Gelato® 2.1

Technical Reference

NVIDIA®

Editor: Larry Gritz

Date: 29 September, 2006
(with corrections, 11 January, 2006)

Document ID: AS-00951-001_v2.1.04

Copyright 2004-2006 NVIDIA Corporation. All Rights Reserved.

The specification manual (the “Manual”) provided herewith may be reproduced by user for internal use only so long as such copies of the Manual contain the NVIDIA copyright notice as shown above. Subject to the terms and conditions contained herein, user may freely utilize the Manual to create calls, libraries and/or software that implement the specifications set forth in the Manual (“Implementations”). Provided, however, user shall not use NVIDIAs copyright notice or use language to the effect that the Implementations are “Gelato®-Compliant” or “NVIDIA®-Certified” unless (1) the Implementations meet all the specification requirements in the then-current Manual and (2) NVIDIA has verified in its sole discretion that the Implementations meet the specification requirements in the then-current Manual. NVIDIA disclaims any and all warranties with respect to the Manual, the Implementations created therefrom, and any use of or modifications to the Implementations, and user shall not make any representations or warranties on behalf of NVIDIA with respect to the Implementations or its use, and/or the distribution of the Implementations or such modifications. While NVIDIA may in its sole discretion verify that certain Implementations meet certain specification requirements in the then-current Manual, it shall not be obligated to do so. User acknowledges and agrees that if and when NVIDIA verifies an Implementation submitted by user, such verification is not on-going since the Manual may be updated and modified from time to time in NVIDIAs sole discretion. In the event that NVIDIA verifies that certain Implementations meet the specifications requirements in the then-current Manual, user may incorporate the following language only with respect to such verified Implementations: “This Implementation is Gelato®-Compliant as of [Date of NVIDIA Verification]. This Implementation is based on the specification requirements set forth in the Gelato Manual provided by NVIDIA® Corporation.”

NO WARRANTY

THE ACCOMPANYING MANUAL AND SERVICES (IF ANY) PROVIDED BY NVIDIA TO USER ARE PROVIDED ”AS IS.” NVIDIA DISCLAIMS ALL WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF TITLE, MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.

LIMITATION OF LIABILITY

NVIDIA SHALL NOT BE LIABLE TO USER, USERS CUSTOMERS, OR ANY OTHER PERSON OR ENTITY CLAIMING THROUGH OR UNDER USER FOR ANY LOSS OF PROFITS, INCOME, SAVINGS, OR ANY OTHER CONSEQUENTIAL, INCIDENTAL, SPECIAL, PUNITIVE, DIRECT OR INDIRECT DAMAGES (WHETHER IN AN ACTION IN CONTRACT, TORT OR BASED ON A WARRANTY), EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. THESE LIMITATIONS SHALL APPLY NOTWITHSTANDING ANY FAILURE OF THE ESSENTIAL PURPOSE OF ANY LIMITED REMEDY. IN NO EVENT SHALL NVIDIAS AGGREGATE LIABILITY TO USER OR ANY OTHER PERSON OR ENTITY CLAIMING THROUGH OR UNDER USER EXCEED THE AMOUNT OF MONEY ACTUALLY PAID BY USER TO NVIDIA FOR THE MANUAL PROVIDED HEREWITH.

VERIFICATION

NVIDIA has no obligation to provide any verification services with respect to the implantations created pursuant to the specifications set forth in the Manual.

Contents

1 Introduction **1**

I The Major APIs **7**

2 The Gelato C++ Scene API **9**
2.1 Basic Concepts 9
2.2 Parameters 13
2.3 Renderers, Cameras, Outputs, and Rendering 14
2.4 Attributes 21
2.5 Transformations 29
2.6 Shaders and Lights 32
2.7 Motion Blur 35
2.8 Geometric Primitives 36
2.9 Procedural Geometry Generators and Scene Files 44
2.10 Error Management 47
2.11 Re-rendering 50
2.12 Example Scene Specification 51

3 Pyg: A Python-Based Scene File Format **53**
3.1 Motivation 53
3.2 Basics 54
3.3 API Calls 55
3.4 Example Pyg Scene File 60
3.5 Calling Gelato from Python 61
3.6 Efficiency Issues 61
3.7 Binary Pyg 62
3.8 Conversion to Pyg: the `topyg` program 63

4 Gelato Attributes and Commands **67**
4.1 Camera Attributes 67
4.2 Output Attributes 73
4.3 Scene-wide Attributes 76
4.4 Per-object Attributes 87
4.5 Commands 95

5 Shading Language **99**
5.1 Basics 99
5.2 Data Types 100
5.3 Shader Parameters 104
5.4 Shader Metadata 106
5.5 Language Syntax 107
5.6 Expressions 114
5.7 Global variables 116
5.8 Standard Library Functions 119
5.9 Formal Language Syntax 150

II Using Gelato **155**

6 Running Gelato **157**
6.1 `gelato` Command Line Operation 157
6.2 Startup File 159
6.3 Environment variables 159
6.4 Remote display with `iv` 160
6.5 Preview Modes 160
6.6 Network-Parallel and Multi-Threaded Rendering 162

7 Cameras and Image Output **165**
7.1 The Camera 165
7.2 Image Resolution and Framing 172
7.3 Image Output 174
7.4 Gelato's Bundled Image I/O Plugins 177
7.5 Antialiasing and Filtering 181
7.6 Stereo Rendering 184
7.7 Multiple Cameras 185

8 Using Shaders **187**
8.1 Shader Basics 187
8.2 Compiling Shaders with `gslc` 188
8.3 Listing shader parameters with `gsoinfo` 190
8.4 Shader Metadata Conventions 191
8.5 Using Units 194

9 Textures **197**
9.1 Converting images to texture with `maketx` 197
9.2 Texture Formats 201
9.3 Cube-Face Directions 206

10 Illumination **207**
10.1 Shadows . . . 207
10.2 Reflections . . . 220
10.3 Indirect Illumination . . . 225
10.4 Ambient Occlusion . . . 228
10.5 HDRI and Image-Based Lighting . . . 233
10.6 Caustics . . . 235
10.7 Subsurface Scattering . . . 238

11 Sorbetto Re-rendering **245**
11.1 Re-rendering Rules . . . 245
11.2 Idioms for iterative relighting . . . 246
11.3 Idioms for improving performance . . . 248

III Developer Tools **251**

12 Generator Plugins and Scene File Readers **253**

13 Image I/O Plugins **257**
13.1 Formats and Plugins . . . 257
13.2 Writing Image I/O Plugins . . . 258
13.3 Image Writers . . . 259
13.4 Image Readers . . . 264
13.5 Other Helper Functions . . . 267
13.6 Example: JPEG ImageIO . . . 269
13.7 Using ImageIO plugins to handle pixels directly . . . 274

14 Calling C++ functions from Shaders **277**
14.1 Calling Conventions . . . 277
14.2 Compiling . . . 283
14.3 Basic Example . . . 283
14.4 Texture, Noise, Attributes, and Symbols . . . 284

15 Examining compiled shaders with `libgsoargs` **289**
15.1 The `GsoArgs` class . . . 289
15.2 Using `gsoargs.h` and `libgsoargs` . . . 291
15.3 Example: `gsoinfo` source code . . . 293

IV Appendices **297**

A Public API Header Files **299**
A.1 gelatoapi.h . . . 299
A.2 paramtype.h . . . 304
A.3 errormanager.h . . . 307

A.4 shadeop.h 310
A.5 imageio.h 315
A.6 gsoargs.h 322

B Glossary **325**

Index **331**

1 Introduction

Welcome to NVIDIA's Gelato ®.

Gelato is a new hybrid software/hardware rendering system designed to flexibly create beautiful final imagery for film and other high-end applications. The two stated goals at the outset of the design of Gelato were:

- To capitalize on revolutionary new programmable graphics hardware to accelerate rendering operations, take advantage of the faster-than-Moore's-law rate of improvement in graphics hardware over time, and work toward having interactive previewing of final renderings (even if those final renderings still take minutes or hours).

- Never to compromise on image quality, robustness, power, programmability, flexibility, or any other property necessary for a top-of-the-line film rendering system. In particular, we will never let Gelato's use of graphics hardware result in an inferior rendering product.

We believe that we have achieved both of these goals, and are proud to present the Gelato renderer.

Things are always in flux

This *Technical Reference* manual[1] is a living, growing document. The C++ API and shading language are thoroughly documented (in a reference manual kind of way), but much of the technical reference is currently limited to not much more than a brief description of the command-line options to the various programs. We expect the *Technical Reference* to continue to grow and become more complete with each successive version of Gelato.

Please check the "Release Notes" document in the software distribution for up-to-date details about exactly what is unimplemented, known bugs, etc. Subsequent releases, which we anticipate will come in rapid succession, will each have more implemented features, fixed bugs, documentation, examples, and so on.

We appreciate your patience. We also seek your feedback. These API's and features are not set in stone. Please do not hesitate to send us any complaints or suggestions (or compliments).

[1] We call this document the "technical reference manual" as an homage to the document of the same name that IBM published for the original PC. That document's unusual openness and technical detail is widely credited for the PC architecture's rapid adoption and the subsequent transformation of the previously-fractured microcomputer industry.

New API's

This technical reference manual documents several API's: a C++ scene description API, a Python binding, a shading language, and several lesser API's for plugins of various types. While we are not currently seeking any formal standardization of these specifications, we hope that their use will become widespread, resulting in multiple familes of compatible products and services. To that end, we encourage the development of software that uses and implements the API's discussed herein, with only the minor restrictions set forth in the notice on page ii. Enjoy.

Unique Features

Sometimes it can be hard to find the needles buried in the haystack. So at this time, we'd like to call your attention to some of the really interesting and unique features of our API and software. Please dig deeper in the manual, try out, or ask us for more information about the features that intrigue you.

- **Hardware accelerated.** Gelato is designed from the ground up to have various internal operations accelerated by commodity programmable graphics hardware (such as NVIDIA's QuadroFX® line). This is completely transparent to the user, and does not in any way affect flexibility or image quality. It can, however, greatly improve performance. Subsequent releases of Gelato will take even more advantage of graphics hardware, and future graphics hardware will be even faster and more capable. For the past few years, graphics hardware has been doubling in speed every 6-12 months, whereas CPU's have been doubling in speed roughly every 18 months. So we believe that renderers based on graphics hardware will not only perform well now, but will over time diverge rapidly from the performance of CPU-only renderers.

- **Re-rendering.** When repeatedly rendering a frame with just the lighting changing on each iteration, Gelato's *Sorbetto* option allows incremental re-rendering an order of magnitude or more faster than a full rendering of the frame. This re-rendering produces identical final pixels to a "full" render – including antialiasing, motion blur, transparency, and ray tracing.

- **Stereo rendering and multiple cameras.** Gelato will render multiple cameras at once much faster than rendering each camera separately, subject to the limitation that the shading (colors and tessellation decisions) are made from a single "shading" camera. This is especially useful for rendering stereo views, with both right and left eyes rendering simultaneously at substantial time savings and requiring only one license.

- **Ray tracing.** Gelato is capable of efficient ray tracing of large scenes, including ray-traced shadows, reflections, indirect global illumination, photon-mapped caustics, and "ambient occlusion" visibility queries. Don't be fooled by Gelato's use of graphics hardware acceleration — we have not let that limit the feature set of the renderer.

- **New C++ API.** Gelato's main API is a modern, C++-based API. We have tried to keep it simple (few calls) and orthogonal (usually one best way to accomplish a task). There are only a few types of geometric primitives, but they are very general. All object and scene

attributes (such as surface color or camera shutter) are set through a single `Attribute` call. Custom variables (such as shader parameters and geometric primitive "vertex variables") are set through a single, simple `Parameter` call.

- **State queries and saved state.** A program or plug-in making C++ API calls to Gelato may ask for the current value of any graphics state attribute. There are also calls in Gelato's API's to save all or part of the current state, name it, and later restore all or part of that saved state. This makes it possible to easily transfer collections of attributes from one part of your scene hierarchy to another, non-descended, place in the hierarchy.
- **Scene format reader plug-ins.** We like to say that Gelato is "format agnostic." Rather than prescribing a single scene file format (forcing you to convert all data into that format), Gelato has a simple API for *scene format plug-ins*. When a file is input, the DSO/DLL for that format is dynamically loaded and told to read the scene file. Thus, you may store your scene in any format for which we, you, or a third party provide such a plug-in, and you may freely mix different files in different formats within a single scene.
- **Python binding.** Gelato includes a scene format plug-in that reads Python scripts that make calls to our API. This provides an extremely flexible, fully scriptable method of scene input.
- **Layered shaders.** Instead of allowing only a single surface, displacement, and volume shader per object, Gelato allows, for example, several surface shaders to be called in turn, with the user able to specify that one shader's outputs be connected to another shader's inputs. This allows one to *compose* the operations of component shaders without modifying (or even having access to) the source code of any of the shaders involved. For example, you can make any surface glossy by layering a "gloss" shader atop any other shader, without needing the source code to either.
- **Shader metadata.** Gelato's Shading Language allows you to embed annotations in your shaders that describe the shader and its parameters. These "metadata" are embedded in the compiled shader and may be read by other applications. This is intended primarily to allow shaders to give UI hints to applications that allow users to adjust shader parameters.
- **Units.** Gelato allows you to associate physical units (e.g., cm, m, ft, s) with the scene scale, and to convert among units in shaders. This allows you to write shaders that are physically accurate and also portable across scenes or productions regardless of that modeling units used.
- **No uniform or varying.** Gelato's Shading Language does not contain the keywords "varying" or "uniform," and there is no need to use such declarations in your shaders. Do not worry about performance — when shading many points at once, Gelato tracks which variables take on different values at different points, and which ones have the same value at all points. But such things are handled automatically and dynamically at run time, and therefore much more efficiently than if the user had been responsible for making the correct declarations. Since there is no way to declare uniform variables anyway, that means that strings may be varying. It also means that any shader parameter may be overridden by an interpolated "primitive variable" without any special declarations.

- **Place the camera, not the world.** The Gelato universe starts off in *world space*. You may place a `Camera` within that world, just like you would place lights or objects. (In fact, you can even place multiple cameras, although currently only the first declared camera is rendered; we hope to lift this restriction in the future.) There is no need to treat the camera as the original origin, carefully placing the world with the inverse transformation. Of course, if there is no `Camera` at all, Gelato correctly infers that you intended the original coordinate system to be the camera, and that world space is marked by the `World` call.

- **Geometry sets.** It is possible to name groups of primitives (and of course, one primitive may be in many groups). These named geometry sets may be used to specify collections of primitives visible by a particular camera, used for ray tracing, comprising area lights, and potentially for many other future uses.

- **No eyesplits.** Gelato will never complain about "eye splits." There are no parameters to adjust. No missing geometry. No artifacts of any kind. Never.

- **Image I/O plug-ins.** Similar to Gelato's scene format agnosticism, Gelato also has no required image formats for either input or output. There is a simple API for writing plug-ins that read and write image formats. An image output plug-in can allow you to have the renderer write out image files in the format of your choice, and image input plug-ins can be used to read texture, display images, and convert images from one format to another. Gelato ships with image I/O plug-ins for TIFF, OpenEXR, JPEG, PPM, PNG, HDR, DDS, Maya IFF, and Targa formats, as well as one for displaying to an interactive image viewier.

- **Samples are cheap.** The default sampling rate for Gelato is 4x4 spatial samples, and 16 time samples per pixel. This is not expensive, and when you have scenes for which it is not good enough, you should feel free to turn up the `"spatialquality"` and `"temporalquality"` camera attributes. You can make these absurdly high with surprisingly low impact on overall rendering time. Furthermore, spatial and temporal (motion blur) sampling rates are controlled independently, so there's no need to oversample spatially just to get good motion blur, or vice versa.

