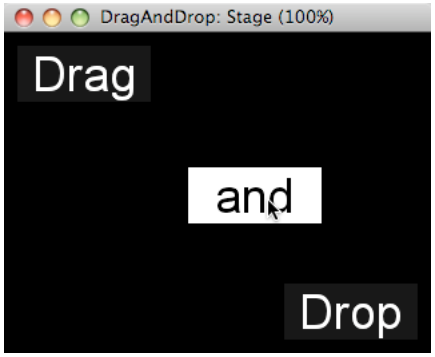
- The operation is divided up into multiple steps
- A regular event is used to process the operation
- Property variables are used to monitor the state of the operation as it progress

Based on these principles, your code consists of the following steps:

1. Starting the operation.
2. Processing the operation as it progresses.
3. Dealing with the final state when the operation is complete.

Sample program

To understand the logic involved, download and launch the movie DragAndDrop.dir. Click the word “and”, drag it to the word “Drop”, and then release it.



A drag and drop operation starts with a mouseDown and ends when the mouse is released.

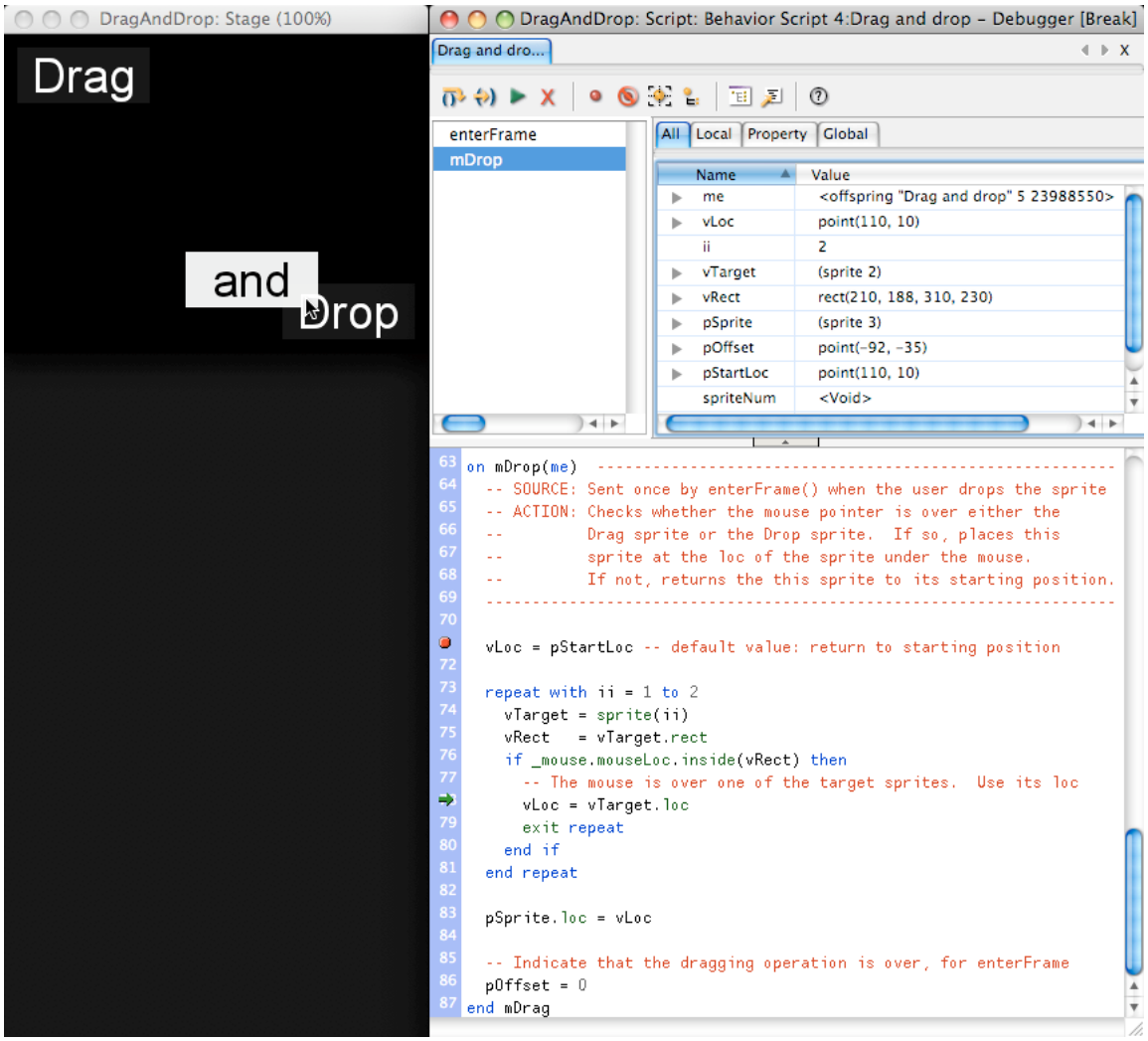
In the DragAndDrop.dir movie:

1. The operation starts when you click the And sprite. The `on mouseDown()` handler in the Drag and Drop behavior initializes the operation and sets a property which is used to track the state of the operation.
2. As you drag the mouse pointer, the position of the And sprite is updated. Once, per frame, the `on enterFrame()` handler moves the And sprite to the same position relative to the mouse.
3. When you release the mouse, the operation ends. As part of its action, the `on enterFrame()` handler checks if the mouseDown property is TRUE. If it is not, the mouse is released by the user, and the `mDrop()` handler is called to complete the operation.

Here is a simplified version of behavior that handles the Drag and Drop operation. It is attached to the sprite displaying the word “And”.

```
property pStartLoc
property pSprite
property pOffset
on mouseDown(me)
pSprite   = sprite(me.spriteNum)
pStartLoc = pSprite.loc
pOffset   = pStartLoc - the mouseLoc
end mouseDown
on enterFrame(me)
    if not pOffset then
    exit
    else if the mouseDown then
        me.mDrag()
            else
        me.mDrop()
    end if
end enterFrame
on mDrag(me)
pSprite.loc = the mouseloc + pOffset
end mDrag
on mDrop(me)
vLoc = pStartLoc
vTarget = sprite("Drop")
vRect = vTarget.rect
if _mouse.mouseLoc.inside(vRect) then
vLoc = vTarget.loc
end if
pSprite.loc = vLoc
pOffset = 0 -- stop calling mDrag() on each #enterFrame event
enf mDrag
```

If you are unsure how this behavior works, place a debug breakpoint at the beginning of each of the handlers, launch the movie, and start to drag the And sprite. The Debugger window will open, and you will be able to step through the code, line by line to see what it does. For more information on using the Debugger, see Debugging scripts in director.



Debugger

Tracking the progress of the operation

Notice how the `on enterFrame()` handler checks the state of the dragging operation. The operation can be in one of the following states:

Inactive pOffset is VOID or 0.

Drag pOffset will be a point and _mouse.mouseDown will be TRUE.

Drop The user has just released the mouse. The property pOffset will still be a point but _mouse.mouseDown will have become FALSE.

Events for asynchronous operations

This section introduces the events that you can use to monitor the progress of an asynchronous operation.

Director generates a number of different events on every frame. See "Using frame events wisely" on page 397. In addition to these events, you can create timeout objects (see below) and #timeMS events with 3D members. See "timeMS event callback" on page 359.

When performance is a priority, you should use an on enterFrame event handler, as demonstrated in the demo movies in"Sample program" on page 422. For projects where you are more concerned with creating simple code, or where you are synchronizing sprite movements with the mouse, use the following:

#stepFrame events sent to objects that are added to the actorList. See "actorList and #stepFrame events" on page 426 for more details.

Timeout objects See "Timeout objects" on page 428 for more information.

These two solutions allow you to generate regular events as they are needed, and to stop generating them when your code no longer requires them.

Note: *To update sprite display or positions, it is recommended that you use #enterFrame events. Using #stepFrame events for such operations may affect performance. See "Using frame events wisely" on page 397 for a comparison of frame event performance.*

actorList and #stepFrame events

This section describes the use of the actorList for operations where performance is not critical.

Director provides a special list called the actorList, which is designed to simplify the control of asynchronous operations. You can add behavior instances, child instances of parent scripts, and even scripts themselves to the actorList. Once, per frame, Director sends a #stepFrame event to all objects listed in the actorList. For every object included in the actorList, any code inside an on stepFrame event handler will be executed once per frame.

Director also sends a #stepFrame event when any of the following commands are executed:

- go()
- play() (3D)
- updateStage()

Any calls to go(), play(), or updateStage() made inside an on stepFrame() handler will be silently ignored. See on stepFrame for more details of the #stepFrame event and its timing.