- **Sparse spatial databases in shaders.** In addition to being able to save the sparse caches for occlusion and indirect illumination, shaders may create their own such databases to store the results of arbitrary computations, save them to disk, or read existing databases to disk for quick interpolation.

- **Preview modes.** Using the `-preview` flag (or equivalent `Attribute` commands) it is possible to quickly preview a Gelato scene at reduced quality.

Basic Gelato vs. Gelato Pro

Gelato comes in two editions: an unsupported "basic" Gelato that is distributed at no charge and a fully supported, commercial Pro edition.

NEW!

Basic Gelato does not limit resolution or image quality, watermark or otherwise change the output, limit features, or degrade performance, except as described below. It's free, but unsupported — use at your own risk and without expecting support. You may still use it for commercial work. Enjoy!

With the addition of a license file (available from NVIDIA after registering and paying for the software), Gelato Pro is enabled. When no license is found at all, only the basic features will be enabled. If there is a license, but too many renders are happening simultaneously, `gelato` will issue an appropriate message and exit. So you should not have to worry about lapsing in and out of Pro mode just because you are using all your licenses at once.

Gelato Pro is distinguished from basic Gelato in the following ways:

- Gelato Pro supports rapid re-rendering of a frame when only lights change (also known as "Sorbetto").
- Gelato Pro supports multithreading (using more than one CPU on a single machine when rendering a single frame).
- Gelato Pro supports network-parallel rendering (using multiple machines on a network when rendering a single frame).
- Gelato Pro comes in native 64-bit versions, whereas basic Gelato is 32-bit only.
- Gelato Pro allows users to write DSO Shadeops (see Chapter 14), basic Gelato does not.
- Gelato Pro is fully supported by NVIDIA, including testing, tech support, help, updates, and early access to new versions and beta tests.

Over time, some new features or optimizations may be put in Gelato Pro only, and some old features that were previously Pro-only may be migrated into basic Gelato. But we will never take a feature previously in basic Gelato and suddenly make it Pro-only.

Sections of the *Technical Reference* dealing with facilities only available in Gelato Pro have this marginal notation.

Gelato Pro

Comments / Corrections

Please contact `gelatosupport@nvidia.com` with comments or corrections to this document.

Part I

The Major APIs

2 The Gelato C++ Scene API

This chapter describes Gelato's C++ API for describing scenes. Gelato's API routines can loosely be broken down into those that alter the graphics state *attributes*, and those that create geometric primitives. Some attributes are properties that apply to the entire scene (for example, camera parameters), but other attributes are properties that may vary from object to objects (for example, color, shader assignments, and transformation). Geometric primitives, when declared, inherit the current attribute state (including transformations and shader assignments).

2.1 Basic Concepts

2.1.1 Classes, Namespaces, and Headers

The primary API for talking Gelato is through a C++ class called `GelatoAPI`, which is defined in the header file `gelatoapi.h`.

A number of other header files contain useful types, classes, and functions:

- `paramtype.h` is used by `gelatoapi.h` and provides a way to express data types to certain Gelato API routines.
- `errormanager.h` is used by `gelatoapi.h` and provides a definition of overrideable error managers and handlers.
- `gsoargs.h` provides the `GsoArgs` class that is used to get information about the parameters of compiled shaders (see Chapter 15).
- `imageio.h` contins definitions that you will need in order to create plugins that allow Gelato to read or write image formats (see Chapter 13).
- `shadeop.h` provides definitions useful for writing C++ code that may be called from your shaders (see Section 14). `noise.h` provides declarations of noise-like functions that may be called from within shadeop plugins.
- `color.h` and `vecmat.h` provides classes for 3-colors, 3- and 4- vectors, 4×4 matrices, and 3D bounding boxes. None of these classes are required to use the Gelato API, but we use them internally and thought that others may find them useful.

The definitions in the above header files are in `namespace Gelato`. You may reference these types and functions either by explicitly using the `Gelato::` prefix, or by simply stating

```
using namespace Gelato;
```

after including the header files (then the prefixes are unnecessary). For brevity, the remainder of the *Technical Reference* will assume that you are `using namespace Gelato`, and will therefore omit the `Gelato::` prefix.

2.1.2 Renderer Object

The renderer itself is a C++ object of type `GelatoAPI`. The `GelatoAPI` class provides only an interface; it contains no data members nor non-virtual methods. Actual renderer implementations are assumed to be subclassed from `GelatoAPI`, and thus inherit the `GelatoAPI` interface.

Renderer objects may be created by calling `GelatoAPI::CreateRenderer()`, which returns a pointer to a `GelatoAPI` object. All subsequent communication to the renderer is through `GelatoAPI` class methods called through the pointer to the renderer. Deleting the renderer frees all resources associated with the renderer. For example:

```
GelatoAPI *r = GelatoAPI::CreateRenderer();
r->Camera ("main");
... API calls through r ...
r->Render ("main");
delete r;   // Finished with this renderer
```

Multiple renderers may be created in succession or simultaneously, and API calls to different renderers may be freely interleaved. API calls to one renderer do not have any semantic side effects upon other renderer objects, however multiple simultaneous renderers may have performance-related side effects upon each other if they perform functions that requires competition for system resources.

Renderers may come in different implementations, selected by an optional parameter to `CreateRenderer`. While the default renderer is expected to produce images of high quality, alternate renderer implementations may provide low-quality previews, or even perform tasks other than creating images (such as merely archiving the sequence of API calls for later playback with `Input`). Optional arguments to `CreateRenderer` may also be used to supply a user-defined error manager / error handler.

2.1.3 Hierarchical Graphics State

The renderer maintains a *graphics state machine*, which is a set of names and values, called *attributes*. Some of those parameters (such as image resolution, camera projection, and output files) apply to the entire scene or to a particular camera. Other parameters may be different for each geometric primitive. Per-object attributes include such things as shader assignments and transformations.

The attribute state is hierarchical, in the sense that attributes (and, if you want, just the transformations) may be pushed and popped using a stack.

A scene specification is mostly a series of declarations of geometric primitives, with calls to alter the attribute state between primitives.

2.1.4 Copy-on-write versus Modify mode

Normally, when geometric primitives (such as patches) are declared, they permanently take on the characteristics of the *current* attribute state, including their object transformation. From that

point onward, the primitive keeps a reference to that attribute set, which is used as the primitive is processed. Further changes to attributes may affect yet-to-be-declared primitives, but will not change the attributes of any primitives which have already entered the system. This is known as *copy-on-write* semantics, because once a geometric primitive references an attribute state, attempts to change (write) an attribute will copy the entire attribute state and change the copy, not the original set of attributes that is already referenced by the primitive.

There is also *modify mode*, triggered by the `Modify` API call, which can refer to one or more existing attribute states by name. When in modify mode, changes to attributes will affect the original attribute states under modification, even if primitives already refer to them. This is primarily used for re-rendering, although there are also many uses for modify mode even when rendering a single frame.

2.1.5 Data Type Declarations

The renderer already knows about many parameters that you might want to pass, such as the `"fov"` attribute to specify field of view, or the parameter `"P"`, which is used to pass 3D positions of geometric control vertices. These names are all *pre-declared*, which means that the renderer recognizes the names, knows what type of data they represent, and knows how to use the data.

In order to make the renderer extremely expandable, there are a number of ways that users can pass arbitrary data, for example, to be later used by a user-supplied shader. The renderer won't know what type of data you are passing in these cases, but there are two ways that you can convey the type information: either by embedding the type declaration in the attribute or parameter name, or by passing an explicit `ParamType` class. For example,

```
r->Parameter ("vertex point P", &Pvalues);
```

Here we pass a parameter `"P"`, which has data type `point` and has interpolation type `vertex`. Alternately, we could make the equivalent call that explicitly conveys the type using a `ParamType` structure:

```
r->Parameter ("P", ParamType(PT_POINT,INTERP_VERTEX), &Pvalues);
```

The definition of the `ParamType` class is:

```
class ParamType {
public:
    // Construct from base type and optional interp (assume non-array)
    ParamType (ParamBaseType basetype, ParamInterp interp=INTERP_CONSTANT);

    // Construct with array length and optional interp
    ParamType (ParamBaseType basetype, short arraylen,
               ParamInterp interp=INTERP_CONSTANT);

    // Construct from a string (e.g., "vertex float[3]").  If no valid
    // type could be assembled, set basetype to PT_UNKNOWN.
    ParamType (const char *typestring);
    ...
};
```

Valid base types include PT_FLOAT, PT_INT, PT_COLOR, PT_POINT, PT_VECTOR, PT_NORMAL, PT_MATRIX (16 floats forming a 4x4 matrix), and PT_HPOINT (4 floats forming a homogeneous point). Valid interpolation types include INTERP_CONSTANT (the default, no interpolation), INTERP_PERPIECE, INTERP_LINEAR, and INTERP_VERTEX.

The ParamType class is defined in the header file paramtype.h, which is automatically included by gelatoapi.h. ParamType is in namespace Gelato.

2.1.6 Interpolation Type

A common task is to pass user data along with a geometric primitive, and have the renderer automatically interpolate the data over the primitive and make it available to the user's shaders. In addition to specifying the data type, as above, you may also specify the *interpolation* type (sometimes called *storage class*). In addition to being able to pass a single value for the entire primitive (as above, with no interpolation type specified), all primitives support the interpolation types vertex, which requires the same number of values as the control vertex positions ("P" or "Pw") and is interpolated in the same manner as the positions; and linear, which is linearly (or bilinearly) interpolated across the primitive, and therefore may have a different number of values than the control vertices. Some primitives also support the interpolation type perpiece, which supplies one value for each section of the primitive (one per face for a Mesh, one per curve for a Curves). It's easy to see how interpolation type is combined with data type in the following example:

```
int nverts[1] = {3};      // Number of vertices for each face
int verts[3] = {0, 1, 2}; // Vertex index sequence
float Pvals[3][3] = { /* points */ };
float Cvals[3][3] = { /* colors */ };
float temp = 98.6;

r->Parameter ("vertex point P", &Pvals);
r->Parameter ("vertex color C", &colvals);
r->Parameter ("float temperature", &temp);
r->Mesh ("linear", 1, nverts, verts);
```

2.1.7 Parameter Lists

Many API routines take a variable number of parameters. Among these routines are all geometric primitive and shader declarations. For example, the Mesh call above passes positions, colors, and user data named "temperature." Below is another illustration using a slightly different form of passing parameters, where the type is explicitly passed as a ParamType rather than implicitly passed as part of the parameter name:

```
float Pvals[3][3] = { /* points */ };
float widthvals[3] = { /* widths */ };

r->Parameter ("P", ParamType(PT_POINT,INTERP_VERTEX), &Pvals);
r->Parameter ("width", ParamType(PT_FLOAT,INTERP_VERTEX), &widthvals);
r->Points (3);
```

Using types embedded in the parameter names is equivalent to passing an explicit ParamType. You are free to use whichever is more convenient for your application. Humans tend to prefer to use the type/name strings, whereas various automatic or machine-driven translations prefer passing a ParamType.

2.2 Parameters

GelatoAPI::**Parameter** (const char *name, float value)
GelatoAPI::**Parameter** (const char *name, int value)
GelatoAPI::**Parameter** (const char *name, const char *value)
GelatoAPI::**Parameter** (const char *name, const float *value)
GelatoAPI::**Parameter** (const char *name, const int *value)
GelatoAPI::**Parameter** (const char *name, const char **value)
GelatoAPI::**Parameter** (const char *name, const void *value)

GelatoAPI::**Parameter** (const char *name, ParamType type, float value)
GelatoAPI::**Parameter** (const char *name, ParamType type, int value)
GelatoAPI::**Parameter** (const char *name, ParamType type, const char *value)
GelatoAPI::**Parameter** (const char *name, ParamType type, const float *value)
GelatoAPI::**Parameter** (const char *name, ParamType type, const int *value)
GelatoAPI::**Parameter** (const char *name, ParamType type, const char **value)
GelatoAPI::**Parameter** (const char *name, ParamType type, const void *value)

Saves a single named parameter into the "pending parameter" list for subsequent use by `Command`, `Camera`, `Output`, `Shader`, `Light`, or a geometric primitive. Any of these routines will use all of the pending parameters, and will clear the pending list. Note that the `Parameter` routine does not copy the data itself, so the user should be careful not to overwrite or free data until after the `Command`, `Camera`, `Output`, `Shader`, `Light`, or a geometric primitive call has been made.

The *value* may be a `float`, `int`, or string (passed as a `char *`), or a pointer to an array of `float`, `int`, or `char *` values. Data consisting of multiple `float` values (including `color`, `point`, `vector`, `normal  hpoint`, or `matrix` data) should be passed using the `float *` version. The data type and declaration must match – for example, `"float fov"` may be called by pasing a `float` or a `float *`, but any of the other varieties of `Parameter` will produce a runtime error. It's also possible to simply pass a raw `void *` pointing to the data, but in that case there is no compile-time type safety.

Two flavors of `Parameter` exist: one passes an explicit *type* in the form of a `ParamType` object, while the other deduces the type from the *name* itself (as described in Section 2.1.5).

EXAMPLES:

```
r->Parameter ("float Kd", 0.5);
r->Parameter ("Ks", ParamType(PT_FLOAT), 0.25);
r->Shader ("surface", "plastic");   // Uses the Kd and Ks parameters

r->Parameter ("vertex point P", &P);
r->Parameter ("C", ParamType(PT_COLOR,INTERP_LINEAR), &C);
r->Patch ("linear", 4, 4);  // Uses the P and C parameters
```

2.3 Renderers, Cameras, Outputs, and Rendering

static GelatoAPI * GelatoAPI::**CreateRenderer** (const char **type*=NULL, Gelato::ErrorManager **err*=NULL)

CreateRenderer creates a renderer of given *type* (if *type* is unspecified, a default type is assumed), returning a pointer to a GelatoAPI object to which subsequent API requests may be sent. The renderer will be destroyed and its resources freed simply by deleting it.

Gelato recognizes the following renderer *types*:

- No *type*, or an empty string for *type*, indicates the default renderer, which is a full high-quality rendering of the scene.
- If *type* is "pyg", the renderer created will be one that outputs to stdout the Pyg equivalent of all API calls it receives (see Chapter 3 for details on the Pyg format).
- If *type* begins with "pyg" followed by whitespace, the renderer created will output Pyg corresponding to the API calls, writing to the filename specified by the remainder of the *type*. For example, CreateRenderer("pyg myfile.out") will write Pyg commands to the file "myfile.out".
- If *type* ends with ".pyg", the renderer created will output Pyg corresponding to the API calls, writing to the filename specified by *type*. For example, CreateRenderer ("myfile.pyg") will write Pyg commands to the file "myfile.pyg".
- If *type* is "remote", a renderer will be created as a separate process with communication going over a socket connection. Furthermore, the socket communication with a remote renderer is asynchronous and non-blocking, which means that you can send a Render call and get control back immediately, while the remote renderer is continuing to render the image.

If an ErrorManager * is passed in as the optional second argument, the renderer will use it for error reporting. If no ErrorManager is passed in, or if the pointer is NULL, a default error manager / handler will be created. See Section 2.10 for a description of how to override the error management behavior.

Multiple renderers may be created in succession or simultaneously, and API calls to different renderers may be freely interleaved. API calls to one renderer do not have any semantic side effects upon other renderer objects, however multiple simultaneous renderers may have performance-related side effects upon each other if they perform functions that requires competition for system resources.

EXAMPLE:

```
GelatoAPI *r = GelatoAPI::CreateRenderer ("out.pyg");
...
delete r;
```

`GelatoAPI::`**Camera** `(const char *`*`name`*`)`

Creates a camera located at the local (CTM) origin, facing in the direction of the local $+z$, with local $+x$ pointing toward the right of the screen and the local $+y$ pointing toward to top of the screen (note that this makes the camera inherently "left-handed").

There may be multiple `Camera` statements. All cameras that have corresponding `Output` statements will render images, but cameras without `Output` statements will not produce images (unless *no* `Output` statement is present at all in the scene, in which case the first camera declared will be automatically given a default `Output`). The first `Camera` declared defines the world-to-camera transformation and upon which all tessellation and shading computations are based (even if it has no image outputs).

The `Camera` function also creates a new geometry set with the given *name* and adds it to the active geometry set list (so subsequent geometry will, by default, be visible in all cameras). For a given camera, only objects included in the geometry set that corresponds to the camera name will be rendered.

Any pending parameters (set by `Parameter`) give camera and image formation attributes (summarized below and explained in detail in Section 4.1). `Camera` copies the parameter values and clears the pending parameter list (making it safe for the user to subsequently reuse or free the memory referenced by the preceeding `Parameter` calls).

All cameras must have distinct *name*s. If `Camera` is called with the *name* of an existing camera, the old camera defintition will be replaced by the new camera definition.

EXAMPLE:

```
r->SetTransform (M);
r->Parameter ("float fov", 32.0f);
int res[2] = { 640, 480 };
r->Parameter ("int[2] resolution", &res);
r->Parameter ("float fov", fov);
r->Camera ("maincam");
```

Camera Attribute	Meaning (default value)
"string projection"	Projection ("perspective")
"float fov"	Vertical field of view (90)
"float[4] screen"	Portion of the projection plane to image (-xres/yres, xres/yres, -1, 1)
"float near"	Near clipping plane distance (0.1)
"float far"	Far clipping plane distance (1e6)
"int[2] resolution"	Image resolution (640, 480)
"float pixelaspect"	Pixel aspect ratio (1)
"float[4] crop"	Subimage to render (0, 1, 0, 1)
"float[2] shutter"	Shutter open and close time for motion blur (0, 0)
"float fstop"	f/stop for depth of field (default: no DOF)
"float focallength"	Lens focal length (default: no DOF)
"float focaldistance"	Distance to sharp focus (default: no DOF)
"int[2] spatialquality"	Number of subpixel antialiasing regions (4, 4)
"int temporalquality"	Number of time values for motion blur (16)
"int dofquality"	Number of lens values for motion blur (16)
"int[2] limits:bucketsize"	Size of pixel rectangles used as work units (32, 32)
"string bucketorder"	Order of bucket traversal ("horizontal")
"int limits:bucketblocksize"	Size of bucket clustering (2)
"int limits:autopassreduction"	Reduce temporal & dof quality when not needed? (1)
"float preview"	If nonzero, activates preview mode and overrides shadingquality (0)
"int[2] preview:spatialquality"	Spatial quality override for preview mode (1, 1)
"int preview:temporalquality"	Temporal quality override for preview mode (1)
"int preview:dofquality"	DOF quality override for preview mode (1)
"int[2] preview:bucketsize"	Bucket size override for preview mode (2048, 2048)
"float preview:dicecurvature"	Dicing curvature metric override for preview mode (90)
"int preview:thincurves"	For preview mode, draw one of every *n* curves. (10)
"int preview:thinpoints"	For preview mode, draw one of every *n* points. (10)
"float preview:limitpixelsize"	For preview mode, objects with bounding boxes smaller than this (in pixels covered) are drawn just as boxes. (0)
"float preview:limitdistance"	For preview mode, objects farther than this distance are drawn just as bounding boxes. (1.0e38)
"string hider"	Selects alternate hider ("default")
"string shadertype"	Selects alternative shading system ("gsl")
"string stereo:projection"	Projection for stereo cameras. ("off-axis")
"float stereo:separation"	Camera separation for stereo cameras. (0)
"float stereo:convergence"	Convergence distance for stereo cameras. (0)
"string stereo:shade"	Which view to shade for stereo cameras. ("center")

`Renderer::`**Output** `(const char *`*name*`, const char *`*format*`,`
`const char *`*data*`, const char *`*camera*`)`

Specifies an output image for rendered pixels, for a particular camera.

The *name* parameter is a string that gives the name of an image file, or other destination. Multiple simultaneous output images (presumably each with different data) may be specified simply by having multiple `Output` statements with different *name*s. There is no set limit to the number of output images. If `Output` specifies an output name that already exists, the previous definition will be replaced by the new definition of that output.

The *format* is a string that specifies the type of file format to write. Formats supported natively by Gelato include: `"tiff"`. Additional image formats may be supported by user-written image output format DSO/DLL's (see Chapter 13). If *format* is NULL or points to the string `"null"`, the output will never be used (thus, an output may be effectively deleted by replacing it and using `"null"` for *format*).

The *data* parameter is a string that indicates what data to output, and may include any of the following:

- `"rgb"`: output a 3-channel file containing color.
- `"rgba"`: output a 4-channel file containing color and alpha (coverage/opacity).
- `"z"`: create an image containing the z depth of the closest surface in each pixel.
- `"avgz"`: create an image containing the average depth of the nearest and second-nearest surface (used for "Woo shadows").
- `"volz"`: create a volume shadow map.[1]
- The type and name of any "global" shader state variable, such as `"normal N"`.
- The type and name of any "output variables" of shaders.

Any pending parameters (set by `Parameter`) give output attributes (summarized below and explained in detail in Section 4.2). `Output` copies the parameter values and clears the pending parameter list (making it safe for the user to subsequently reuse or free the memory referenced by the preceeding `Parameter` calls).

Parameters respected by `Output` include:

[1] Please see Section 10.1.3 for detailed information about generating and using volume shadow maps.

`"string filter"`	The name of the pixel filter to use.
`"float[2] filterwidth"`	The width (in *x* and *y*) of the pixel filter to use.
`"float gain"`	The image gain (1 for none).
`"float gamma"`	The gamma correction (1 for no correction).
`"float dither"`	The dither amplitude (0 for no dither).
`"int[4] quantize"`	The *zero*, *one*, *min* and *max* quantization levels.
`"string stereo:left"` `"string stereo:right"`	Left and right override filenames (stereo mode only)
`"int zchannels"`	The number of *z* channels to save (`"volz"` mode only)
`"float opaquewidth"`	Width for combining opaque *z*'s (`"volz"` mode only)
`"int dynamic"`	If nonzero, indicates a *dynamic* or virtual image (generally a shadow map).
other	Additional data will be passed down to the image output format driver.

Please consult Section 4.2 for detailed explanation of these parameters and their possible values.

Any *parameterlist* tokens other than the ones above will be passed along to the image output format driver. Check the documentation for the specific image driver to see what optional parameters it can take.

NEW!

If the *format* is the ASCII representation of a hexidecimal number (starting with `"0x"`), it will be assumed to be the in-memory address of an `ImageOutput` object. This object will be used directly rather than the usual use of *format* giving the name of a DSO/DLL containing an `ImageOutput`. Please consult Sections 7.3.3 and 13.7 for more information.

EXAMPLE:

```
r->Output ("beauty.tif", "tiff", "rgba", "maincam");
int quant[4] = { 0, 65535, 0, 65535 };
r->Parameter ("int[4] quantize", quant);
r->Output ("specularpass.tif", "tiff", "color Cspec", "maincam");
```

The above commands create two display output streams: (1) a TIFF file `beauty.tif` containing the color and alpha of the image using default filtering and quantization, and (2) a color TIFF file `"specularpass.tif"` containing the `"Cspec"` surface shader output variable, as a 16-bit-per-channel file.

`GelatoAPI::`**World** `()`

`World` marks the end of the section where scene-wide attributes, cameras, and outputs may be set, and the beginning of the section where geometric primitives may be declared. It also resets the CTM to `"world"` space.

If no `Camera` statement was ever encountered, a camera is added to the scene assuming that the current CTM is the world position and the camera's origin was the *original* CTM position (before any transformations were encountered).

GelatoAPI::**Render** ()
GelatoAPI::**Render** (const char **camera*)

Render signals the end of the scene geometry, and triggers final rendering of all the output images. All cameras that have corresponding Output calls will be rendered, and for each camera only objects included in the geometry set that corresponds to the camera will be rendered. Cameras that have no corresponding Output will not produce images, unless *no* Output statement is present at all in the scene, in which case the first camera declared will be automatically given a default Output.

Generally, only one call to Render is allowed, and the only GelatoAPI call valid after Render is GetAttribute. However, when in re-render mode (set by Attribute("int rerender", 1)), multiple Render calls are legal, with certain scene elements allowed to be changed between subsequent re-renders. Details on re-rendering may be found in Section 2.11.

After a call to Render, the graphics state (including CTM) is restored to the way it was immediately following the World call.

Default "live" renderers are *blocking*, that is, the Render call will not return to the caller until all rendering is completed. However, an asynchronous "remote" renderer, created with CreateRenderer("remote"), is *non-blocking*, meaning that Render it will immediately return even as the remote render continues to work. When a non-blocking render is performed, until rendering is complete, the only API calls allowed are GetAttribute and Command; all other API calls will be ignored and will set the appropriate error codes. Only some commands are honored, and only some attributes may be queried during an in-progress render; others will produce errors.

The status of a non-blocking render may be queried with:

```
GetAttribute ("float render:progress", &done)
```

This returns a value between 0.0 and 1.0 indicating how much of the scene is completed rendering. A value of exactly 1.0 indicates that the render is complete (nearly complete renders will not accidentally be rounded up to 1.0 if the render is not yet complete).

It is possible to cause a non-blocking render to abort its render with Command ("return").

GelatoAPI::**Command** (const char *name)

Executes a command signified by the token *name*.

Any pending parameters (set by Parameter) give optional parameters specific to that particular command. Command copies the parameter values and clears the pending parameter list (making it safe for the user to subsequently reuse or free the memory referenced by the preceeding Parameter calls).

Individual renderer implementations conforming to the Gelato API are free to recognize nonstandard commands and are encouraged to use this function as an extension mechanism (much as they do for renderer-specific Attribute tokens).

Gelato recognizes the following commands. Please consult Section 4.5 for detailed explanations of each command and its options.

Command	Functionality
"exit"	Stop rendering (abort frame, exit program).
"return"	Stop rendering (abort frame, return from Render).
"wait"	Wait for render to finish.
"system"	Execute a shell command or run a program.
"bucketpriority"	Change bucket rendering priority.
"fileread"	Alert the renderer that a file was read.
"mkdir"	Make a directory.
"unlink"	Delete a file.

GelatoAPI::**Comment** (const char *format, ...)

For renderer implementations that are creating an archive file, insert a comment using the usual C/C++ printf formatting rules. The archiving renderer implementation, not the caller, is responsible for ensuring that the comment is output with the right syntax to be perceived as a comment in the given format (such as prepending "#" and outputting a linefeed at the end, if it is outputting Pyg).

Comment calls are ignored by "live" renderers that are creating images rather than command archive files.

EXAMPLES:

```
r->Comment ("this is a comment");
```

2.4 Attributes

Attributes are properties of the scene or of objects. Examples of scene attributes include image resolution and camera projection. Examples of object attributes include color, object transformation (position/orientation), and shader assignments.

As it receives scene file commands, the renderer keeps track of the *current attribute state* — that is, the full set of attributes and their values. When a geometric primitive is declared, a copy of the current attribute state is *bound*, or permanently attached, to that geometric primitive. Thus, setting attribute values can affect the appearance of subsequently declared geometry, but does not change previously declared geometry. Because attributes may be changed for each object, it is convenient to save the attribute state, modify attributes and declare geometric primitives, then restore the attribute state to its prior condition.

GelatoAPI::**Attribute** (const char *name, float value)
GelatoAPI::**Attribute** (const char *name, int value)
GelatoAPI::**Attribute** (const char *name, const char *value)
GelatoAPI::**Attribute** (const char *name, const float *value)
GelatoAPI::**Attribute** (const char *name, const int *value)
GelatoAPI::**Attribute** (const char *name, const char **value)
GelatoAPI::**Attribute** (const char *name, const void *value)

GelatoAPI::**Attribute** (const char *name, ParamType type, float value)
GelatoAPI::**Attribute** (const char *name, ParamType type, int value)
GelatoAPI::**Attribute** (const char *name, ParamType type, const char *value)
GelatoAPI::**Attribute** (const char *name, ParamType type, const float *value)
GelatoAPI::**Attribute** (const char *name, ParamType type, const int *value)
GelatoAPI::**Attribute** (const char *name, ParamType type, const char **value)
GelatoAPI::**Attribute** (const char *name, ParamType type, const void *value)

Sets a particular attribute (specified by name as a string) to the data *value*. The *value* may be a float, int, or string (passed as a char *), or a pointer to an array of float, int, or char * values. Data consisting of multiple float values (including color, point, vector, normal , hpoint, or matrix data) should be passed using the float * version. The data type and declaration must match – for example, "float fov" may be called by pasing a float or a float *, but any of the other varieties of Attribute will produce a runtime error. It's also possible to simply pass a raw void * pointing to the data, but in that case there is no compile-time type safety.

Two flavors of Attribute exist: one passes an explicit *type* in the form of a ParamType object, while the other deduces the type from the *name* itself (as described in Section 2.1.5).

If the attribute name begins with the string "user:", the name and value are added to the attribute state even if it is not recognized as a renderer attribute. This value can later be retrieved through the shading language function attribute(). User attributes *must* have their types declared in the attribute name.

Certain attributes apply to the entire scene and cannot vary from object to object. Attempts to set these attributes with Attribute after World will be ignored, and an error

message will be printed. The individual attributes' descriptions will point out which ones cannot be set per-object.

EXAMPLES:

```
// Examples of types declared in the name
r->Attribute ("float fov", 45.0f);        // Pass a float
r->Attribute ("string projection", "perspective");  // string
int res[2] = { 640, 480 };
r->Attribute ("int[2] resolution", res);  // Pass int[2]
float temp = 98.6;
r->Attribute ("float user:temperature", &temp);  // ptr to float

// Examples of passing an explicit type
r->Attribute ("fov", ParamType(PT_FLOAT), 45.0f);
r->Attribute ("projection", ParamType(PT_STRING), "perspective");
```

`bool GelatoAPI::`**GetAttribute** `(const char *name, float &value)`
`bool GelatoAPI::`**GetAttribute** `(const char *name, int &value)`
`bool GelatoAPI::`**GetAttribute** `(const char *name, char * &value)`
`bool GelatoAPI::`**GetAttribute** `(const char *name, float *value)`
`bool GelatoAPI::`**GetAttribute** `(const char *name, int *value)`
`bool GelatoAPI::`**GetAttribute** `(const char *name, char **value)`
`bool GelatoAPI::`**GetAttribute** `(const char *name, void *value)`

Gets the value of a particular attribute (specified by name as a string) in the current attribute state. If the attribute is found, its value is written into the memory pointed to by *value*, and `GetAttribute` returns `true`. If the attribute name is not found or its type does not match the type declaration in *name*, `GetAttribute` returns `false` and the data in *value* is not altered. It is up to the user to ensure that *value* points to a big enough area for the data type requested.

EXAMPLE:

```
int resolution[2];
bool found = r->GetAttribute ("resolution", resolution);
```

`GelatoAPI::`**PushAttributes** `()`
`GelatoAPI::`**PopAttributes** `()`

Save and restore the per-object attribute state, including the current transformation (see `PushTransform` and `PopTransform`, section 2.5). Upon `PopAttributes`, the current attribute set is replaced by the attribute set that was in effect at the corresponding `PushAttributes`. It is perfectly legal to *nest* blocks delimited by `PushAttributes`/ `PopAttributes`.