Note: *To update sprite display or positions, it is recommended that you use #enterFrame events. Using #stepFrame events for such operations may affect performance. See "Using frame events wisely" on page 397 for a comparison of frame event performance.*

Adding an object to the actorList

To add an object to the actorList, use the following syntax:

```
_movie.actorList.append(object)
```

Removing an object from the actorList

To remove an object from the actorList, use the following syntax:

```
_movie.actorList.deleteOne(object)
```

See "Using an actorList event to remove an object" on page 428 for precautions to take when an object removes itself from the actorList inside its own on `stepFrame()` handler.

Clearing the actorList

To clear everything out of the actorList, you can use one of the following techniques:

- `_movie.actorList = []`
- `_movie.actorList.deleteAll()`
- `clearGlobals()`

  Note: *This command also deletes all global variables.*

Order of execution

Director sends a #stepFrame event to each of the objects on the actorList after it generates the #exitFrame event, and before it generates a #prepareFrame event. No updates to the Stage display will occur until after all `on prepareFrame()` handlers have completed. You can also send a custom event to the actorList at any time, using the following syntax:

```
-- Lingo syntax
call(the actorList, #eventSymbol{, parameters})
// JavaScript syntax
_movie.call(symbol("eventSymbol"),_movie.actorList{,parameters})
```

The objects on the actorList will receive and process the event in the order that they appear on the actorList.

Example

To test the use of the `call()` method on the actorList, download and launch the actorListTest.dir.

The actorListTest.dir movie places three script objects on the actorList.

- An instance of the Set Color and Text parent script.
- The instance of the Rotate Text behavior that is attached to the rotating text sprite.
- A pointer to the Just Delete Me script.

The Rotate Text script uses an `on stepFrame()` handler to rotate the sprite that it is attached to. The two other script objects do not have an `on stepFrame()` handler. Their role is to illustrate other aspects of the way the actorList works.

Sending a custom message to the actorList

If you click on the top button in the movie actorListTest.dir, a #customEvent message is sent to the actorList with two parameters. The second parameter is an empty property list. The Set Color and Text instance receives the #customEvent call first, and uses it to modify the contents of the property list. The Rotate Text instance then receives the #customEvent call along with the modified property list. It uses the contents of the property list to change the display in the rotating text sprite. Finally, the Just Delete Me script then receives the message and plays a sound.

This illustrates:

- The different type of script objects which can usefully be added to the actorList.
- The order in which objects receive events depends on the order in which they appear in the actorList.
- The use of a list to pass data between objects on the actorList.

Using an actorList event to remove an object

If you click on the lower button, the event #deleteMe is sent to the actorList. The Set Color and Text instance receives the #deleteMe call first, and uses it to remove itself from the actorList. There are now only two objects remaining on the actorList, but Director considers that it has already sent the #deleteMe call to the first item on the list. For this reason, the event is never sent to the Rotate Text behavior. Instead, it is sent immediately to the second item on the list; the Just Delete Me script deletes itself from the actorList, but the text sprite continues to rotate.

The same effect will occur if you use a #stepFrame event to remove an object from the actorList. If there is only one object on the actorList, or if it is not a problem if the next object misses a #stepFrame event, then you do not need to do anything special. If you want to be sure that all objects on the actorList receive a #stepFrame event in all circumstances, then you can use the handler below:

```
on RemoveFromActorList(aInstance)
    vIndex = (the actorList).getPos(aInstance)
    if not vIndex then
        exit
    end if
    (the actorList).deleteAt(vIndex)
    if vIndex <= (the actorList).count then
        -- The instance now at <vIndex> will not receive the
        -- #stepFrame event automatically. Send it manually:
    call(#stepFrame, [the actorList[vIndex]])
    end if
end RemoveFromActorList
```

The actorListTest.dir movie contains a movie script named ActorList Handlers that includes a similar handler. To use it, call the `RemoveFromActorList()` method from an `on stepFrame()` handler.

```
on stepFrame(me)
--  Do whatever else needs to be done
RemoveFromActorList(me)
end stepFrame
```

Note: *When you are authoring a movie, Director will stop sending #stepFrame events when the movie is stopped. When the movie starts again, any objects that are still on the actorList will start receiving #stepFrame events again. You may want to clear all objects from the actorList before you restart your movie. One way to do this is to place a* `clearGlobals()` *call in the movie's* `prepareMovie()` *handler.*

Timeout objects

A timeout object is a script object that acts like a timer and sends a message at regular intervals. This is useful for scenarios that require specific things to happen at regular time intervals, or after a particular amount of time has elapsed. A timeout object can send messages that can in turn call event handlers inside child objects or in movie scripts.

Callbacks from timeout objects do not occur at precise intervals. It is not possible, for example, to create a metronome that will give a steady beat. Callbacks from timeout objects are triggered in the "idle" period that occurs after the Stage image has been updated, and before the next stage update is due. If a callback falls due during the time when the screen is updating, it will be delayed until the update is complete. This delay is cumulative. Director will make no attempt to catch up by triggering the next callback sooner. The period property of timeout objects determines the shortest period between two callback events, not the average period.

Timeout callbacks are only issued while the movie is running. While authoring, timeout callbacks will stop as soon as you stop the movie. If you have any persistent timeout objects, they will start generating callbacks again when you restart the movie.

Creating a timeout object

You create a timeout object by using the `new()` keyword. You must specify a name for the object, a handler to be called, and the frequency with which you want the handler to be called. After a timeout object is created, Director keeps a list of currently active timeout objects, called timeoutList.

Syntax

The Lingo syntax for creating timeout objects changed with Director MX 2004 (D10), in order to make it consistent with the way other objects are created. Movies authored in Director MX (D9) and earlier have a scriptExecutionStyle property set to a value of 9, which allows you to use the syntax found in Director MX and earlier.

```
-- Lingo syntax when scriptExecutionStyle is set to 10
variableName = timeout().new(timeoutName, timeoutPeriod, timeoutHandler {, targetObject})
-- Lingo syntax when scriptExecutionStyle is set to 9 (for older movies only)
variableName = timeout(timeoutName).new(timeoutPeriod, timeoutHandler {, targetObject})
// JavaScript syntax (all versions since Director 7)
variableName = new timeout(timeoutName, timeoutPeriod, timeoutHandler, targetObject)
```

This statement uses the following elements:

variableName is the variable you are placing the timeout object into.

timeout indicates which type of Lingo object you are creating.

timeoutName is the name you give to the timeout object. This name appears in the timeoutList. It is the name property of the object. If you create a new timeout object with the name of an existing timeout object in the same movie, the existing timeout object will be replaced. Timeout namespace is restricted to the current window. Two windows displaying the same movie will each have their own timeoutList.

new() creates a new object.

timeoutPeriod is an non-negative integer indicates the minimum number of milliseconds after which the timeout object should call the handler you specify. This is the period property of the object. For example, a value of 2000 calls the specified handler every 2 seconds. Setting the timeoutPeriod to 0 stops the timeout object from sending any callbacks.

timeoutHandler is the symbol name of the handler you want the object to call. This is the timeoutHandler property of the object. You can use either a symbol or a string. For example, a handler called on accelerate() could be specified as #accelerate or “accelerate”.

targetObject indicates which child object’s handler should be called. This is the target property of the object. It allows specificity when many child objects contain the same handlers. If you omit this parameter, Director looks for the specified handler in the movie script.

Examples

The following statement creates a timeout object named timer1 that calls an `on accelerate()` handler in the child object car1 every 2 seconds:

```
-- Lingo syntax (with a scriptExecutionStyle of 9)
vTimer = timeout("timer1").new(2000, #accelerate, car1)
-- Lingo syntax (with a scriptExecutionStyle of 10)vTimer = timeout().new("timer1", 2000,
#accelerate, car1)
// JavaScript syntaxvTimer = new timeout("timer1", 2000, "accelerate", car1)
```

Note: *The timeout properties time and persistent are set implicitly when a new timeout object is created. The value if time is set to the current value of _system.milliseconds, and the value of persistent is set by default to 0. If you create a new timeout object with the same name as an existing one, the current value of persistent will be maintained.*

Deleting a timeout object

Use the forget() (Timeout) method to delete a timeout object. The following handler deletes the timeout object that activated it.

```
on doItOnce(me, aTimeout)
    aTimeout.forget()
    -- Add code here that will be executed only once
end doItOnce
// JavaScript syntax
function doItOnce(me, aTimeout)
{
aTimeout.forget();
// Add code here that will be executed only once
}
```

The following handler will delete all timeout objects that are currently active:

```
-- Lingo syntax
on killAllTimeouts()
    ii = the timeoutList.count
    repeat while ii
        timeout(ii).forget()
        ii = ii - 1
    end repeat
end killAllTimeouts
// JavaScript syntax
function killAllTimeouts(){
ii = _movie.timeoutList.count
for(ii;ii>0;ii--){
timeout(ii).forget()
}
}
```

Note: *_movie.call("forget", _movie.timeoutList) deletes every second timeout object. When the first timeout object is deleted, the second timeout object will become the first item in the timeoutList, but the #forget event will now be sent to the second object in the list, so the object that is now first in the list will never receive it.*

Timeout object properties

Timeout objects have the following properties which you can both get and set:

- "name" on page 431
- "period" on page 431

name

This is the name of the timeout object as defined when the object is created. You can change the name of a timeout object at any time. If, in any given movie, you create a timeout object with the same name as another timeout object, the first object will be replaced by the new one.

```
-- Lingo syntax
timeout("John").name = "George"
timeout().new("Peter", 40, #keepTheBeat)
put _movie.timeoutList.getLast().name
-- "Peter"
-- JavaScript syntax
timeout("John").name = "George";
new timeout("Peter", 40, "keepTheBeat");
trace(_movie.timeoutList.getLast().name);
// Peter
```

period

The minimum number of milliseconds between timeout events sent by the timeout object to its timeoutHandler. Setting the period to 0 will suspend all further callbacks to the timeoutHandler, but it will not suspend system messages relayed by the timeout. See "Relaying system events with timeout objects" on page 434.