EXAMPLE:

```
r->PushAttributes()
r->PopAttributes()
```

`GelatoAPI::`**SaveAttributes** `(const char *`*`name`*`, const char *attrs = NULL)`

Create a named alias for part or all of the current per-object attribute state in a global dictionary of name/attribute state pairs. The *name* may be used with `RestoreAttributes`.

The optional string *attrs* is a comma-separated list of which attributes should be saved (passing NULL indicates that all attributes should be saved). Those attributes may be any of the named attributes described throughout this section, and also recognize the following special names:

`"transform"`	The current transformation
`"shaders"`	All shader assignments, except for lights
`"surface"`	Just the surface shader assignment
`"displacement"`	Just the displacement shader assignment
`"volume"`	Just the volume shader assignment
`"lights"`	The active light list
`"trimcurve"`	The trim curve for `Patch` primitives.
`"user"`	All of the `user:` attributes

EXAMPLE:

```
SaveAttributes ("leftarm");
SaveAttributes ("creature_shaders", "shaders,C,opacity");
```

`GelatoAPI::`**RestoreAttributes** `(const char *`*`name`*`, const char *attrs = NULL)`

Replaces some or all of the current per-object attribute state with the saved attribute state with the given *name* (set by `SaveAttributes`).

The optional string *attrs* is a comma-separated list of which attributes, out of those saved by the corresponding `SaveAttributes`, should be restored (passing NULL indicates that all saved attributes should be restored). The meanings of the attribute names are as described by the documentation for `SaveAttributes`.

EXAMPLE:

```
RestoreAttributes ("leftarm", "C");
RestoreAttributes ("creature_shaders");
```

`GelatoAPI::`**Modify** `(const char *`*`namepattern`*`=NULL)`

The `Modify` call sets up a mode in which all attribute states whose `"name"` attribute matches the regular expression *namepattern* are subject to modification by subsequent API calls. In modify mode, these changes will alter all primitives that are bound to the attribute states found. In contrast, when not in modify mode, attribute changes only affect subsequently-declared geometry and cannot change previously-declared primitives (i.e., the attribute state usually has a “copy-on-write” semantic, but does not when in modify mode).

The set of attribute states that may be found to match the pattern are all those that were active at the time of declaration of any shader, light, or geometric primitive that is still

extant (not yet culled or discarded), plus all attribute states marked by `SaveAttributes`. The matching of *namepattern* versus the `"name"` attribute is per the usual POSIX `regex` Basic Regular Expression rules. If *namepattern* is `""` (the empty string), NULL, or is not supplied at all, all extant attribute states will match. In the case of attribute states saved by `SaveAttributes`, the pattern match is still performed on the `"name"` field of the attribute state iteslf, not on the lookup key passed to `SaveAttributes`.

A second call to `Modify` may change the set of attribute states that are undergoing modification. When in modify mode, `PopAttributes` will exit modify mode, returning to the usual copy-on-write semantics of attribute changes.

When in modify mode, there is still a current attribute state, but it will not undergo changes in modify mode unless its name matches *namepattern*. Some API calls only apply to the current attribute state:

- `GetAttribute` and `SaveAttributes` apply to the current attribute state, not the list of modifying states (since they must get from or save just one attribute state, it's nonsensical to apply these to the whole list).
- `PushAttributes`, `PushTransform`, and `PopTransform` apply to the current attribute state, although attribute or transformation changes within a push/pop sequence will still apply to the entire list as long as modify mode is active.
- When in modify mode, any geometric primitives added will inherit the current attribute state. They will continue to be modified by subsequent attribute changes only if their name matched *namepattern*.

Modify mode ends and the renderer returns to the usual copy-on-write semantics for attribute changes upon a call to `World`, `Render`, or the `PopAttributes` corresponding to the push/pop surrounding the `Modify` call.

Note that whether the renderer is in modify mode is a global property of the renderer, not a part of the hierarchical graphics state. Thus, you should not expect a sequence of push/pop operations to be alternating back and forth between modify mode or copy-on-write.

EXAMPLE:

```
r->Light ("rim", "pointlight")
r->Attribute ("string name", "obj1")
... geometry for object 1 ...
r->Attribute ("string name", "obj2")
... geometry for object 2 ...

r->PushAttributes ()
r->Modify ("obj2")
r->LightSwitch ("rim", 1)
r->PopAttributes ()
```

The above example creates a light source, makes two objects (implicitly illuminating both objects with the light), then modifies all attribute states referenced by the second object by turning the light off for those states.

Scene-wide Attribute	Meaning (default value)
"string pass"	Name of the rendering "pass" ("").
"string units:length"	Physical length units of "common" space ("").
"float units:lengthscale"	Length unit scale of "common" space units (1).
"string units:time"	Time units of Motion and shutter times ("").
"float units:timescale"	Time unit scale (1).
"float units:fps"	Frames per second (24).
"int rerender"	Put the renderer in "re-render" mode (0)
"int rerender:memory"	Size of re-render caches, in KB (0)
"string rerender:filepattern"	Files to monitor for re-render updates ("")
"int rerender:reshaderays"	Re-shade rays when they hit a changed object? (1)
"int limits:gridsize"	Number of points to try to shade at once (256)
"int limits:texturememory"	Maximum texture cache size, in KB (20480)
"int limits:texturefiles"	Maximum number of open texture file handles (1000 Linux, 100 Windows)
"float trimcurve:quality"	Quality scale on trim curve approximation (1)
"int limits:trimcurvememory"	Maximum trim cache size, in KB (10240)
"int limits:transparentlayers"	Max number of transparent layers (-1)
"color limits:opacitythreshold"	The opacity that is considered fully opaque for culling purposes (1, 1, 1)
"color shadow:opacitythreshold"	The opacity below which transparent objects will not appear in shadow depth maps (1, 1, 1)
"int limits:transparentgrids"	Batch size for processing transparent grids (32)
"int ray:maxdepth"	Maximum ray recursion depth (2)
"color ray:maxdepthcolor"	Color of rays beyond the max recursion depth (0,0,0)
"string dice:fixeduname"	Name of *u* parameter metric for fixed dicing ("u")
"string dice:fixedvname"	Name of *v* parameter metric for fixed dicing ("v")
"string path:input"	Search path for scene files (".:$GELATOHOME/inputs")
"string path:texture"	Search path for texture files (".:$GELATOHOME/textures")
"string path:shader"	Search path for compiled shaders (".:$GELATOHOME/shaders")
"string path:generator"	Search path for generator DSO's (".:$GELATOHOME/lib")
"string path:imageio"	Search path for image format input/output DSO's (".:$GELATOHOME/lib")

...continued on next page...

Scene-wide Attribute	Meaning (default value)
...continued from previous page...	
`"string spatialdb:read"`	Loads the named spatial database.
`"string spatialdb:readonly"`	Loads the named spatial database and always uses it (not just when values are close).
`"string spatialdb:write"`	Marks the named spatial database to be saved after rendering is completed.
`"string spatialdb:writeonly"`	Marks the named spatial database to be saved, guarantees to reads.
`"string texture:substtexture"`	Name of substitute texture (`""`).
`"string texture:substenv"`	Name of substitute environment map (`""`).
`"int texture:automipmap"`	Automatically generate mipmaps? (1)
`"int verbosity"`	Verbosity of status messages echoed to console (`1`)
`"string error:filename"`	Error log file instead of `stderr` (`""`)
`"int statistics:level"`	Statistics detail level (`0`)
`"string statistics:filename"`	Statistics log file instead of `stderr` (`""`)
`"int debug:dso"`	Echo debugging info about DSO loading (`0`)
`"string debug:filesread"`	Filename for list of all files read during rendering (`""`)
`"int debug:shadernan"`	Enable detection of NaN's in your shaders (`0`)
`"string griddump:filename"`	File to save all shaded grids (`""`)
`"int griddump:binary"`	Grid dump uses binary format if nonzero (0)
`"string renderer:name"`	The brand name of the renderer (e.g., `"NVIDIA Gelato"`). [N.B. – Read only.]
`"int[4] renderer:version"`	Major, minor, release, and patch numbers (e.g., `{ 2, 1, 0, 0 }`). [N.B. – Read only.]
`"string renderer:versionstring"`	The release numbers expressed as a string (e.g., `"2.1.0.0"`). [N.B. – Read only.]
`"float render:progress"`	Completed portion of the rendering (1.0 indicates completed). [N.B. – Read only.]
`"int render:success"`	Did the last render finish without aborting? [N.B. – Read only.]
`"string net:workers"`	List of network-parallel rendering workers (`""`).
`"string net:workercmd"`	Command to run on net workers (`""`).
`"int net:tint"`	If nonzero, tint buckets from different net workers (0).
`"int pyg:binary"`	Output Pyg files in binary format (0)
`"int pyg:separateparams"`	Output Pyg parameters separately rather than inline (0)
`"int pyg:indent"`	Change Pyg output indentation level (0)
`"int limits:inputlevels"`	How many recursive Generator levels should be output as Pyg (0)

Per-Object Attribute	Meaning (default value)
"color C"	Default surface base color (1, 1, 1)
"color opacity"	Default surface opacity (1, 1, 1)
"int holdout"	Holdout matte mode (0)
"string orientation"	Which way the normals face ("outside")
"int twosided"	Is the object visible from both sides? (1)
"float shadingquality"	Shading quality – roughly shades per pixel (1)
"string name"	Object name, used mainly for clear error reporting ("")
"string geometryset"	Name of active geometry sets ("camera")
"matrix transform"	CTM at shutter open time. (read only)
"int dice:binary"	Round tessellation rates to powers of two? (0)
"float dice:curvature"	Angle (in degrees) that force tessellations (10)
"float dice:highcurvature"	Angle (in degrees) for sharp corners (120)
"float dice:motionfactor"	Scaling for decreased tessellation and shading for moving objects (1)
"int dice:keepcreases"	Does any tangent discontinuity force tessellation? (0)
"int dice:rasterorient"	Take orientation into account for tessellation? (1)
"float[2] dice:fixed"	Fixed parametric tessellation rate (0, 0)
"int cull:occlusion"	Should objects be occlusion culled? (1)
"string shading:view"	Meaning of I for camera grids ("camera").
"string trimcurve:sense"	Whether to discard the inside or outside of trim curves on Patch primitives ("inside")
"float trimcurve:curvature"	curvature threshold for trim curve shape (10)
"int light:nsamples"	Number of area light samples (1)
"float displace:maxradius"	Maximum radial displacement (0)
"string displace:maxspace"	Coordinate system in which max radial displacement is measured ("common")
"float shadow:bias"	Default bias for ray and mapped shadows (0.01)
"int ray:opaqueshadows"	Are ray-traced objects opaque regardless of their shaders (1)
"int ray:motion"	Do rays consider motion-blur of objects? (0)
"int ray:displace"	Truly displace objects when ray tracing? (1)
"float indirect:maxerror"	Error metric for interpolation of indirect light (0.25)
"float indirect:maxpixeldist"	Max distance for interpolation of indirect light (20)
"int indirect:minsamples"	Minimum number of nearby indirect samples to interpolate (3)
"string indirect:spatialdb"	Name of default indirect spatial database ("indirect.sdb")

...continued on next page...

Per-Object Attribute	Meaning (default value)
...continued from previous page...	
"float occlusion:maxerror"	Error metric for interpolation of occlusion (0.25)
"float occlusion:maxpixeldist"	Max distance for interpolation of occlusion (20)
"int occlusion:minsamples"	Minimum number of nearby occlusion samples to interpolate (3)
"string occlusion:spatialdb"	Name of default occlusion spatial database ("occlusion.sdb")
"int rerender:locked"	Do not reshade objects when re-rendering (0)
"int rerender:cachelights"	Cache light when re-rendering (1)

2.5 Transformations

As it receives scene commands, the renderer keeps track of the *current transformation* (sometimes called the *CTM* for "current transformation matrix"). When a geometric primitive is declared, a copy of the CTM is permanently attached to that primitive, which will consider the CTM as its `"object"` space. Therefore, any spatial quantities (such as vertex positions `"P"`) passed on the primitive are relative the CTM that was in effect at the time that the geometric primitive command was encountered. Thus, transformation commands affect the appearance of subsequently declared geometry, but do not change previously declared geometry (unless modify mode is invoked, see Section 2.4).

A number of named coordinate systems are predefined, or implicitly defined by API methods such as `World` and `Camera`. The predefined coordinate systems are:

`"world"` The coordinate system active at `World`.

`"camera"` The coordinate system with its origin at the center of the camera lens, x-axis pointing right, y-axis pointing up, and z-axis pointing into the screen.

`"screen"` The coordinate system of the camera's image plane (after perspective projection, if one is specified). Coordinate (0,0) in the center of the screen.

`"raster"` 2D pixel coordinates. The upper left corner of the image in `"raster"` space is (0,0), and the lower right corner is $(xres, yres)$.

`"NDC"` 2D Normalized Device Coordinates. The upper left corner of the image in `"NDC"` space is (0,0), and the lower right corner is (1,1).

It is convenient to save the transformation state, modify the transformation and declare geometric primitives, then restore the transformation to its prior condition. A command is provided to perform this action:

`GelatoAPI::`**PushTransform** `()`
`GelatoAPI::`**PopTransform** `()`

> Save and restore the current transformation. Upon `PopTransform`, the current transformation is set to the transformation that was in effect at the corresponding `PushTransform`. It is perfectly legal to *nest* blocks delimited by `PushTransform`/ `PopTransform`.

Remember that the current transformation is actually part of the attribute state, therefore the CTM is also saved and restored (along with the rest of the attribute state) by `PushAttributes` and `PopAttributes`.

A variety of commands are available to replace or modify the current transformation. The two most fundamental (and upon which all others are based) are `SetTransform` and `AppendTransform`.

`GelatoAPI::`**SetTransform** `(const float *M)`
`GelatoAPI::`**SetTransform** `(const char *`*`name`*`)`

> Replace the current transformation with the 4x4 matrix supplied. If the `SetTransform` routine is called before `World`, then M is assumed to be relative to `"camera"` space, whereas a call to `SetTransform` after `World` assumes that M is relative to `"world"` space.

If a string is passed rather than a matrix, replace the current transformation with the named transformation, which may either be the name of a saved attribute state defined by `SaveAttributes` (which must, of course, have saved the transformation), or the name of a standard coordinate system ("world", "camera", "screen", "raster", "NDC").

EXAMPLES:

```
Matrix4 M(1, 0, 0, 0,  0, 1, 0, 0,  0, 0, 1, 0,  3, 0, 0, 1);
r->SetTransform ((const float *)&M)

r->SetTransform ("world");
```

`GelatoAPI::`**AppendTransform** `(const float *M)`

Concatenate the given 4x4 transformation matrix onto the current transformation.

EXAMPLE:

```
Matrix4 M(1, 0, 0, 0,  0, 1, 0, 0,  0, 0, 1, 0,  3, 0, 0, 1);
r->AppendTransform ((const float *)&M);
```

(This example assumes that the `Matrix4` type consists of 16 contiguous `float` variables, and so can be safely cast to a `float *`.)

The `SetTransform` and `AppendTransform` routines, which replace and concatenate the CTM, respectively, are fully general. For several of the most useful and common transformations, there are specific routines that have a more compact, simpler syntax:

`GelatoAPI::`**Translate** `(float x, float y, float z)`

Prepend the current transformation with the given translation. This is identical to the call:

```
Matrix4 M(1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, x, y, z, 1);
r->AppendTransform ((float *)&M)
```

EXAMPLE:

```
r->Translate (2, 0, 0);
```

`GelatoAPI::`**Rotate** `(float angle, float x, float y, float z)`

Prepend the current transformation with a rotation of *angle* degrees about the axis defined by (x,y,z).

EXAMPLE:

```
r->Rotate (30, 0, 0, 1);
```

`GelatoAPI::`**Scale** `(float sx, float sy, float sz)`

Prepend the current transformation with a scale factor of (*sx, sy, sz*). This is idenical to the call:

```
Matrix4 M(sx, 0, 0, 0, 0, sy, 0, 0, 0, 0, sz, 0, 0, 0, 0, 1);
r->AppendTransform ((float *)&M)
```

EXAMPLE:

```
r->Scale (sx, sy, sz);
```

2.6 Shaders and Lights

GelatoAPI::**Shader** (const char *shaderusage, const char *shadername,
const char *layername=NULL)
GelatoAPI::**Shader** (const char *shaderusage)

Sets the shader specified by *shadername* to be used as the current shader of the given *shaderusage*. Valid tokens for *shaderusage* include "surface", "displacement", or "volume", and must match the shader type that was declared in the shader source code. A generic shader (whose type is shader) may be instanced for any shader usage, provided it only performs operations legal for the given usage. Implementations of the Gelato API are free to support additional shader types and usages. A shader of a type or usage not supported by a particular implementation will be ignored.

If only the *shaderusage* is given (with no shader name), that shader usage will be cleared (i.e., no shader will be assigned for that usage).

The optional *layername*, if specified, gives a name to the shader layer (for later use with ConnectShaders). See ShaderGroupBegin for information on how to specify more than one shader of each type.

Any pending parameters (set by Parameter) give parameters specific to the particular shader being used. Shader copies the parameter values and clears the pending parameter list (making it safe for the user to subsequently reuse or free the memory referenced by the preceeding Parameter calls).

EXAMPLE:

```
r->Parameter ("float Kd", Kd);
r->Shader ("surface", "plastic");
```

GelatoAPI::**ShaderGroupBegin** ()
GelatoAPI::**ShaderGroupEnd** ()

GelatoAPI::**ConnectShaders** (const char *srclayer, const char *srcparam,
const char *dstlayer, const char *dstparam)

ShaderGroupBegin and ShaderGroupEnd denote that repeated calls to Shader within the block should create a group of shaders of the same type which will be executed in sequence on subsequent geometry. This lets you, for example, bind several "surface" shaders at once. Any layer names passed toShader are valid only within the enclosing ShaderGroup block (and thus, calls to ConnectShaders must be inside the same ShaderGroup block).

ConnectShaders connects two shaders such that the source shader's parameter named by *srcparam* will be used as input for the destination shader's *dstparam* (overriding any shader defaults or other bindings for that parameter). *srclayer* and *dstlayer* both refer to shaders on the object, referenced by the layer names passed to calls to Shader. The *srcparam* either be the name of a parameter of *srclayer* (presumably, but not required to be, an output parameter) or the name of a global variable (such as P, N, C, etc.), and

dstparam must be either the name of a parameter to *dstlayer* or the name of a global variable.

The source and destination parameters are presumed to be of the same data type, and if they are arrays, they must have the same array length. Type conversions are fairly flexible for non-arrays, though; in particular, any triple (color point, normal, vector) may connect to another triple, a float may connect to a triple (replicating the value for all three components), and a triple may connect to a float (passing just the first component). Failure of the names to be found or of the parameter data types to be compatible will result in an error message, and no connection being made.

It is possible to connect a single channel of a triple to a scalar type, or a scalar to one channel of a triple, or one channel of a triple to a channel of a triple. It is also possible to connect an individual array element to a scalar type, or a scalar to an array element, or an array element to another array element (in which case the two arrays need not be the same size). The element data types must still be compatible with each other, but like scalars, triple-to-triple, float-to-triple and triple-to-float conversions are done automatically).

These routines may be used to create a layered light source, but in that case it would be `Light` calls inside the `ShaderGroupBegin`/ `ShaderGroupEnd` block, and all of the `Light` calls are expected to have the same *lightid*. (See the description of `Light`, below, for details.)

EXAMPLE:

```
r->ShaderGroupBegin ();
  r->Parameter ("string texturename", txname);
  r->Shader ("surface", "texmap", "layer1");
  r->Parameter ("float Kd", Kd);
  r->Shader ("surface", "plastic", "layer2");
  r->ConnectShaders ("layer1", "out", "layer2", "Ks");
r->ShaderGroupEnd ();
```

Example of automatic type conversion:

```
r->ConnectShaders ("layer1", "floatvar", "layer2", "colorvar");
```

Examples of array element and color channel connections:

```
r->ConnectShaders ("layer1", "floatarrayvar[3]", "layer2", "floatvar");
r->ConnectShaders ("layer1", "colorvar", "layer2", "colorarrayvar[1]");
r->ConnectShaders ("layer1", "floatvar", "layer2", "colorvar[1]");
r->ConnectShaders ("layer1", "colorvar[1]", "layer2", "floatvar");
```

`GelatoAPI::`**Light** `(const char *lightid, const char *shadername, const char *layername=NULL)`

Makes a new light source or replaces an existing light source. The light source is also added to the list of active light sources in the attribute state (that is, it will illuminate subsequent geometry). The light source (or layer) will use the light shader specified by *shadername*, and its `"shader"` space will be the CTM at the time of the `Light` call.

lightid parameter is a unique identifier for the light. If *lightid* is the same as that of an existing light, the old definition of that light will be replaced by the new definition of the light. If *shadername* is NULL or points to the string `"null"`, the light will never be used, effectively removing it from the scene.

When within a `ShaderGroupBegin` / `ShaderGroupEnd` block, `Light` will append a layer to the light being specified. All layers of the light must use the same *lightid*. The optional *layername*, if specified, gives a name to the layer for later use with `ConnectShaders`.

Any pending parameters (set by `Parameter`) give parameters specific to the particular shader being used. `Light` copies the parameter values and clears the pending parameter list (making it safe for the user to subsequently reuse or free the memory referenced by the preceeding `Parameter` calls).

EXAMPLES:

```
float lightcolor[3] = { .9, 1, .7 };
r->Parameter ("color lightcolor", lc);
r->Light ("fill1", "pointlight");

r->Parameter ("float intensity", 10.0f);
r->Parameter ("string geometry", "myareageom");
r->Light ("key", "uberlight");
```

`GelatoAPI::`**LightSwitch** `(const char *lightid, bool onoff)`

Add or remove a light from the active light list. The *lightid* is the identifier passed to a previous call to `Light`. If *onoff* is `false`, the light is removed from the active light list, and therefore does not shine on subsequently declared geometry. If *onoff* is `true`, the light is added to the active light list (if it not already active), and therefore will shine on subsequently declared geometry.

EXAMPLE:

```
r->Light ("key", "spotlight");
...
r->LightSwitch ("key", false);
r->Patch (...);   // key light will not shine on this patch
```

2.7 Motion Blur

Geometry and transformations may be motion-blurred over the course of a rendered frame to simulate how a real camera captures light over a finite interval rather than instantaneously. The shutter interval of the camera (i.e., the time range over which light exposes the image) is specified to the `Camera` call (see Section 2.3) as the optional the `"float[2] shutter"` Parameter.

Both geometric primitives and transformations may be blurred (i.e., described in a changing way over time) using the `Motion` method described below. The camera shutter interval need not be identical to the times given to `Motion` calls, and multiple `Motion` calls that apply to the same object need not have identical time values.

Motion blur will only take effect if a camera shutter interval is specified. If the camera is not given a `"shutter"` parameter, all objects will be drawn in their configuration at the earliest time of their motion descriptions.

`GelatoAPI::`**Motion** `(int ntimes, float time0, ...)`
`GelatoAPI::`**Motion** `(int ntimes, const float *times)`

`Motion` marks the start of a *motion block* and specifies (either with a variable parameter list or an array) *ntimes* time values. It is expected that the `Motion` call is immediately followed by (1) exactly *ntimes* transformation calls (e.g., `Translate`, `AppendTransform`, etc.) of the same name but with different numerical parameters; or (2) *ntimes* geometric primitives (including `Parameter` calls, if necessary) that differ in the values of their parameters. Each of the *ntimes* transformations or primitives corresponds to the position or shape at the respective time passed to the `Motion` call.

Motion-blurred transformations will be linearly interpolated over each motion segment. At times before the first motion time, the transformation will be identical to that of the first motion time; at times after the last motion time, the transformation will be identical to that of the last motion time. That is, transformations simply do not move outside their specified time intervals.

Motion-blurred geometric primitives also linearly interpolate their control vertices or parameters over each motion segment. Primitives *do not exist* outside the range of their motion segments. Thus, objects whose motion descriptions only partially overlap the camera's shutter interval may indeed only be imaged for part of the shutter interval.

For motion-blurred geometric primitives, all *ntimes* primitives must be the same primitive type and the same "shape." That is, a primitive may not "morph" between two types (say, from a `Mesh` into a `Patch`), nor may it change the list of parameters or their sizes (e.g., the primitive must have the same number of control vertices at each time). Only the *numerical values* of the parameters may change over time.

EXAMPLE:

```
r->Motion (2, 0.0, 1.0/48.0);
r->Translate (0, 0, 0);
r->Translate (0, 0, 1);
```

The above example makes a single translation, but with two time values, so that the transformation translates one unit in z over the time interval between $t = 0$ and $t = 1/48$.

2.8 Geometric Primitives

The renderer supports 0-, 1-, and 2D geometric primitives. The 0D primitives are points, useful for particle systems. The 1D primitives are line or curve segments, useful for hair. The 2D primitives are broken down into patches and meshes.

Primitives inherit current attributes

Geometric primitives, when declared, inherit the current attribute state (including transformations and shader assignments). Therefore, all spatial (point, normal, vector, matrix) data passed to the primitive, including control vertices, are expressed in the object coordinate system, and will be transformed to a common space by the renderer. Subsequent changes to attributes do not affect previously-declared primitives (unless modify mode is invoked, see Section 2.4).

Control vertices and primitive variables

With the exception of `Sphere`, all other primitives have their shapes specified by a series of *control vertices*. The actual shape of the object is defined by some kind of interpolation of the control vertices. Control vertices are passed via `Parameter` as the variable `"vertex point P"`. Some primitives allow *rational* (4-D homogeneous) points, which are passed as the variable `"vertex hpoint Pw"`. All primitives defined by control vertices must have either `"P"` or `"Pw"` (but not both).

A primitive may have use `Parameter` to override the default surface color or surface normal by attaching the variables `"color C"` or `"normal N"`, respectively. If no `"C"` is attached, the default surface color will be the value of the `"C"` Attribute (see Section 4.4.4). It is probably not useful to override the default normal `"N"` for truly curved-surface primitives (such as `Patch` or `Mesh("catmull-clark")` but is frequently used when using faceted geometry such as `Mesh("linear")` to provide smoothed normals. .

Other *primitive variables* may be attached to geometric primitives via `Parameter`. These are automatically interpolated, according to the interpolation type, and passed along to become the values of any identically-named parameters of any of the surface, displacement, or volume shaders attached to the primitive.

Primitive variables have several choices of interpolation type. All primitives support interpolation type `vertex`, which requires the same number of values as the control vertex positions (`"P"` or `"Pw"`) and is interpolated in the same manner as the positions; and `linear`, which is linearly (or bilinearly) interpolated across the primitive, and therefore may have a different number of values than the control vertices. Some primitives also support the interpolation type `perpiece`, which supplies one value for each section of the primitive (one per face for a `Mesh`, one per curve for a `Curves`).

2.8.1 Patches

`GelatoAPI::`**Patch** `(const char *interp, int nu, int nv)`

Specifies a rectangular array of $nu \times nv$ control vertices forming a mesh that is piecewise linear or cubic. In this simple form, the mesh is parameterized uniformly on $[0,1]$

using the specified interpolation type: `"linear"` for piecewise linear, or one of several piecewise cubic types, including `"bezier"`, `"bspline"`, and `"catmull-rom"`. Different interpolation types may be used in each direction by specifying the *u* and *v* interpolations separated by a comma (e.g., `"bspline,linear"` for a patch mesh that is a piecewise cubic B-spline in *u* and piecewise bilinear in *v*). It is important for *nu* and *nv* to be appropriate for the interpolation type (e.g., ≥ 2 for linear, ≥ 4 for cubic, $4+3i$ for Bezier).

Pending parameters (set by `Parameter`) give control vertices and primitive variables to be interpolated across the surface. The parameters must include either 3D control vertices (passed as the `"P"` parameter), or 4D `hpoint` control vertices (`"Pw"`) to indicate a rational patch.

Primitive variables with interpolation type `vertex` require $nu \times nv$ values (i.e., the same number of values as `"P"` or `"Pw"`; `linear` primitive variables require 4 values, which are interpolated bilinearly across the patch; and primitive variables without an interpolation type require a single data value which does not vary across the patch.

`Patch` copies the parameter values and clears the pending parameter list (making it safe for the user to subsequently reuse or free the memory referenced by the preceeding `Parameter` calls).

EXAMPLE:

```
float P[] = { 1, 0, 0,   1, 1, 0,   0, 1, 0,  -1, 1, 0,  -1, 0, 0,
             -1, -1, 0,  0, -1, 0,  1, -1, 0,  1, 0, 0,   1, 0, -3,
              1, 1, -3,  0, 1, -3, -1, 1, -3, -1, 0, -3, -1, -1, -3,
              0, -1, -3, 1, -1, -3, 1, 0, -3 };
r->Parameter ("P", ParamType(PT_POINT,INTERP_VERTEX), P);
r->Patch ("bspline,linear", 9, 2);
```

`GelatoAPI::`**Patch** `(int nu, int uorder, const float *uknot, float umin, float umax, int nv, int vorder, const float *vknot, float vmin, float vmax)`

Specify a rectangular array of $nu \times nv$ control vertices forming a NURBS mesh. In this more complex form, the user may explicitly specify *u* and *v* orders, knot vectors, and parameter subranges. The length of *uknot* must be $nu + uorder$ and the length of *vknot* must be $nv + vorder$. The surface is defined over the parametric range $[umin \ldots umax, vmin \ldots vmax]$.

Pending parameters (set by `Parameter`) give control vertices and primitive variables to be interpolated across the surface. The parameters must include either 3D control vertices (passed as the `"P"` parameter), or 4D `hpoint` control vertices (`"Pw"`) to indicate a rational patch.

Primitive variables with interpolation type `vertex` require $nu \times nv$ values; `linear` primitive variables require 4 values, which are interpolated bilinearly across the patch; and primitive variables without an interpolation type require a single data value which does not vary across the patch.

`Patch` copies the parameter values and clears the pending parameter list (making it safe for the user to subsequently reuse or free the memory referenced by the preceeding `Parameter` calls).

EXAMPLE:

```
float uknot[] = { 0, 0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 4 };
float vknot[] = { 0, 0, 1, 1 };
float Pw[] = { 1, 0, 0, 1,   1, 1, 0, 1,   0, 2, 0, 2,  -1, 1, 0, 1,
              -1, 0, 0, 1,  -1, -1, 0, 1,  0, -2, 0, 2,  1, -1, 0, 1,
               1, 0, 0, 1,   1, 0, -3, 1,  1, 1, -3, 1,  0, 2, -6, 2,
              -1, 1, -3, 1, -1, 0, -3, 1, -1, -1, -3, 1, 0, -2, -6, 2,
              1, -1, -3, 1,  1, 0, -3, 1 };
rj->Parameter ("vertex hpoint Pw", Pw);
r->Patch (9, 3, uknot, 0, 4, 2, 2, vknot, 0, 1);
```

```
GelatoAPI::TrimCurve (int nloops, const int *ncurves,
                      const int *n, const int *order, const float *knot,
                      const float *min, const float *max, const float *uvw)
```

Trim curves define regions in a `Patch`'s parametric space that will be removed (or will be kept, with the remainder removed, depending on the setting of the `"trimcurve:sense"` attribute; see Section 4.4.8).