This timeout handler decreases the timeout's period by one second each time it is invoked, until a minimum period of 2 seconds (2000 milliseconds) is reached.

```
-- Lingo syntax
on handleTimeout(me, aTimeout)
    aTimeout.period = max(2000, aTimeout.period - 1000)
end handleTimeout
-- JavaScript syntax
function handleTimeout(aPlaceholder, aTimeout) {
aTimeout.period = list(2000, aTimeout.period - 1000).max();
}
```

persistent

Determines whether the given timeout object is removed from the timeoutList when the current movie stops playing. If TRUE, the timeout object remains active. If FALSE, the timeout object is deleted when the movie stops playing. The default value is FALSE.

Setting this property to TRUE allows a timeout object to continue generating timeout events in other movies. This is useful when one movie branches to another with the 'go to movie' command.

The following statement creates a timeout object and makes it persist across movies:

```
-- Lingo syntax (Director 10 and later)
global gTO
gTO = timeout().new("perpetualMotion", 50000, "doItAgain")
gTO.persistent = TRUE
// JavaScript syntax
_global.gTO = new timeout("perpetualMotion", 50000, "doItAgain")
_global.gTO.persistent = true;
```

target

Indicates the child object that the given timeout object will send its timeout events to. Timeout objects whose target property is not a script or script instance will send their events to a handler in a movie script. The target is sent as the first parameter of all callbacks, even when the callback is in a movie script. See "Associating custom properties with timeout objects" on page 435 for ideas on how to utilize this feature.

This property is useful for debugging behaviors and parent scripts that use timeout objects. This statement displays the name of the child object that will receive timeout events from the timeout object timerOne in the Message window:

```
-- Lingo
put timeout("timerOne").target
// Javascript
trace(timeout("timerOne").target)
```

Time

To determine when the next message will be sent from a particular timeout object, check its time property. The value returned is the point in time, in milliseconds, when the next timeout message will be sent. You can change the time when the timeout object will next create a callback by setting its time property. The following example sets the time for the next callback to 10 seconds from now.

```
-- Lingo syntax
timeout("timer1").time = the milliseconds + 10000
// JavaScript syntax
timeout("timer1").time = _system.milliseconds + 10000;
```

You cannot specify the time property when you create a new timeout object, but you can set it at any time. The following lines create a timeout object which will start triggering callbacks after a delay of 5 seconds, and will subsequently trigger a callback every second.

- vTimeout = timeout().new("Take5", 1000, #WaitASecond)
- vTimeout.time = the milliseconds + 5000 Setting the time of a timeout object whose period is 0 will not trigger any callbacks.

timeoutHandler

Represents the name of the handler that will receive timeout messages from the given timeout object. Its value can be either a symbol, such as #timeExpiredHandler, or a string such as "timeExpiredHandler". Use the timeoutHandler as a handler within the timeout object's target object, or in a movie script if the timeout object has no target specified. If no handler of that name is found, callbacks will fail silently.

Example

This statement sets the timeoutHandler of the timeout object Quiz Timer to #countDown.

```
-- Lingo
timeout("Quiz Timer").timeoutHandler = #countDown
// Javascript
timeout("Quiz Timer").timeoutHandler = "countDown"
```

Using timeoutList

When you begin creating timeout objects, you can use timeoutList to check the number of timeout objects that are active at a particular time. The following statement sets the variable ii to the number of objects in the timeoutList by using the count property:

```
ii = _movie.timeoutList.count
```

You can also refer to an individual timeout object by its index number in the list.

The following statement deletes the second timeout object in timeoutList by using the forget() method:

```
timeout(2).forget()
```

Order of service

If more than one timeout object is ready to send its callback message, the order in which the messages are sent will be the reverse order of their appearance on the timeoutList. You can remove a timeout object from the timeoutList, without destroying the object. If it is no longer on the timeoutList, it will not trigger any callbacks. The movie script handlers below allow you to test both these ideas.

```
-- Lingo syntax
global gTimeouts
on testTiming()
    gTimeouts = []
    repeat with vName in ["Paul", "George", "John", "Peter"]
        vTimeout = timeout().new(vName, 1, "WhoGoesFirst")
        gTimeouts.append(vTimeout)
    end repeat
end
on WhoGoesFirst(placeholder, aTimeout)
    put aTimeout.name
    _movie.timeoutList.deleteOne(aTimeout)
end
-- In the Message window
testTiming()
-- "Peter"
-- "John"
-- "George"
-- "Paul"
put the timeoutList, gTimeouts
-- [] [timeOut("Paul"), timeOut("George"), timeOut("John"), timeOut("Peter")]
```

If you return a timeout to the timeoutList, it will start generating callback events again.

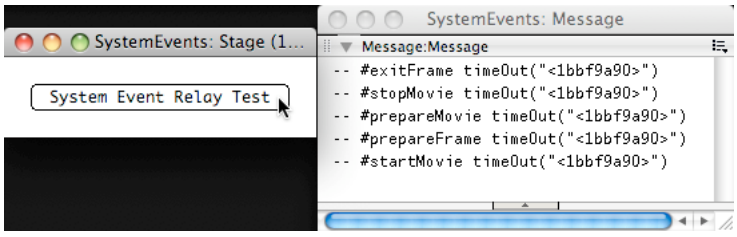
```
(the timeoutList).append(gTimeouts[2])
-- "George"
// JavaScript syntax
function testTiming(){
_global.gTimeouts = list("Paul", "George", "John", "Peter");
for(ii=1;ii<5;ii++){
vName = gTimeouts[ii];
vTimeout = new timeout(vName, 1, "WhoGoesFirst");
gTimeouts[ii] = vTimeout;
}
}
function WhoGoesFirst(placeholder, aTimeout) { trace(aTimeout.name);
_movie.timeoutList.deleteOne(aTimeout);
}
-- In the Message window
testTiming(); // George // Paul // Peter // John trace(_movie.timeoutList, gTimeouts) // <[]>
<[timeOut("Paul"), timeOut("George"), timeOut("John"), timeOut("Peter")]>
--If you return a timeout to the timeoutList, it will start generating callback events again.
_movie.timeoutList.append(gTimeouts[2])
0
// George
```

Relaying system events with timeout objects

When you create timeout objects that target specific child objects, you enable those child objects to receive system events. Timeout objects relay these events to their target child objects.

The system events that can be received by child objects include #prepareMovie, #startMovie, #stopMovie, #prepareFrame, and #exitFrame. You can include handlers for these events in child objects, and make the child objects respond to them for whatever purposes you see fit.

System events received by child objects are also received by movie scripts, frame scripts, and other scripts designed to respond to them. To test this behavior, download and launch the SystemEvents.dir. Click on the System Event Relay Test button and observe the output in the Message window.



The SystemEvents.dir movie creates a timeout object with a period of 0 to generate system events

When you click the button, a child instance of the script System Event Relay Test is created. An edited version of the script is shown below.

```
property pTimeout
on new(me)
vName = "<" & the last word of string(me)
pTimeout = timeout().new(vName, 0, #invalidHandler, me)
pTimeout.persistent = TRUE -- to survive "go movie"
return me
end new
on forget(me)
pTimeout.forget()
pTimeout = VOID
end forget
on exitFrame(me, aTimeout)
put #exitFrame, aTimeout
go movie the movieName
end
on startMovie(me, aTimeout)
put #startMovie, aTimeout
me.forget()
end
```

This technique is useful for detecting events such as #stopMovie in a child object that is stored as a global or as an object on the actorList.

If you use a behavior as the target for a timeout object, the behavior will receive #prepareFrame and #exitFrame events twice: once from the sprite it is attached to, and once from the timeout object.