Sets the current *trim curve* that will apply to subsequently defined `Patch` primitives. This API call actually sets an attribute, and thus the trim curve is saved and restored with the `PushAttributes`, `PopAttributes`, `SaveAttributes`, and `RestoreAttributes` routines. Trimming will be turned off for subsequent geometry if *nloops* is zero (in which case the values of the other arguments are unused).

Trim curves define regions in a `Patch`'s parametric space that will be removed. The region is defined by *nloops* closed loops, each of which consists of *ncurves*[*i*] 2D nonuniform rational curve segments (i.e. NURBS curves). Thus, the total number of trim curve segments is the sum of all elements in *ncurves*. In respective order of the loops and curve segments, *n*[*j*] is the number of rational control points, *order*[*j*] is the order of the curve segment, *min*[*j*] and *max*[*j*] are the minimum and maximum parametric values of the curve segment. The *knots* array contains the knot vectors for all the curve segments (with each segment j having $n[j]+order[j]$ knot values). The *uvw* array contains the control points (with each segment j having $n[j]$ control points). The control points themselves each consist of three floating-point numbers giving a *rational* (u,v,w) 2D position in parametric space of the patch.

The symmetry between the specification of trim curves as NURBS curves and the specification of `Patch` primitives as NURBS surfaces should be apparent.

EXAMPLE:

```
int ncurves[] = { 1 };
int n[] = { 9 };
int order[] = { 3 };
float uknot[] = { 0, 0, 0, .25, .25, .5, .5, .75, .75, 1, 1, 1 };
float min[] = { 0 };
float max[] = { 1 };
float uvw[] = { 1, .5, 1,  1, 1, 1,  1, 2, 2,  0, 1, 1,
```

```
                0, .5, 1,  0, 0, 1,  1, 0, 2,  1, 0, 1,  1, .5, 1 }

    r->TrimCurve (1, ncurves, n, order, knots, min, max, uvw);
```

2.8.2 Meshes

GelatoAPI::**Mesh** (const char *interp, int nfaces,
const int *nverts, const int *verts)

Specifies a connected mesh of faces. A mesh is not constrained to have rectangular connectivity — each face may have any number of vertices (3 or more) and any number of faces may share a vertex.

The *interp* parameter may be "linear" to indicate a linear interpolation (i.e., a flat polygon mesh), or "catmull-clark" to indicate Catmull-Clark subdivision should be used to smooth the mesh. Other interpolation schemes may be added in the future.

The *nfaces* parameter is the number of faces in the mesh. The array $nverts[0..nfaces-1]$ contains the number of vertices in each face. The array *verts*, whose length is the sum of all the entries in *nverts*[], contains the vertex indices, in order, of all faces.

Pending parameters (set by Parameter) give control vertices, primitive variables to be interpolated across the surface, or other parameters that control mesh behavior. The parameters must include 3D control vertices (passed as the "P" parameter); other parameters are optional.

Any vertex primitive variable must have a number of entries high enough to accommodate the highest vertex index in *verts*[]. All linear primitive variables must have the same number of data items as the length of the *verts* array, and are in the same order as the vertex indices themselves. Mesh allows perpiece variables, which have a number of data items equal to *nfaces*, one per face. As usual, primitive variables without an interpolation type supply only one value for the entire mesh.

Several special parameters control behavior and shape of the mesh, rather than merely providing user data that will be interpolated:

"int border"

For meshes using "catmull-clark" interpolation, this parameter controls the behavior of the surface on faces abutting the boundary of an open mesh:

0 (default) The usual Catmull-Clark rules will apply, in which border faces are not actually drawn.

1 Border faces are drawn, with the boundary edge interpolating to the B-spline formed by the edge vertices, but with sharp corners (each edge is a separate B-spline, not continuous across corner vertices).

2 Border faces are drawn, with the boundary edge interpolating to the B-spline formed by the edge vertices, with round corners (the edge interpolates as one continuous B-spline, even around the corner vertices).

Meshes using "linear" interpolation are not affected by the "border" parameter.

"int[N] holes"

Indicates that the array of *N* integers supplies a list of faces (indexed beginning with 0) in the mesh that are holes that will not generate geometry to be drawn.

"perpiece string __attributes"

NEW!

Indicates the attribute states (previously named with SaveAttributes) to apply to each face. This allows the specification of a single mesh that has different attributes on a per-face basis, such as having multiple sets of faces each of which has a different surface shader. If this optional parameter is not specified, all faces will inherit the default attribute state of the primitive (that is, the attributes locally active at the time of the Mesh statement).

"string[N] __attributes"
"perpiece int __attributesindex"

NEW!

The functionality of these two parameters is identical to that of "perpiece string __attributes", but allows an alternate specification: an array of attribute names, and a per-face integer index into that array. This may be a more convenient or more compact representation for some applications. It is an error for any face to specify an index that is greater than or equal to the length of the __attributes array.

These special primitive variables control the behavior of the subdivision mesh, and are not passed down to the shader parameters as are ordinary user-defined primitive variables.

Mesh copies the parameter values and clears the pending parameter list (making it safe for the user to subsequently reuse or free the memory referenced by the preceeding Parameter calls).

EXAMPLE:

```
int nverts[] = { 4, 4, 4, 4, 4, 4, 4, 4, 4 };
int verts[] = { 0, 1, 5, 4,    1, 2, 6, 5,     2, 3, 7, 6,
                4, 5, 9, 8,    5, 6, 10, 9,    6, 7, 11, 10,
                8, 9, 13, 12,  9, 10, 14, 13,  10, 11, 15, 14 };
float points[] = { -3, 0, -3,  -1, 0, -3,  1, 0, -3,  3, 0, -3,
                   -3, 0, -1,  -1, 0, -1,  1, 0, -1,  3, 0, -1,
                   -3, 0, 1,   -1, 0, 1,   1, 0, 1,   3, 0, 1,
                   -3, 0, 3,   -1, 0, 3,   1, 0, 3,   3, 0, 3 };
r->Parameter ("vertex point P", (float *)&points);
r->Mesh ("catmull-clark", 9, nverts, verts);
```

2.8.3 Point and Curve Primitives

`GelatoAPI::`**Points** `(int npoints)`

`Points` primitives are for particle systems. *npoints* is the number of distinct point particles.

Pending parameters (set by `Parameter`) give control vertices and primitive variables to be interpolated across the primitive. The parameters must include 3D positions of the points (passed as the `"P"` parameter).

Primitive variables with interpolation type `vertex` require *npoints* values; primitive variables without an interpolation type require a single data value which does not vary from point to point.

If the primitive variable list contains a floating-point variable named `"width"`, the values will be used to determine the diameter of the point primitives. The width is measured in object space units, and defaults to 1.0 if no value is supplied. The `"width"` parmeter may either be `vertex`, specifying the width of each point individually, or no interpolation type to specify a single width for all points in the primitive.

If the primitive variable list contains a floating-point variable named `"rasterwidth"`, the values will be used to determine the diameter of the point primitives, measured in raster (pixel) units. The `"rasterwidth"` parmeter may either be `vertex`, specifying the width of each point individually, or no interpolation type to specify a single raster space width for all points in the primitive.

`Points` copies the parameter values and clears the pending parameter list (making it safe for the user to subsequently reuse or free the memory referenced by the preceeding `Parameter` calls).

EXAMPLE:

```
GelatoAPI *r;

float positions[4][3] = { ... }
float widths[4] = { .01, .02, .01, .04 };
r->Parameter ("P", positions);
r->Parameter ("vertex float width", widths);
r->Points (4);
```

`GelatoAPI::`**Curves** `(const char *interp, int ncurves, int vertspercurve)`
`GelatoAPI::`**Curves** `(int ncurves, int vertspercurve,`
`int order, const float *knot, float vmin, float vmax)`

`Curves` primitives look like thin tubes or ribbons and are very inexpensive to render, making them ideal for hair, fur, grass, struts seen from far away, etc.

This primitive draws *ncurves* individual curves, each of which is formed by *vertspercurve* vertices.

In the first, simpler form, the curves are piecewise linear or piecewise cubic, and are parameterized uniformly on $[0,1]$. The *interp* parameter describes the means of interpolating `vertex` variables, including `"P"`, along each individual curve. An *interp* value of `"linear"` indicates piecewise linear curves, whereas `"bezier"`, `"bspline"`, or `"catmull-rom"` indicate a piecewise-cubic interpolation of the specified basis. The *vertspercurve* must be an appropriate number for the particular interpolation type (i.e., ≥ 2 for linear, ≥ 4 for cubic, and $4+3i$ for Bezier).

The second form allows complete specification of arbitrary order and knot vector, much like `Patch`. The length of *knot* must be $vertspercurve + order$. Each individual curvelet follows a NURBS curve defined over the parametric range [*vmin*...*vmax*].

Pending parameters (set by `Parameter`) give control vertices and primitive variables to be interpolated across the curves. The parameters must include either 3D control vertices (passed as the `"P"` parameter), or 4D `hpoint` control vertices (`"Pw"`) to indicate rational curves.

The number of values required for any `vertex` primitive variable is $ncurves \times vertspercurve$. Primitive variables with interpolation type `linear` require $2 \cdot ncurves$ values, one for each curve endpoint, and will be linearly interpolated along the length of each curve. `perpiece` primitive variables require *ncurves* values, one for each curve, and will be constant along each individual curve. Primitive variables without interpolation types, as usual, require a single value that will be used for all the curves in the primitive.

`Curves` copies the parameter values and clears the pending parameter list (making it safe for the user to subsequently reuse or free the memory referenced by the preceeding `Parameter` calls).

An optional floating-point `"width"` parameter specifies the diameter of each curve in object space, and may be `perpiece` (one width value for each individual curve), `linear` (two width values for each individual curve, varying linearly along the length of the curve), `vertex` (same number of values and interpolation method as the vertex positions), or without an interpolation type (if all individual curves have the same diameter). If no `"width"` parameter is supplied, the diameter of all curves will be 1.0.

The individual curves, though actually shaped like ribbons, always face toward the viewing position (camera or ray) so that they subtend the full width, thus looking as if they were a thin tube rather than a ribbon. However, if you supply a `linear` or `vertex` `"N"` parameter, the curves will appear as ribbons whose orientation is fixed to be perpendicular to the normals supplied. Keep in mind that the primitive is designed for very thin (possibly subpixel) tubes or ribbons. Curves with exceptionally large widths may no longer look as good, due to the kinds of approximations used to render large numbers of them efficiently.

EXAMPLE:

```
// Make two 4-point Bezier curve segments that taper from a width
// of 0.1 at the base to 0.05 at the tip.
float positions[8][3] = { ... }
float widths[4] = { .1, 0.05, 0.1, 0.05 };
r->Parameter ("vertex point P", &positions);
```

```
r->Parameter ("linear float width", &widths);
r->Curves ("bezier", 2, 4);
```

2.8.4 Spheres

GelatoAPI::**Sphere** (float radius, float zmin, float zmax, float thetamax)

Creates a sphere with the given *radius*, centered at the origin of the local coordinate space. The *zmin* and *zmax* parameters can cut off the top and bottom of the sphere if they are not equal to $\pm radius$. The equation of the surface is:

$$\begin{aligned}
\phi_{min} &= \begin{cases} \operatorname{asin}\left(\frac{zmin}{radius}\right) & \text{if} \quad zmin > -|radius| \\ -\pi/2 & \text{if} \quad zmin \leq -|radius| \end{cases} \\
\phi_{max} &= \begin{cases} \operatorname{asin}\left(\frac{zmax}{radius}\right) & \text{if} \quad zmax < |radius| \\ \pi/2 & \text{if} \quad zmax \geq |radius| \end{cases} \\
\phi &= \phi_{min} + v \cdot (\phi_{max} - \phi_{min}) \\
\theta &= u \cdot (thetamax \cdot \pi/180) \\
x &= radius \cdot \cos\theta \cdot \cos\phi \\
y &= radius \cdot \sin\theta \cdot \cos\phi \\
z &= radius \cdot \sin\phi
\end{aligned}$$

If $thetamax < 0$ or $zmax < zmin$, the sphere is turned "inside-out" in the expected way.

Pending parameters (set by `Parameter`) give primitive variables to be interpolated across the primitive. Primitive variables with interpolation type `linear` require four values; primitive variables without an interpolation type require a single data value which does not vary across the patch. The `Sphere` is the only geometric primitive that is specified without control vertices (`"P"` or `"Pw"`). Because of this, `vertex` primitive variables are not accepted.

`Sphere` copies the parameter values and clears the pending parameter list (making it safe for the user to subsequently reuse or free the memory referenced by the preceeding `Parameter` calls).

EXAMPLE:

```
r->Sphere (1.0, -1.0, 1.0, 360.0);
```

2.9 Procedural Geometry Generators and Scene Files

`GelatoAPI::`**Input** `(GelatoAPI::Generator *procedure, const float *boundingbox=NULL)`

Submits to the renderer an already-created object, derived from the `GelatoAPI::Generator` class:

```
class Generator {
public:
    Generator () { }
    virtual void ~Generator () = 0;
    virtual bool bound (float *bbox) { return false; }
    virtual void run (GelatoAPI *rend, const char *params) = 0;
};
```

If *boundingbox* is `NULL`, the procedure's `bound()` routine will be called, which will either write a bound into `bbox[0..5]` and return `true`, or will return `false` without setting a bound. The bound is an axis-aligned bounding box (6 floats: xmin, xmax, ymin, ymax, zmin, zmax) in the local coordinate system.

The generator's `run` method will be invoked if and when the renderer needs the contents of the bounds. If no bounds are supplied to `Input` and also the generator's `bound()` function returns `false`, no bounds are available and so the generator will be invoked immediately.

The generator *must* be dynamically allocated with `new`, because Gelato is going to `delete` it when the renderer no longer needs it. Once you pass a particular Generator object to `Input()`, it belongs to the renderer, and the caller is no longer responsible for deleting it.

`GelatoAPI::`**Input** `(const char *name)`
`GelatoAPI::`**Input** `(const char *name, const float *boundingbox)`

Causes the renderer to read commands from a named source, which may be a scene file, a Generator DSO/DLL, the output of a program or shell command, or simply the string itself. Note that scene file readers and generators are really the same thing, since scene files merely invoke generators whose job is to read that file format.

The first word (up to a space) of *name* is presumed to be the name of a shared library (DSO or DLL) in the "generator" path (see Section 4.3.6). The library will be loaded and the following function in the DSO, with C linkage, will be called to construct and return a `Generator` object:

```
extern "C" {
    GelatoAPI::Generator *name_generator_create (const char *command);
}
```

The Generator returned is then is handled as if were given directly to the renderer, with the remainder of *name* (after the first word) supplied as the parameter to the Generator's `run` method.

If the symbol < follows the name of the generator, the remainder of the string will be executed as a shell commmand with `stdout` redirected to a temporary file, whose name will be used as the argument to the generator plugin. Assuming that the generator's purpose is to parse a file of a given format, this has the effect of executing a script to generate renderer input for any format for which there is a generator plugin. For example, the following executes the command "`make_sphere -radius 1`" and uses its terminal output as pyg input:

```
r->Input ("pyg < make_sphere -radius 1");
```

If the symbols << follow the name of the generator, the remainder of the string will be written to a temporary file, whose name will be used as the argument to the generator plugin. Assuming that the generator's purpose is to parse a file of a given format, this has the effect of allowing you to specify renderer commands in any input format for which there is a generator plugin. For example, the following creates a sphere using a Pyg command, via the C++ interface:

```
r->Input ("pyg << Sphere (1, -1, 1, 360)");
```

If *name* is a scene file in the "input" path (see Section 4.3.6), then that file name is used as the sole argument to a Generator whose name is the format of the file. The format is presumed to be the file extension (i.e., the characters in the filename following the last period). In other words,

```
Input("teapot.pyg")
```

is equivalent to

```
Input("pyg teapot.pyg")
```

Regardless of whether it is a generator DSO, scene file, program to run, or string to evaluate, if a *boundingbox* (consisting of 6 floats, xmin, xmax, ymin, ymax, zmin, zmax) is supplied (or if the Generator's `bound` method exists and returns `true`), the renderer will only read/invoke the commands if and when the contents of the bounds are required. If no *boundingbox* is supplied (and the Generator's `bound` method returns `false`), the input source is read immediately. Note that immediately-read sources may have side effects on the attribute state, but delayed-read sources (those with bounding boxes) do not have any such side effects because the input may happen much later.

EXAMPLES:

```
// Read commands from pyg file, immediately
r->Input ("teapot.pyg");

// Note that the above is equivalent to:
r->Input ("pyg teapot.pyg");

// Run myproc DSO/DLL, with args, if/when the contents of bbox are needed
float bbox[6] = {0, 3, -4, 2.6, -10, 10};
r->Input ("myproc 3.7 moo 8.9", bbox);

// Run the perl script with arguments, use its stdout as 'pyg' input
r->Input ("pyg < make_chair.pl -width 40 -height 80");
```

```
// Interpret a bit of Pyg directly, when the contents of bbox are needed
r->Input ("pyg << Sphere (2, -2, 2, 360)", bbox);
```

2.10 Error Management

Errors may occur during the execution of Gelato API calls. These may be caused by incorrect input (such as inconsistent or invalid data passed to the API routine), system errors (such as not finding a requested texture on disk), or for other reasons. The default behavior is to print error messages to `stderr`, or to log them to a file (specified by `Attribute ("string error:filename")`.

Client applications that need custom error handling may provide their own error manager and/or handler to the renderer. An *error manager* is a class declared in `errormanager.h`. You cannot subclass it, but you can create an `ErrorManager` for each `GelatoAPI` renderer, or create just one `ErrorManager` and share it among multiple renderers. Think of it as a wrapper for an error handler. The *error handler* as a "functor" (a class that behaves like a function) that is really just a callback function for processing an error or warning. You can create your own error handler functor from scratch, which can receive the callbacks and perform whatever action you prefer.

The basic sequence is to subclass `ErrorHandler` to create a low-level handler with the desired behavior, create an `ErrorManager` wrapping the handler, then pass the `ErrorManager` to `CreateRenderer`.

These classes and methods are all defined in `errormanager.h` and are within `namespace Gelato`. See Appendix A.3 for the full class definitions.

`ErrorHandler` class

```
class GELATO_EXPORT ErrorHandler {
public:
    virtual ~ErrorHandler () {}
    virtual void operator() (ErrCode errcode, const char *msg);
};
```

This minimal class definition is a simple callback for error messages. Alternative handlers may be created by subclassing `ErrorHandler` and replacing the virtual `operator()` (and optionally, the virtual destructor, if you add data members that need to be properly destroyed).

When the callback is made, the *errcode* is an error code describing the type of error, and *msg* is the actual error message. Valid error codes currently consist of: `ERR_INFO` (information only), `ERR_WARNING` (warning, may not really be wrong), `ERR_ERROR` (probably something wrong, but renderer will attempt to recover), `ERR_SEVERE` (severe, probably unrecoverable, error), `ERR_MESSAGE` (message, no error, usually for debugging).

EXAMPLE:

```
using namespace Gelato;

class TrivialHandler : public ErrorHandler {
  public:
    virtual void operator() (ErrCode code, const char *msg) {
        std::cerr << "Err " << (int)code << ": \"" << msg << "\"\n";
    }
```

```
};
```

`ErrorManager` methods

```
static ErrorManager *
    ErrorManager::Create (ErrorHandler *handler=NULL,
                          int verbosity=VERBOSITY_NORMAL)
```

Creates an `ErrorManager` that wraps the given `ErrorHandler`, using the specified verbosity level. If *handler* is NULL, a default handler will be created.

The *verbosity* may take on one the values: `VERBOSITY_QUIET`, the handler will only be passed errors; `VERBOSITY_NORMAL`, the handler will be passed errors and warnings; `VERBOSITY_INFO`, the handler will be passed errors, warnings, and also various informational messages.

```
int ErrorManager::Verbosity (void) const
```

Retrieves the current verbosity of the error manager, one of:
`VERBOSITY_QUIET`, `VERBOSITY_NORMAL`, `VERBOSITY_INFO`.

```
void ErrorManager::Verbosity (int verbosity)
```

Changes the verbosity of the error manager. Valid values for *verbosity* are
`VERBOSITY_QUIET`, `VERBOSITY_NORMAL`, or `VERBOSITY_INFO`.

```
ErrorHandler * ErrorManager::Handler (void) const
```

Returns a pointer to the error handler currently associated with the error manager.

```
void ErrorManager::Handler (ErrorHandler *handler)
```

Changes the error handler associated with this error manager. This merely replaces the error manager's pointer to the error handler, but does not free the previous error handler.

```
void ErrorManager::Info (const char *format, ...)
void ErrorManager::Warning (const char *format, ...)
void ErrorManager::Error (const char *format, ...)
void ErrorManager::Severe (const char *format, ...)
void ErrorManager::Message (const char *format, ...)
```

These are the ErrorManager routines that accept errors at various levels. The verbosity setting determines which will be supressed and which will be passed on to the handler.

`GelatoAPI` methods

static GelatoAPI * GelatoAPI::**CreateRenderer** (const char **type*=NULL, ErrorManager **err*=NULL)

When a non-NULL pointer to an `ErrorManager` is passed to `CreateRenderer`, that error manager will used by the renderer.

ErrorManager & GelatoAPI::**Err** (void)

Retuns a reference to the renderer's `ErrorManager`. This reference allows you to manipulate the renderer's error manager, for example, by replacing its error handler, changing its verbosity level, or even directly issuing `Error` or other calls that will go through the error handler.

Error Manager/Handler Example

```
using namespace Gelato;

class TrivialHandler : public ErrorHandler {
  public:
    virtual void operator() (ErrCode code, const char *msg) {
        std::cerr << "Err " << (int)code << ": \"" << msg << "\"\n";
    }
};

TrivialHandler *myhandler = new TrivialHandler;
ErrorManager *mymanager = new ErrorManager (myhandler);
GelatoAPI *rend = CreateRenderer (NULL, mymanager);
...
delete rend;
delete mymanager;
delete myhandler;
```

2.11 Re-rendering

Re-render mode is established by setting a global attribute before the first `World` call:

```
Attribute ("int rerender", 1)
```

In re-rendering mode, multiple `Render` calls are legal, and GelatoAPI calls are allowed to change certain scene elements between `Render` calls.[2] Re-renders will attempt to perform only the subset of computations necessary to re-render the scene given the specific changes that were made since the prior `Render` call.

When in re-rendering mode, subsequent to the first `Render` call, the following allowances and restrictions are in effect:

- After a call to `Render`, the graphics state (including the current transformation matrix) is restored to the way it was immediately following the first `World` call. By default, the scene is unchanged for subsequent re-renders.
- `World` is expected to be called as usual for re-rendered frames, with changes to global attributes, cameras, and outputs occurring prior to the `World` call, and changes to per-object attributes or geometry occurring after the `World` call.
- `Attribute` called prior to `World` may replace the values of global attributes for subsequent re-renders.
- `Camera` may re-emit a camera with altered parameters or position or add a new camera.
- `Output` may be called to add a new output, change the properties of an existing output image, or delete an output (by passing NULL or the string `"null"` as the format name).
- `Light` may be called to add a light, alter an existing light, or delete a light (by passing NULL or the string `"null"` as the shader name).
- The `Modify` API call may be used to enter a mode in which per-object attributes may be changed for some set of objects identified by name.

Individual renderer implementations conforming to the Gelato API may not fully support all of the potential modifications to the scene when re-rendering. Typical examples include that an implementation may disallow movement of the camera when re-rendering, allow only changing of certain attributes, or disallow addition of geometry during re-rendering. Unsupported changes may be silently ignored or may trigger a warning, but in either case the set of allowed and disallowed re-rendering changes should be fully documented (as it is for Gelato in Chapter 11).

[2] When not in re-render mode, only one call to `Render` is allowed, and the only GelatoAPI call valid after `Render` is `GetAttribute`.

2.12 Example Scene Specification

```
#include <stdio.h>
#include <stdlib.h>
#include "gelatoapi.h"


// Make a NURBS cylinder
void
make_nurbs_cylinder (GelatoAPI *r)
{
    float uknot[] = { 0, 0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 4 };
    float vknot[] = { 0, 0, 1, 1 };
    float Pw[] = { 1, 0, 0, 1,   1, 1, 0, 1,   0, 2, 0, 2,  -1, 1, 0, 1,
                  -1, 0, 0, 1,  -1, -1, 0, 1,  0, -2, 0, 2,  1, -1, 0, 1,
                   1, 0, 0, 1,   1, 0, -3, 1,  1, 1, -3, 1,  0, 2, -6, 2,
                  -1, 1, -3, 1, -1, 0, -3, 1, -1, -1, -3, 1, 0, -2, -6, 2,
                  1, -1, -3, 1,  1, 0, -3, 1 };

    r->Attribute ("shadingquality", 1);
    r->Parameter ("vertex hpoint Pw", Pw);
    r->Patch (9, 3, uknot, 0, 4, 2, 2, vknot, 0, 1);
}



int
main (int argc, char **argv)
{
    // Create a renderer object
    GelatoAPI *r = GelatoAPI::CreateRenderer ();

    // Check if we created the renderer okay. This might have failed,
    // for instance, if a license wasn't found.
    if (r == NULL) {
        fprintf (stderr, "Couldn't create renderer! Exiting...\n");
        exit (1);
    }

    // Set camera and image parameters
    r->Attribute ("projection", "perspective");
    float shutter[2] = { 0, 1 };
    r->Attribute ("shutter", &shutter);
    int res[2] = { 640, 480 };
    r->Attribute ("resolution", &res);
    r->Attribute ("fov", 45);

    // Specify an output image
    r->Output ("test.tif", "tiff", "rgba", "maincam");

    r->World ();    // Signal the end of the camera section

    // Set an ambient light
    r->Parameter ("float intensity", 0.01);
    r->Light ("amb1", "ambientlight");

    // Set a point light
    r->PushTransform ();
    r->Translate (2, -1, 11);
    r->Light ("pt1", "pointlight");
    r->PopTransform ();

    // Set a color attribute
```

```
    float C[3] = { 0, 1, 0 };
    r->Attribute ("C", &C);

    // Set specular-highlight color
    float specCol[3] = { 1, 1, 1 };
    r->Parameter ("color specularcolor", &specCol);
    r->Shader ("surface", "plastic");

    // Motion-blurred transformation
    r->Motion (2, 0.0, 1.0);
    r->Translate (-0.1,0,12);   // corresponding to time 0
    r->Translate (0.1,0,12);    // corresponding to time 1

    r->Rotate (50, 1, 0, 0);
    make_nurbs_cylinder (r);

    r->Render ("maincam");

    delete r;
}
```

3 Pyg: A Python-Based Scene File Format

Although Gelato does not dictate one specific scene file format, it does propose and provide a scene-reading plugin for a Python[1]-based scene file format called *Pyg* (standing for "Python for Gelato"). This chapter describes the use of Python as scene input.

3.1 Motivation

In addition to making direct calls to a renderer through the C++ API documented in chapter 2, it is often useful to store renderer commands and object models in *scene files*. Scene files are very helpful in a variety of circumstances:

- Sometimes it is simply easier to have a program (or human) output an ASCII file of renderer commands, rather than make direct C++ API calls to a renderer.
- You may not want the scene generation program and scene rendering program to be linked together and resident in memory at the same time (for reasons including reduction of RAM necessary, execution safety, or licensing issues).
- By generating parts of a scene once to a file, then having the renderer read the file when needed, you can save the cost of repeatedly generating the models if the frame must be repeated. The savings can be substantial by reusing the generated model for all frames of an animation, if the object does not deform or change over time.
- By storing parts of a scene in an ASCII format, it can be convenient to write *filters* in scripting languages such as Perl or Python whose function is to transform your scene in some way prior to being given to the renderer. This kind of trickery can make up for many deficiencies in modeling systems.

The Gelato API does not dictate one specific scene file format. Rather, Gelato allows user plugins that read scene formats and make corresponding C++ API calls, as documented in Section 12.

However, Gelato does define a scripted scene file format based on the Python language and includes a reader plugin for this format. This format is named "Pyg", and files in this

[1] Python is a widely-available, freely-distributed programming language. See `www.python.org`.

format should have the extension `.pyg`. Thus, the generator which reads files of this type is `pyg.generator.so`.

Pyg's proceduralism, flexibility, and access to Python's standard library are distinctive and attractive scene-file features. Although it is a relatively slow format to parse, this is perhaps not of critical importance in applications where flexibility is most required. For example, Pyg may be an interesting format for "master" scene-files - its procedural nature can allow sophisticated control over larger subsidiary scene-files (such as geometry) which may be stored in some more-optimized form. Gelato's DSO-based Input call can allow all these different file types to coexist transparently.

Pyg files are Python programs, with associated bindings to allow the Python scripts to call the `GelatoAPI` class methods. The remainder of this section documents the Pyg binding.

3.2 Basics

Pyg files are Python scripts. The scripts have access to a renderer object called `GelatoRenderer`. This object has a method corresponding to each C++ API call. For example, corresponding to the C++ `Attribute` method is:

```
GelatoRenderer.Attribute ("float fov", 45)
```

Also defined for each API call is a defined function that does not need to be referenced from `GelatoRenderer`, for example,

```
Attribute ("float fov", 45)
```

The above implicitly sets the option for `GelatoRenderer`.

Every C++ API call has an identically-named Python API call that performs the same functions, and which generall takes the same arguments in the same order. Exceptions are listed below.

Unlike C, there is never need to use pointer indirection to pass data arguments. Float or string arguments may be passed directly. More complex data (such as points, matrices, or arrays) may be passed as Python *sequences*. The sequences may be tuples (delimited by parenthesis), lists (delimited by brackets []), or any other object that obeys the sequence protocol (such as `xrange`).

For example, the `Attribute` to set `"C"` to a color may be called as:

```
Attribute ("color C", (1, 0.5, 0.5))
```

Because the script is Python, it is also legal to assign the sequence to a variable, then pass the variable as the data argument:

```
pink = (1, 0.5, 0.5)
Attribute ("color C", pink)
```

For methods that take parameters (which in C++ would be passed using the `Parameter` API call), the Python API allows passing parameters on the API call itself, as alternating strings and values, following the required arguments.

For example, in Python, the `Shader` call may be called like this (almost exactly mimicking the C++ convention):

```
Parameter ("float Kd", 0.5)
Parameter ("float Ks", 0.75)
Parameter ("string texturename", "grid.tx")
Shader ("surface", "paintedplastic")
```

or the parameters may be passed as part of the `Shader` call itself:

```
Shader ("surface", "paintedplastic", "float Kd", 0.5,
        "float Ks", 0.75, "string texturename", "grid.tx")
```

3.3 API Calls

Because there is a nearly one-to-one correspondence between the C++ and Python API calls, we provide only brief functional descriptions of each routine, below. We try to point out differences between the C++ and Python entry points, where they exist, and refer the reader to Chapter 2 for details on the functionality of each routine.

AppendTransform `(matrix m)`

Concatenate a matrix (given as a sequence of 16 numbers) onto the CTM.

Attribute `(string name, object value)`

Set an attribute to the given value. Depending on the attribute being set, the *value* may be a number, string, or sequence.

Camera `(string name, ...params...)`

Creates or replaces a camera.