You can determine the source of the call by checking for a second parameter. The `on exitFrame()` Lingo handler below ensures that it is not activated when the call comes from a timeout object.

```
on exitFrame(me, aTimeout)
    if not voidP(aTimeout) then exit
end if
-- If we get here the event came from the spriteend
```

Associating custom properties with timeout objects

When the target of a timeout object is not a script or a script instance, the timeout will direct its callback to a handler in a movie script. In all cases, the callback handler will receive two parameters:

- The target property of the timeout
- A reference to the timeout object itself

You can exploit this feature as a way to associate a custom property to a timeout object.

The following sample code includes two movie script handlers: The `createCountdown()` handler creates a timeout object. The `Countdown() handler` is the timeoutHandler for the timeout object.

```
on createCountdown()
    vTimeout = timeout().new("Countdown", 1000, #Countdown, 10)
    vTimeout.time = the milliseconds
end createCountdown
on Countdown(aRemaining, aTimeout)
    put aRemaining
        if not aRemaining then
        aTimeout.forget()
    end if
    aTimeout.target = aRemaining - 1
end Countdown
```

When you call `createCountdown()`, it sets the target of the new timeout object to the integer 10. The `Countdown()` handler receives this value as the first parameter and then modifies the target value of the timeout. The result is a countdown from 10 to 0, printed out in the Message window.

The target value can be any Lingo type, including a list or a property list. In general, keep the following in mind:

- When using a reference to a script instance as a target, the target handler in that particular script instance is called. This technique does not allow the use of custom properties.
- When using a reference to anything other than a script instance (such as a property list) as a target, the target handler in a movie script is called. This technique allows the use of custom properties.

Downloading data from a remote server

Sometimes, in your projects, you may need to retrieve a file that is located on a remote server. Your end users will not be able to use the contents of that file until a copy of the file is downloaded onto their computers.

This topic describes the concepts in retrieving data from a remote server and explains how to retrieve data in an asynchronous operation.

Downloading or preloading

If your movie is playing back as a Shockwave application inside a browser, you will not have any access to the end-user's local disks. The local copy of the file will be written into the browser's cache folder, but your Shockwave movie will not be able to know what file path is used. This process is known as preloading. For preloading, use the preloadNetThing() method.

If your movie is playing in Director's authoring environment or in a projector, you can have direct access to the end-user's local disks (or at least to those folders for which the current user has write privileges). In this case, you can specify a path to a file in a local folder to which the remote file will be downloaded. You can use the downloadNetThing () method in this case.

You can also use preloadNetThing(), if you do not wish to specify a local file path. In this case, the local copy of the file will be stored in Director's cache. If your application makes a preloadNetThing() request for the file a second time, the cached version will be used by default.

getNetText and postNetText

You may want to send data to a remote server, and to retrieve confirmation that it was successfully received. The confirmation may be quite a lengthy text. For example, you may send a username and a password to a remote server, and receive in return a set of waiting messages that have been stored in a database.

For this purpose, you can use either the GET HTTP method, through the getNetText() command, or the POST HTTP method, through the postNetText() command. In both cases, Director will retrieve the result, but will wait for you call netTextResult() before providing the result.

Retrieving data as an asynchronous operation

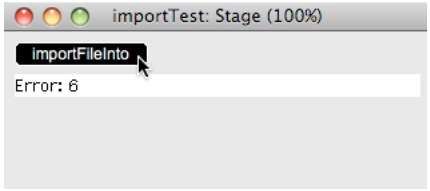
For all four of these remote processes, you need to start the operation and then check on a regular basis whether all the return data has been successfully received. Finally, you will need to treat the result.

Here is a very simple example written as a Lingo behavior to be placed on a button sprite.

```
property pNetID  -- integer id of the net operation
property pURL    -- url of the remote file
property pMember -- text member in which to show result
on mouseUp(me)
    me.mStartOperation()
end mouseUp
on enterFrame(me)
    if pNetID then
        me.mCheckProgress() -- calls mCompleteOperation() when done
    end if
end enterFrame
on mStartOperation(me)
    pMember = member(2) -- <HARD-CODED: name may change>
    pURL    = the moviePath&"importTest.txt"
    pNetID  = preloadNetThing(pURL)
end mStartOperation
on mCheckProgress(me)
    if netDone(pNetID) then
        me.mCompleteOperation()
    end if
end mCheckProgress
on mCompleteOperation()
    vNetError = netError(pNetID)
    pNetID = 0
    case vNetError of
        "OK", "":
        pMember.importFileInto(pURL)
        otherwise: -- an error occurred
        pMember.text = "Error: "&vNetError
    end case
end mCompleteOperation
```

To test this, download and launch the importTest.dir. When you click the ImportFileInto button, the movie reports an error. This is the expected behavior.

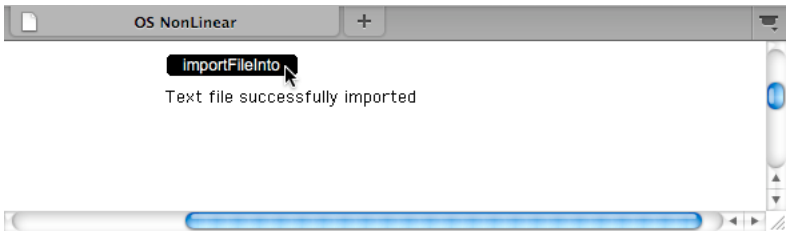
The error message indicates that the file required for preloading does not exist.



The error message indicates that the file required for preloading does not exist

To check that the preload process works correctly, create a file named importTest.txt and place it alongside the importTest.dir movie. You can include whatever text you want in the file. If the file exists, then the movie will be able to preload it.

To test this in a browser, you will need to upload both the movie and the text file to the same folder on your server, and then use your browser to connect to the movie.



Testing asynchronous code in a browser

Tracking download progress

Each time you call one of the methods preloadNetThing(), downLoadNetThing(), getNetText(), and postNetText(), it returns a unique positive integer. The return value increases by 1 on each call. You can use this net ID integer to identify that particular net operation.

You can use the net ID with getStreamStatus() or with tellStreamStatus() and streamStatus() to obtain information about how a net operation is progressing.

To see an example of using getStreamStatus() to display a progress bar, download and launch the movie Progress.dir.

The Progress.dir movie shows a textured progress bar as it downloads a 2MB image file from a server

The following handler checks the progress of a net operation identified by aNetID. The handler also sets the width sprite("Progress") to a value between 0 and 100 to show how much of the file has been downloaded.

```
on DisplayProgress(aNetID)
vStatus  = getStreamStatus(aNetID)
vBytesTotal = vStatus.bytesTotal
if vBytesTotal then
vProgress = vStatus.bytesSoFar / float(vBytesTotal)
else
-- The download might not have started, or it may be a
-- streaming download with no fixed length
vProgress = 0.0
end if
vWidth = 100 * vProgress
vSprite = sprite("Progress")
vSprite.right = vSprite.left + vWidth
end DisplayProgress
```

Interacting with PHP scripts

To send data to a PHP script, you can use either the GET HTTP method, through the getNetText() command, or the POST HTTP method, through the postNetText() command. In both cases, Director will retrieve the result, but will wait for you call netTextResult() before providing the result.

Here is a very simple PHP script. It returns the input if the input is a number between 1 and 100, or an error string if the input is not an integer or outside the given range.

```
<?php
$integer = $_REQUEST['input'];
if (!is_numeric($integer)) {
die("Error: number expected");
}
if ($integer > 100) {
die("Error: number greater than 100");
}
if ($integer < 1) {
die("Error: number less than 1");
}
echo $integer;
?>
```

Note that the script uses the $_REQUEST superglobal. This means that you can test this script using either the GET or the POST methods. Create a text file, paste this script into it, then name the script test.php and upload it onto your server.

Suppose the file on your server is accessible as http://my.example.com/director/test.php. You can test how it works by entering URIs such as:

```
http://my.example.com/director/test.php?input=0
http://my.example.com/director/test.php?input=1
http://my.example.com/director/test.php?input=100
http://my.example.com/director/test.php?input=101
http://my.example.com/director/test.php?input=George
```

Note: *If you can test your PHP scripts independently of your Director implementation, you will find it easier to catch errors in the PHP scripts themselves. If you can only test them from a Director movie, it may be difficult to distinguish between Director errors and PHP errors.*

Here is a very simple behavior. You could create a movie, and attach this to a button sprite.

```
property pNetID  -- integer id of the net operation
on mouseUp(me)
me.mStartOperation()
end mouseUp
on enterFrame(me)
if pNetID then
me.mCheckProgress() -- calls mCompleteOperation() when done
end if
end enterFrame
on mStartOperation(me)
vFolder = "http://my.example.com/director/" -- use valid URL
vURI    = vFolder&"test.php?input=50"
pNetID  = getNetText(vURI)
end mStartOperation
on mCheckProgress(me)
if netDone(pNetID) then
me.mCompleteOperation()
end if
end mCheckProgress
on mCompleteOperation()
vNetError = netError(pNetID)
case vNetError of
"OK", "":
vResult = netTextResult(pNetID)
otherwise: -- an error occurred
vResult = "Error: "&vNetError
end case
pNetID = 0
alert vResult
end mCompleteOperation
```

If you want to experiment with postNetText(), you can use the following handler instead of the mStartOperation() handler above.

```
on mStartOperation(me)
vFolder = "http://my.example.com/director/" -- use valid URL
vURI    = vFolder&"test.php"
vData   = ["input": 50]
pNetID  = postNetText(vURI, vData)
end mStartOperation
```

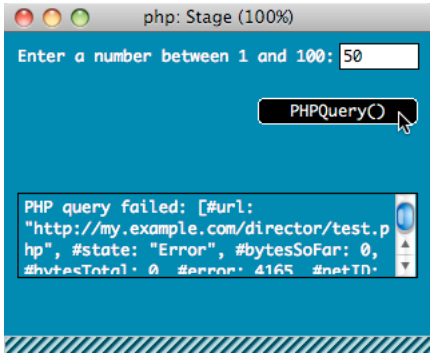
PHP scripts are case sensitive. A PHP variable called 'Input' is not the same as a variable called 'input'. This makes it risky to use symbols as the properties of the property list that acts as the second parameter for the postNetText() call.

Try executing the following lines in the Message window to understand why.

```
 trace(#mixed) -- creates #mixed as a symbol with a lower-case "m"
 -- #mixed
trace(string(#Mixed))
 -- "mixed"
trace(string(#mixer)) -- built-in symbol with an upper-case "M"
 -- "Mixer"
```

To avoid issues due to the case-sensitivity of PHP scripts, always use strings for the property names in the data property list sent with a postNetText() call, as shown in the example above.

To test a postNetText() operation, complete with feedback to indicate that something is happening, download and launch the php.dir.



Testing a postNetText() with an invalid file path to the remote server

If you click on the PHPQuery() button, you see that the progress bar starts to move, and then an error message appears. This indicates that the domain name and folder path used in the PHPQuery() handler of the PHP Query movie script are invalid. You then need to edit the following hard-coded lines, using the values appropriate for your own server:

```
 vFolder   = "http://my.example.com/director/"
vPHP      = "test.php"
```

You may want to use a call to a PHP script to return an image. Displaying an image retrieved from a PHP script in a Shockwave movie requires special care. The importFileInto method will not work. If you set the fileName (Member) of a Bitmap member to the URI of the PHP script, then the operation may fail. It will only succeed if you set the fileName of a member that is already linked to an external file.

Use the following steps to ensure that you can display the image:

1. Create a tiny placeholder image file and place it alongside your Director movie
2. Import this placeholder image as a Bitmap, using the Link To Exernal File option
3. When your PHP operation completes, set the fileName of this linked Bitmap member to the URI of the PHP script.

Querying a MySQL database

Many projects require a Director movie to communicate with an online database. For example, you may be creating a game with a high-score table, and the scores may be recorded in a database on your web server. One common scenario is the use of a PHP script that queries a MySQL database on the same server. When the MySQL data returns a response to the PHP script, the PHP script can echo information back to the Director movie.

The following is an extract of a PHP script which makes a query to a MySQL database.

```
$sql = " SELECT DISTINCT id, screenName, userName FROM User
WHERE User.username = TEST NAME'
AND User.password = 'TESTING'";
$result = mysql_query($sql);
```

You may want to return the result as an XML string. To do so, use the XML Parser Xtra to convert it to a format that Director can use.

Alternatively, you can prepare the output in a format that Director can easily convert to a property list. This has the advantage of using less bandwidth than the same data formatted is an XML string. The disadvantage is that the PHP developer will need to use a proprietary format for the output.

```
$row = mysql_fetch_assoc($result);
$id = $row['id'];
$screenName = $row['screenName'];
$userName = $row['userName'];
$output = '[#id:'.$id.',#screenName:"'.$screenName.'",#userName:"'.$userName.'"]';
echo $output;
```

Chapter 7: Unicode support in Director

Unicode is a computer industry text-encoding standard that lets computers consistently represent and manipulate text expressed in any of the world's writing systems.

Adobe Director supports many features of Unicode and provides support for multilingual data in movies. Using Director, you can create and view movies containing text in languages other than English.

Director uses UTF-8 encoding for Unicode support. External data formats that are supported are DBCS, UTF-8, UTF 16, and UTF 32.

You can use Director's Unicode support to perform tasks such as creating cast members, creating external casts, adding comments for a cast member, naming files, and linking casts in any language. Unicode support in scripting helps you to create variables, function names, and strings in multiple languages.

Upgrade the custom Xtras using the XDK provided with Director to make them Unicode-compliant. For more details, refer to the XDK documentation.

Limitations of Unicode support in Director

- Languages written right-to-left are not supported.
- To view Unicode text correctly in the Shockwave® Player, users must install the required fonts on their computer.
- 3D model names in Unicode are not supported.
- Unicode names for HTTP paths are not supported.
- Because browsers support only ASCII characters in the URL, you cannot play a Director movie in the following scenarios:
  - Playing a movie published to the internet using a Unicode name.
  - Playing a movie with a Unicode name downloaded from the browser.
  - Playing a Shockwave® movie that has an external cast or is an externally linked cast member with a Unicode name accessed from a URL.
- You cannot name a scripting Xtra as a Unicode string using the 'kMoaMmDictType_MessageTable' registry key. Also, you cannot expose lingo functions named in Unicode by using the scripting Xtras.
- The .x32 file on Windows and the .xtra file on Mac that is copied into the configuration Xtras folder cannot have a Unicode name.

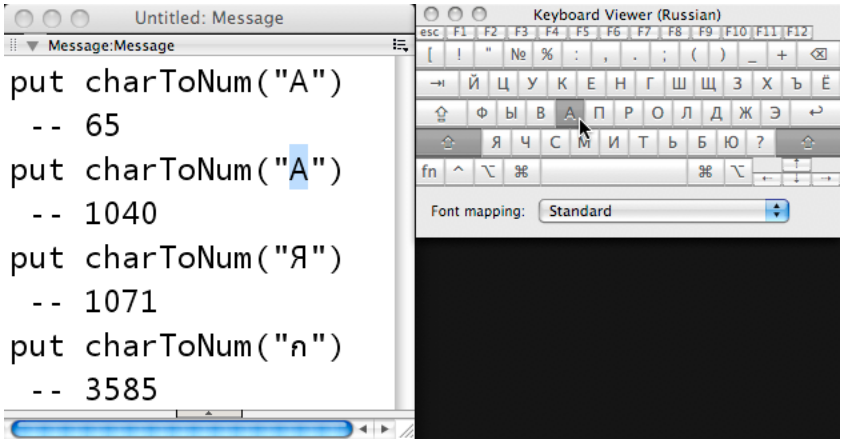
Encoding and fonts

To view Director movies created in any of the Unicode-compliant languages, you must have the required fonts and language packs installed on your computer.

Displaying text on your computer screen requires two different technologies:

- An encoding system: to represent the characters of a language in binary terms that a computer can understand
- A font: to define the shape of those characters

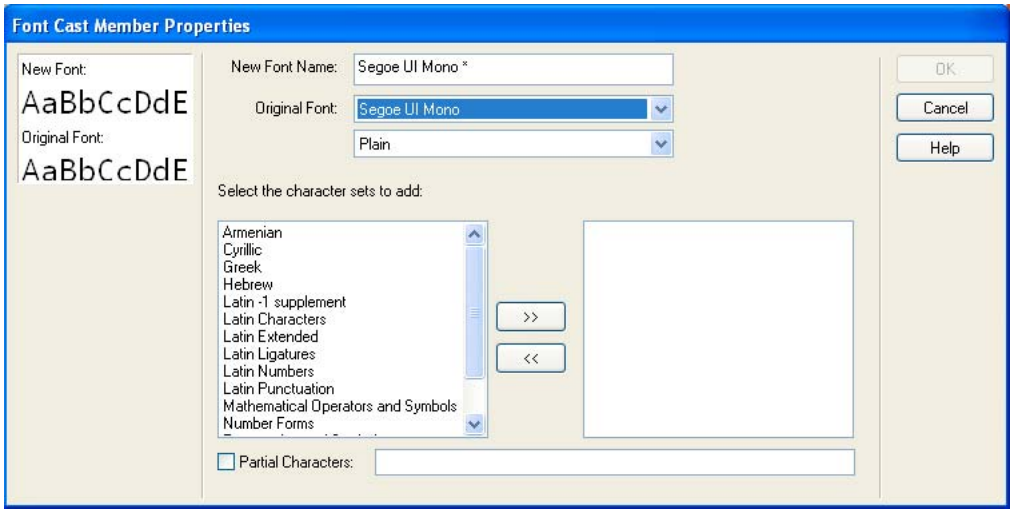
The Unicode standard provides a way to encode over 100,000 different characters from 93 different writing systems. Each character is represented by a unique number. You can use the charToNum() function to discover what that number is for each character that you type. You can find tables of UTF mappings here.



UTF-8 uses different numbers for characters from different writing systems even if they look the same

UTF-8 encoding numbers are not designed to be readable by humans. Humans prefer to read shapes. A font is essentially a look-up table that converts a number into a human-readable shape.

There are few fonts, if any, that contain data for more than 100,000 different characters. Most fonts are designed for a small number of specific writing systems. To display characters from a specific writing system, your end users will need to have fonts that can display that writing system installed on their computers.



The Segoe UI Mono font (used in the Message window above) copes with 15 character sets from 7 writing systems

For a multilingual project that is targeted at users in a variety of locations, you will need to ensure that you include the appropriate fonts with your application. If the variety of languages that you need to display is diverse, it may be difficult to find a single font that covers all your requirements.

If the end-users need to input text in different languages, you can use the `charToNum()` value of the characters they type to indicate which writing system they are using. You can use this to set the font appropriately.

Here is a simple behavior that you can attach to a text sprite. It automatically selects between fonts for Roman, Cyrillic, or Thai characters, depending on the text that the user begins to type.

```
property pFontSelected
on keyDown(me)
  if not pFontSelected then
    vCharNum = charToNum(_key.key)
    me.mSelectFontFor(vCharNum)
  end if
  pass
end keyDown
on mSelectFontFor(me, aCharNum)
  if aCharNum < 65 then
    exit
  else if aCharNum < 128 then
    vFont = "Roman"
  else if aCharNum < 1024 then
    exit
  else if aCharNum < 1280 then
    vFont = "Cyrillic"
  else if aCharNum < 3584 then
    exit
  else if aCharNum < 3712 then
    vFont = "Thai"
  else
    exit
  end if
  vMember = sprite(me.spriteNum).member
  vMember.font = vFont

  pFontSelected = TRUE
end mSelectFontFor
```

For your own projects you will need to customize this behavior by indicating the appropriate code number ranges and the appropriate font names.

Writing systems

If you need to work with a language that you do not understand well, then you may need to learn more about its writing system. This may be very different from the writing system of your own language. This article provides a few ideas on how language systems differ, and how well Director is able to cope with these differences.

Direction

Many languages, like English and Russian, are written from left to right. Other languages, like Arabic and Hebrew, are written from right to left. In both these languages, numbers are written from left to right. This makes them bi-directional.

Director only supports writing systems that are written from left to right.

Letter forms

In most Western languages, there is a distinction between uppercase letters at the beginning of sentences and words, and lowercase letters. The writer distinguishes between these letters by pressing the Shift key. In other writing systems, such as Arabic, there may be different forms for the same character depending on whether it appears at the start of a word, in the middle of a word, or at the end of the word. The shape of a character that you have already typed may change automatically as you insert other characters before or after it.

Director relies on the operating system to provide information on which letter forms to use.

Collation

Different languages, even if they use the same writing system, may use different methods for ordering words in a dictionary. For example, in Spanish, the alphabet consists of 27 letters, with the inclusion of the letter ñ between n and p: …m, n, ñ, o, p, … In Thai, which is written from left to right, certain vowels are written to the left of a consonant that is pronounced first. In the dictionary, the order of the consonants takes priority over the order of the vowels. In writing systems like Chinese, where there is no "alphabetical" order, the collation system is very complex.

The list.sort() method in Director uses character encoding (see numToChar()) to determine order. If you need to use collation on languages other than English, you will probably need to create a custom system, or rely on a third-party database xtra extension to provide you with the required functionality.

Input Method Editors

When you type text messages on a mobile phone, using the number keys, you are using an Input Method Editor (IME). Languages such as Chinese, Japanese, and Korean (CJK languages), where there are a large number of characters, use an IME to map multiple key strokes to a single final character.

Director supports the IME for Chinese, Japanese, and Korean. However, for other writing systems like the Ethiopian Amharic abugida, built-in IME is not available.

Language Packs

Each writing system has specific requirements for display and editing. These are handled at the level of the operating system. You need to be sure that the end-user has the appropriate language packs installed and activated. If you are creating a language-learning application, you may need to provide specific instructions to end-users on how to set up their system to display and input a foreign language correctly.

Supported languages

With the exception of Right-to-Left languages, the Adobe Director development team is committed to supporting all languages whose character set features in the UTF-8 standard. The languages listed below have all been actively tested.

- Brazilian Portuguese
- Chinese – Simplified
- Chinese – Traditional
- Dutch
- English
- French

- German
- Italian
- Japanese
- Korean
- Russian
- Spanish

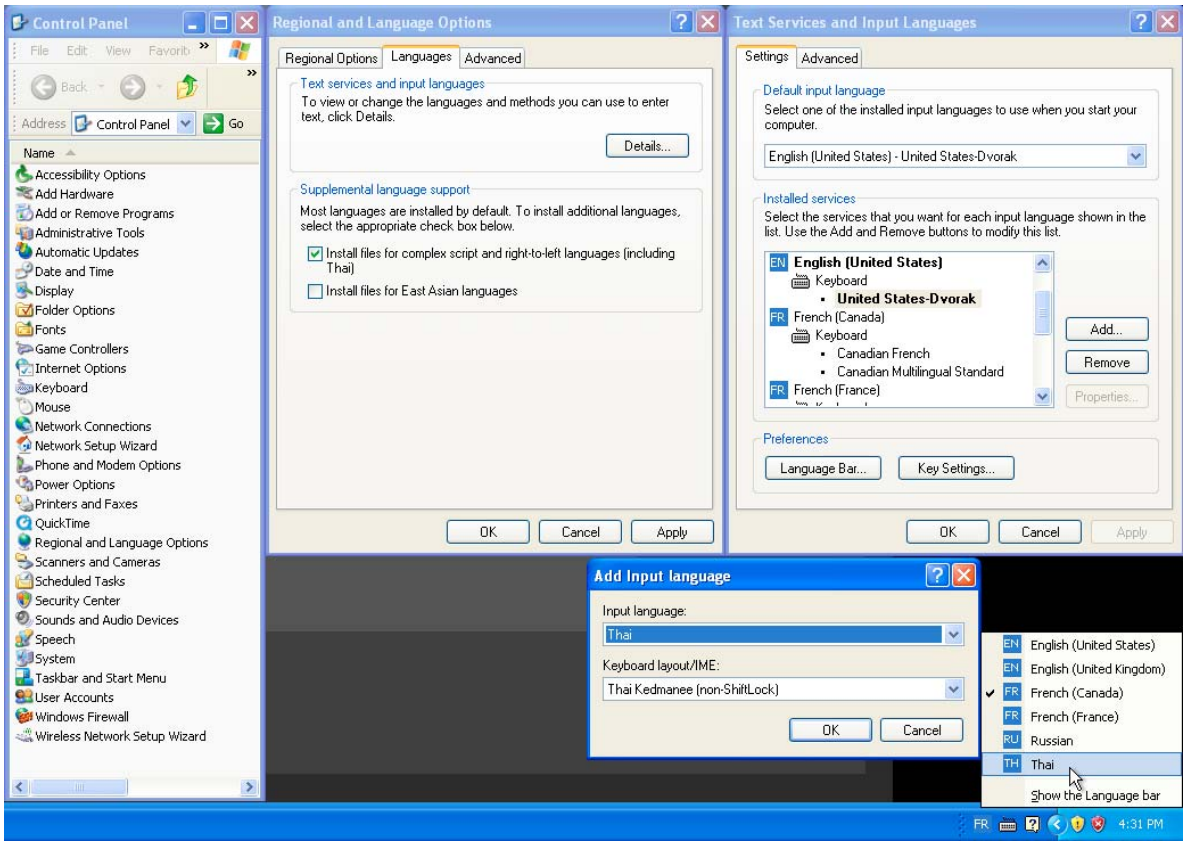
The language that you are working with may not figure in this list. Unless it is a Right-to-Left language, then it is officially supported. If you encounter a generic Unicode issue with any supported language, then contact the Adobe Director development team. This will allow them to provide critical fixes on a case-by-case basis.

Setting up input languages on Windows

The Microsoft® Windows® XP Service Pack 2 (SP2), Microsoft Windows 2000, Windows Vista™, and Windows 7 operating systems have Regional and Language settings, which can be used to add additional languages for keyboard input. These languages and speech settings appear in the Language bar on the desktop. After you select a language and set up a localized keyboard, you can start typing the required text in the document. The keyboard layouts are defined by Microsoft.

1. Install the required fonts in the Windows Fonts directory. For east-Asian languages like Japanese, Chinese (S and T), and Korean, install the corresponding language packs.
2. On your computer, open the Control Panel and double-click the Regional And Language Options icon to open the Regional And Language Options dialog box.
3. Click the Languages tab.
4. Click the Details button to open the Text Services And Input Languages dialog box.
5. In the Settings tab, click the Add button.
6. Select a language from the Input language list.
7. Click OK. The selected language is included in the Installed services list.
8. Select the desired language in the Default input language list.
9. Click Apply, and then click OK to save the settings, and close the Text Services And Input Languages dialog box.
10. Click the Regional Options tab, and then choose the language you selected in the default input language list.

Click Apply and then click OK to save the settings and close the Regional And Language Options dialog box. The Language bar or the Input Method Editor (IME) appears in the system tray of your computer.

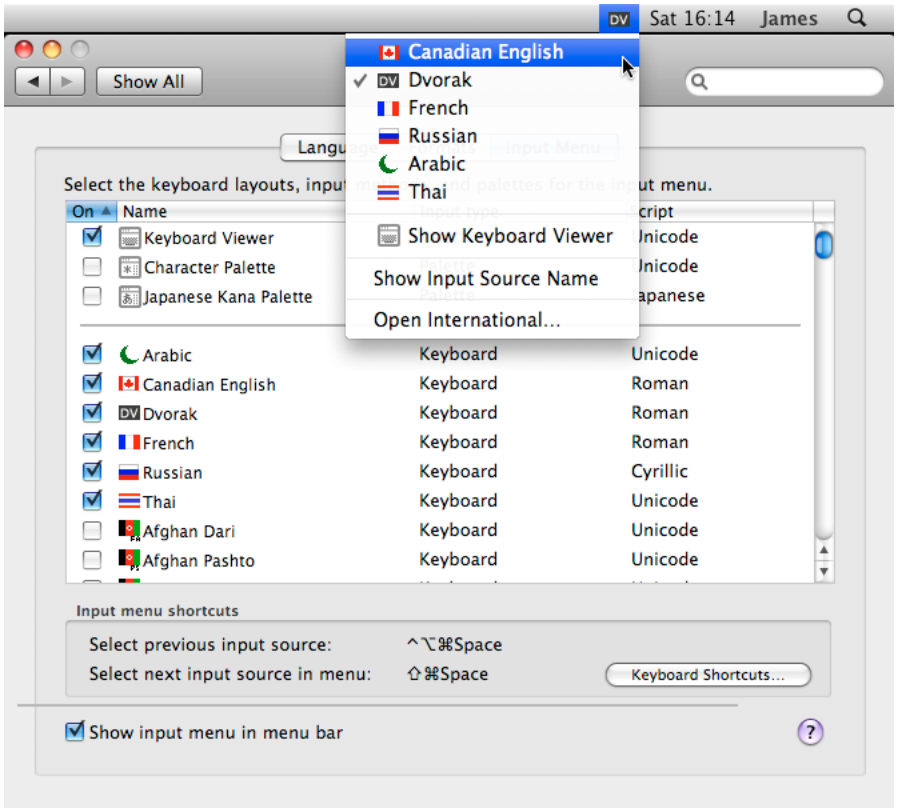


Choosing an input language on Windows XP

Setting up input languages on OS X 10.6 for Macs with Intel processors

You do not have to install language packs on OS X 10.6 for Macs with Intel processors, which come equipped with language packs.

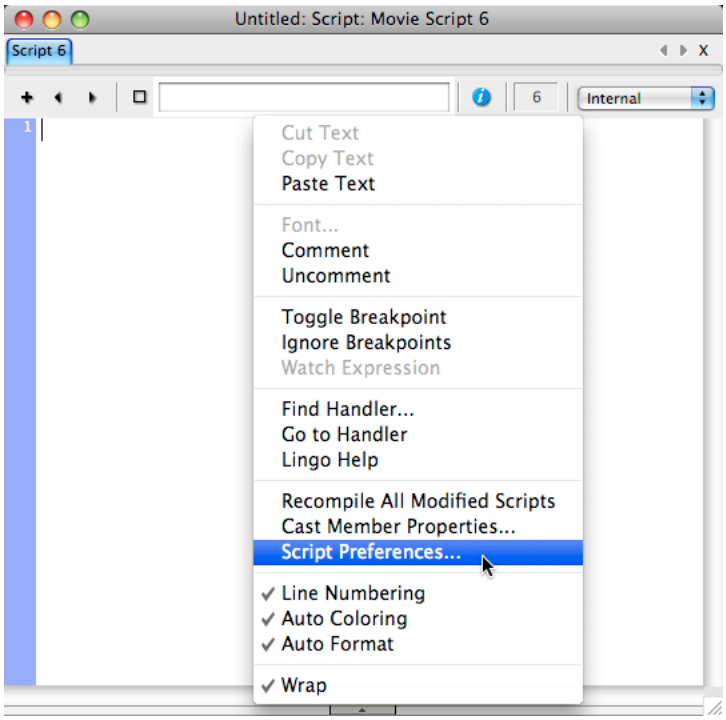
- Click System Preferences to open the System Preferences dialog box.
- Click International. The International dialog box appears.
- Click Input Menu.
- Select a language from the list that appears.
- Check the Show Input In Menu Bar check box to allow rapid switching of input methods
- Close the Input Menu.



The Input Menu anchor of the International pane of the System Preferences window

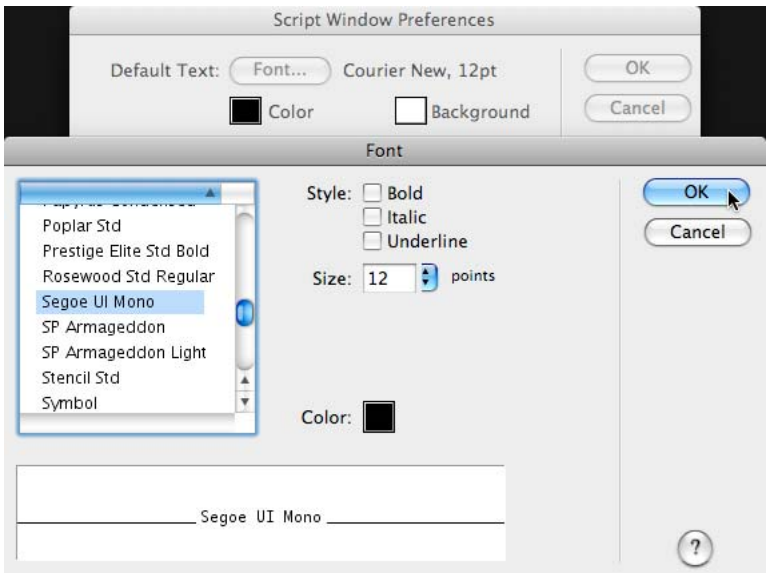
Using Unicode in scripts

Director supports the use of Unicode in scripts and in the Message window. The default font for scripts may not contain the characters that you need to display. To change the font used to display scripts, right-click on the Script window, and select Script Preferences from the contextual menu.



Right-click on the Script window to select Script Preferences

In the Script Window Preferences dialog window, click on the Font button, and select your preferred font, size, style, and color.



Selecting a new font for the Script and Message windows

Many developers prefer to use a monospaced font for writing scripts. There are not many monospaced fonts available which support a wide range of Unicode characters. You can try Ascender Uni Duo, which is relatively more complete.

Creating Director movies in multiple languages

You can use Director's Unicode support to create movies in languages other than English.

Select the language from the language bar (Windows) or International settings (OS X 10.6 for Macs with Intel® processors).

- Open Director.
- Open a text field (such as a text editor or script editor) in which you want to type in the selected language.
- From the font options in the text editor, select a font for the selected language. For the script editor or message window, select Edit > Preferences > Script, and click the Font button in the Script Window Preferences dialog box to set the font.
- Start typing content in the selected language.
- Before you input and display multilingual text in your document, configure the regional language or locale settings on your computer to add the required languages for keyboard input.

Embedding Unicode fonts

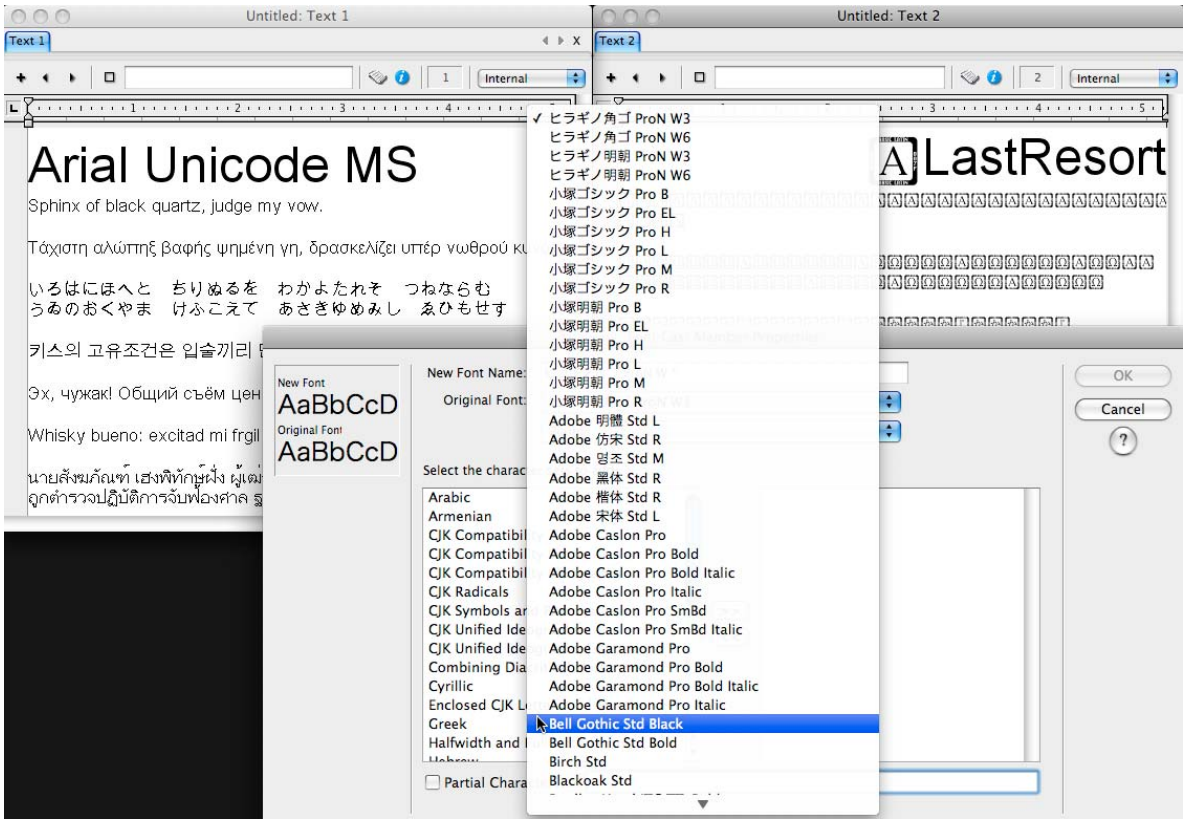
To ensure that your end-users can see text written in languages that require specific fonts, you can create a Font member and deliver it with your movie.

Only fonts that can be converted to an outline are available for embedding as a Font member. To see the full list of outline fonts available on your development computer, execute the following line in the Message window:

```
put member(1).outlineFontList()
```

Note: *In Director, certain fonts that are considered outline fonts on Windows are not available as outline fonts on Macintosh, and therefore cannot be converted to Font members on the Macintosh platform.*

As the illustration below shows, the font Arial Unicode MS can be used to display many different writing in a Director text member on the Macintosh platform, but it does not appear in the list of fonts that can be converted to a Font member.



Certain fonts that support a wide range of writing systems cannot be converted to a Font member on Macintosh

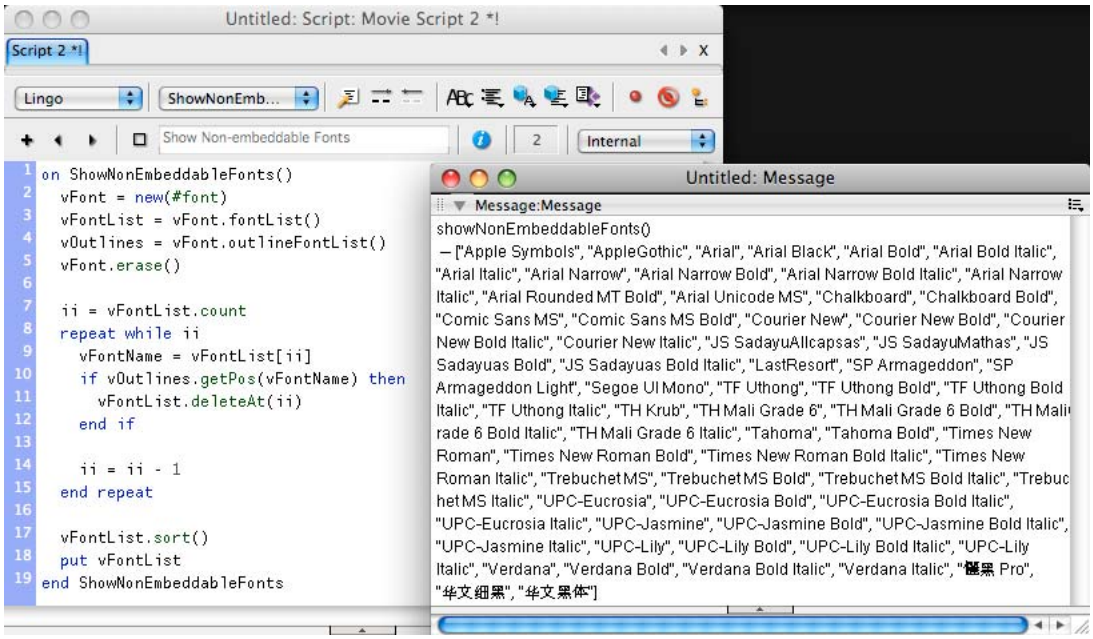
Here is a short list of TTF fonts that are commonly installed on both Mac OS X and on Windows, but which can only be converted to a Font member on Windows:

- Arial
- Arial Black
- Comic Sans MS
- Courier New
- Tahoma
- Times New Roman
- Trebuchet MS
- Verdana

Here is a Lingo movie script handler that will print out the names of the fonts that are available to Director, but which cannot be converted to a Font member.

```
on ShowNonEmbeddableFonts()
  vFont = new(#font)
  vFontList = vFont.fontList()
  vOutlines = vFont.outlineFontList()
  vFont.erase()
  ii = vFontList.count
  repeat while ii
    vFontName = vFontList[ii]
    if vOutlines.getPos(vFontName) then
      vFontList.deleteAt(ii)
    end if
    ii = ii - 1
  end repeat
  vFontList.sort()
  put vFontList
end ShowNonEmbeddableFonts
```

On Windows, you will find that this list consists mostly of bitmapped fonts with the FON extension. On Macinstosh, it may include a number of fonts in the True Type Format (TTF), despite the fact that these fonts are designed as outlines using Apple technology. Click here for more information on the TTF format.



Displaying a list of fonts that cannot be converted to Font members on Macintosh

Note: *Before purchasing a license for a font that you plan to convert to a Font member for your Director application, you may wish to check that the font appears to Director as an outline font on the platform for which you are purchasing the license*

Storing text in any character set

In Director 11, strings must contain only valid UTF-8 characters. This limitation causes problems if you want to represent non-UTF-8 characters in Lingo. For example, the null byte %00000000 is not a valid UTF-8 character. Click here for more examples. As a result, reading or writing binary data to a file is not possible in Director 11.

As a solution, Director 11.5 provides a new Lingo object called `ByteArray`. The `ByteArray` object can be used to store text in any character set, as well as non-text data. In addition, it also enhances some of the existing Director Xtras.

Use a `ByteArray` to do the following:

- Represent non-UTF-8 strings or any binary data
- Devise a framework to serialize the state of a Lingo/JavaScript object
- Read/write binary data, or a mix of binary and text data, from a file
- Handle text files for any character set
- Download or upload non-UTF-8 data from the Internet

Implement algorithms in Lingo that require byte-level access. For example, implement an encryption algorithm in Lingo.

Character sets in Lingo

Lingo exposes methods for querying the supported character sets on a system.

Director supports a subset of the style character-set definitions specified by the IANA. Character sets of ten languages are supported.

Character-set conversion rules

The rules for character conversion are as follows:

- The default character set in all cases is UTF-8.
- If a character cannot be represented in the specified character set, Director substitutes it with the "?" character.
- Best practice is to use UTF-8 encoding wherever possible and to check the support status of a character set before using it.

XML Xtra

XML documents can be created using any encoding, and encoding information is embedded using this XML declaration tag:

```
<?xml version="1.0" encoding="utf-8"?>
```

Further, XML documents can potentially embed binary data blobs inside them, making it impossible to represent the entire XML document using UTF-8.

Director 11.5 introduces a new method in the XML parser to handle byte arrays. The XML parser uses the XML declaration tag of the document to identify the text encoding of a document.