Optional camera attributes may be specified either by previous calls to `Parameter`, or as optional alternating name / value pairs at the end of the `Camera` call itself.

Command `(string name)`

Invoke a renderer command.

Optional controls for certain commands may be specified either by previous calls to `Parameter`, or as optional alternating name / value pairs at the end of the `Command` call itself.

Comment `(string commenttext)`

For a renderer that is creating an command archive file, adds a comment in the appropriate output format. This call is ignored by “live” renderers producing images.

ConnectShaders `(string srclayer, string srcparam, string dstlayer, string dstparam)`

Connect the named parameters of two shader layers together. This call must occur between `ShaderGroupBegin` and `ShaderGroupEnd`.

Curves `(string interp, int ncurves, int nvertspercurve, ...params...)`
Curves `(int ncurves, int nvertspercurve, int order, sequence knot, number vmin, number vmax, ...params...)`

Creates a `Curves` primitive. The *knot* value is a sequence (such as a sequence).

Interpolated parameters may be specified either by previous calls to `Parameter`, or as optional alternating name / value pairs at the end of the `Curves` call itself.

GetAttribute `(string name)`

Returns the named attribute. Aggregate types, such as points, colors, or arrays, are returned as sequences whose elements are simple numbers or strings. Note that this is somewhat different than the C++ binding for `GetAttribute` (which passes a pointer to store the value, rather than returning it). If the attribute is not found, the return value will be `None`.

Input `(string name)`
Input `(string name, sequence boundingbox)`

Reads commands from a named source, which can either specify a scene file or the name of a Generator DSO/DLL. If the *boundingbox* is specified (as a sequence of six numbers), the input will only occur if and when the renderer needs to know the contents of the bounding box.

Light `(string lightid, string shadername, [string layername,] ...params...)`

Creates or replaces a light source. The layer name is optional.

Shader parameters may be specified either by previous calls to `Parameter`, or as optional alternating name / value pairs.

LightSwitch `(string lightid, boolean onoff)`

Turns an existing light on or off for subsequent primitives.

Mesh `(string interp, int_sequence nverts, int_sequence verts, ...params...)`

Creates a `Mesh` primitive.

Interpolated parameters may be specified either by previous calls to `Parameter`, or as optional alternating name / value pairs at the end of the `Mesh` call itself.

Unlike the C++ `Mesh` call, there is no need to pass the number of faces in the mesh — the *nverts* sequence gives the number of vertices for each face, and Python can discern the number of faces from the length of the sequence. The *verts* parameter is a sequence that contains the vertex indices of all vertices of all faces. The length of sequence *verts* should be the sum of all the elements in the *nverts* sequence.

Modify `(string namepattern)`

Turns on modify mode, so that subsequent changes affect all attribute states whose `"name"` attribute matches the regular expression *namepattern*.

Motion `(number time0, number time1, ...)`
Motion `(number_sequence times)`

Starts a Motion block. Note that unlike the C++ binding, you do not need to pass the *number* of time values – Python can discern the number of knots directly by the number of parameters or the length of the sequence.

Output `(string name, string format, string dataname, string camera, ...params...)`

Specifies an output image for rendered pixels, for a particular camera.

Optional parameters may be specified either by previous calls to `Parameter`, or as optional alternating name / value pairs at the end of the `Output` call itself.

Parameter `(string name, object value)`

Add a pending parameter, to be used by a subsequent call to `Camera`, `Output`, `Shader`, `Light`, or a geometric primitive. Depending on the declaration of the parameter being set, the *value* may be a number, string, or sequence.

Patch `(string interp, int nu, int nv, ...params...)`
Patch `(int nu, int uorder, number_sequence uknot, number umin, number umax, int nv, int vorder, number_sequence vknot, number vmin, number vmax, ...params...)`

Creates a patch specified by a rectangular array of control vertices.

Interpolated parameters may be specified either by previous calls to `Parameter`, or as optional alternating name / value pairs at the end of the `Patch` call itself.

Points `(int npoints, ...params...)`

Creates a `Points` geometric primitive.

Interpolated parameters may be specified either by previous calls to `Parameter`, or as optional alternating name / value pairs at the end of the `Points` call itself.

PopAttributes `()`

Restore the attribute state to the values it had when the corresponding `PushAttributes` call was made.

PopTransform `()`

Restore the transformation state to the values it had when the corresponding `PushTransform` call was made.

PushAttributes `()`

Save the attribute state.

PushTransform `()`

Save the transformation state.

Render `( [string camera] )`

Render the scene. The camera name is an optional parameter; if not specified, the default camera is rendered.

RestoreAttributes `(string name)`
RestoreAttributes `(string name, string attrs)`

Replaces some or all of the current attribute state with the saved attribute state with the given *name* (set by `SaveAttributes`).

Rotate `(number angle, number x, number y, number z)`
Rotate `(number angle, vector axis)`

Prepend the current transformation with a rotation of *angle* degrees about the axis defined by (x, y, z). The axis may also be defined as a sequence of 3 numeric values.

SaveAttributes `(string name)`
SaveAttributes `(string name, string attrs)`

Create a named alias for part or all of the current attribute state in a global dictionary of name/attribute state pairs. The *name* may be used with `RestoreAttributes`.

Scale `(number x, number y, number z)`
Scale `(vector scale)`

Prepend the current transformation with a scale factor of (*sx, sy, sz*). The scale values may also be defined as a sequence of 3 numeric values.

SetTransform `(matrix m)`
SetTransform `(string spacename)`

Replace the current transformation, either with the 4x4 matrix supplied as a sequence of 16 numbers, or with the named transformation.

Shader `(string shaderusage, string shadername, [layername,] ...params...)`
Shader `(string shaderusage)`

Sets the shader specified by *shadername* to be used as the current shader of the given *shaderusage*. The optional *layername* is a string that identifies the shader layer.

Shader parameters may be specified either by previous calls to `Parameter`, or as optional alternating name / value pairs.

If only the *shaderusage* parameter is passed, the shader assignments for that usage are cleared.

ShaderGroupBegin `()`

Begin a group of shader layers.

ShaderGroupEnd `()`

End a group of shader layers.

Sphere `(number radius, number zmin, number zmax, number thetamax, ...params...)`

Creates a `Sphere` primitive.

Interpolated parameters may be specified either by previous calls to `Parameter`, or as optional alternating name / value pairs at the end of the `Sphere` call itself.

Translate `(number x, number y, number z)`
Translate `(vector translation)`

Prepend a translation onto the CTM, either specified as three numbers (x, y, z), or a sequence containing 3 numeric elements.

TrimCurve `(int_sequence ncurves, int_sequence n, int_sequence order, number_sequence knot, int_sequence min, int_sequence max, number_sequence uvw)`

Sets the current trim curve.

Note that unlike the C++ binding, there is no parameter giving the number of loops — it is inferred from the length of the *ncurves* sequence that gives the number of curves for each loop.

World `()`

Marks the end of scene-wide attributes, tags the CTM to be `"world"` space, and marks the beginning of per-object information.

3.4 Example Pyg Scene File

This is a Pyg version of the C++-API example listing in Section 2.12. It features a motion-blurred NURBS cylinder, and two lights.

```
def nurbscyl():
    uknot = (0, 0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 4)
    vknot = (0, 0, 1, 1)
    Pw = ( 1, 0, 0, 1, 1, 1, 0, 1, 0, 2, 0, 2, -1, 1, 0, 1, -1, 0, 0, 1,
          -1, -1, 0, 1, 0, -2, 0, 2, 1, -1, 0, 1, 1, 0, 0, 1, 1, 0, -3, 1, 1,
          1, -3, 1, 0, 2, -6, 2, -1, 1, -3, 1, -1, 0, -3, 1, -1, -1, -3, 1,
          0, -2, -6, 2, 1, -1, -3, 1, 1, 0, -3, 1 )
    Patch (9, 3, uknot, 0, 4, 2, 2, vknot, 0, 1, "vertex hpoint Pw", Pw)

Attribute ("string projection", "perspective")
Attribute ("float[2] shutter", (0, 1))
Attribute ("int[2] resolution", (640, 480))
Attribute ("float fov", 45)

Output ("test.tif", "tiff", "rgba", "camera")

World ()
Light ("amb1", "ambientlight", "float intensity", 0.1)

PushTransform ()
Translate (1, 0, 9)
Light ("pt1", "pointlight", "float intensity", 1.0)
PopTransform ()

Attribute ("color C", (1, 0.5, 0.5))
Shader ("surface", "plastic", "float Ks", 0.9, "float Kd", 1)

Motion (0.0, 1.0)
Translate (1, 0, 12)
Translate (1, 0, 13)

Rotate (50, 1, 0, 0)
nurbscyl ()

Render ()
```

Assuming that the above code was in a file called `cyl.pyg`, it could be rendered with the following command:

```
gelato cyl.pyg
```

3.5 Calling Gelato from Python

In addition to naming a Pyg file to render on the `gelato` command line, it is also possible to run an ordinary Python script that makes Gelato API calls. The necessary steps to do this are:

1. The environment variable `$PYTHONPATH` needs to point to the location of the `_gelato.so` module, which is stored in `$GELATOHOME/lib`. For sh/bash:

   ```
   export PYTHONPATH=$GELATOHOME/lib:$PYTHONPATH
   ```

   For csh/tcsh:

   ```
   setenv PYTHONPATH $GELATOHOME/lib:$PYTHONPATH
   ```

2. The Python file needs to:

   ```
   import gelato
   ```

3.6 Efficiency Issues

Using a full Python interpreter adds tremendous overhead to the process of reading a scene file. Furthermore, Python is an interpreted language and has the somewhat unfortunate practice of reading, parsing, and storing the *entire* script before it begins to execute it. That may be fine for a 100-line script, but is undesirable when your Python program contains 500 MB of scene geometry. This leads to very wasteful memory bloat, but even if you had the RAM to burn, full interpretation of Python can take much longer to parse (10x or more) than reading a straightforward ASCII non-scriptable ("flat") scene format.

To alleviate these problems, the Pyg reader `pyg.generator.so` uses two important shortcuts: special treatment of a simple subset of Pyg ("Pyglet"), and Pyg "chunking," explained in the remainder of this section. Both Pyg chunking and Pyglet require a seek-able file as input, not a stream. This means that reading from console input, for example

```
gelato "pyg -" < foo.pyg
```

will be much more inefficient than reading Pyg directly from a file:

```
gelato foo.pyg
```

Pyglet

The Pyg reader automatically recognizes "flat" Python consisting stricly of Gelato API calls with constant arguments, having no variable references, control structures, arithmetic, or other function calls. We call this subset "Pyglet." Note that the output of `topyg` is guaranteed to be in this subset.

When reading a Pyg file, if it is recognized to be the Pyglet subset, the Pyg reader will automatically shift to a less expensive parsing method (speeding up by about 5x) and make the underlying API calls as they are received, rather than reading and parsing the entire file before making any calls.

Therefore, when the full flexibility of Python is not needed, a big performance gain can be achieved by sticking to the Pyglet subset (not using variable substitution, arithmetic, or control flow, ensuring that all function calls are to Gelato API routines and with only constant arguments).

Two environment variables control this behavior. If `PYG_ALWAYS_PYGLET` is set, all input is assumed to be Pyglet. This has even faster performance, since the Pyg reader doesn't even need to analyze the Pyg to determine if it is the Pyglet subset. If `PYG_NEVER_PYGLET` is set, the full Python interpreter will always be used, and the Pyglet subset interpreter will never be used.

Pyg Chunking

Even when the Pyg being read is not part of the Pyglet subset, the Pyg reader will "batch up" lines of Pyg and execute them when they have reached a certain batch size, rather than waiting until the entire file has been read before executing any of it.

Two environment variables control this behavior. If `PYG_ALWAYS_CHUNK` is set, the Pyg reader will attempt to maximally chunk its input, even when below the usual threshold. If `PYG_NEVER_CHUNK` is set, the default Python interpreter will always be used, and no chunking will occur.

3.7 Binary Pyg

Beginning with Gelato 1.1, Pyg supports the encoding of binary arrays of integers and floating point numbers. This makes Pyg files more compact (usually about half the original size), and also greatly speeds up reading of the resulting files (often by a factor of 2–5). Therefore, it is recommended for large scenes to use the binary encodings. Note, however, that Pyg with binary extensions is no longer valid standalone Python.

Binary pyg is recognized by the presence of characters in the Pyg file whose ASCII value is ≥ 128. Specifically, the currently-supported encodings are:

Code (decimal)	code (octal)	Meaning
128	0200	`int` array (little endian)
129	0201	`int` array (big endian)
130	0202	`float` array (little endian)
131	0203	`float` array (big endian)
132-255	0204-0377	reserved

`int` array

Arrays of integers are signified by octal character 0200 or 0201, followed by 4 bytes forming an integer n, followed by $n \times 4$ bytes comprising an array of n 32-bit signed integer values. Both the value n and the n integers following are encoded either *little endian* (least significant byte first) or *big endian* (most significant byte first), depending on whether the initial code was 0200 or 0201, respectively.

`float` array

Arrays of floats are signified by octal character 0202 or 0203, followed by 4 bytes forming

an integer n, followed by $n \times 4$ bytes comprising an array of n 32-bit IEEE floating point values. Both the value n and the n floats following are encoded either *little endian* (least significant byte first) or *big endian* (most significant byte first), depending on whether the initial code was 0202 or 0203, respectively.

When outputting Pyg using `libgelato`, you can cause the Pyg being written to use binary encoding for large arrays using:

```
r->Attribute ("int format:binary", 1);
```

See Section 4.3.12 for details.

3.8 Conversion to Pyg: the `topyg` program

The `topyg` program can generate the Pyg equivalent of any scene file for which there is a generator DSO/DLL, or for any other generator invocation. The program operates by using a Pyg-emitting `GelatoAPI` subclass to handle Gelato scene API calls made by generator DSOs.

`topyg [`*`options`*`]` *`input0`* `[`*`input1`*`...]`

Converts each input command into the equivalent Pyg. Each input may be either a filename (for a format for which a reader generator plugin can be found), or a full command with arguments.

Options include:

`-o` *`filename`*

Causes the resulting Pyg output to be written to the given *filename*. If the `-o` option is not used, the resulting Pyg will be written to the terminal (`stdout`).

`-p` *`path`*

Specifies a search path for generator plugins needed to read the formats of the input files. The search path is a colon-separated list of directories where the plugins may be found.

`-r` *`depth`*

Specifies how many recursive levels of `Input` statements will be fully expanded in the output of `topyg`. Beyond this level, such files will merely be referenced by an `Input` statement in `topyg`'s output. The default is 0, meaning that all scene commands that are the equivalent of `Input` will simply be echoed as `Input` calls. As an example,

```
topyg -r 1 in.pyg > out.pyg
```

will make `out.pyg` a copy of `in.pyg`, except that any files referenced by `Input` statements in `in.pyg` will be expanded inline in `out.pyg`. However, since the recursive limit is 1, any `Input` statements in a file included by `in.pyg` will still be referenced by `Input` statements in `out.pyg`, not fully echoed.

`-v`

Turns on verbose mode, causing extra information to be printed as `topyg` does its job.

Generator paths for `topyg`

NEW!

Remember that `topyg` needs to load generator plugins (see Chapter 12) to read the input files. By default, `topyg` searches for generator plugins in the following locations, in this order:

1. the path(s) specified by the `-p` command-line option, if it was used.
2. the path(s) specified by the environment variable `GELATO_GENERATOR_PATH`, if set.
3. the current working directory.
4. `$GELATOHOME/lib`.

Using `topyg` to convert other formats to Pyg

If you have a scene file in some format other than Pyg, you can use `topyg` to produce the equivalent Pyg file, assuming that you have a scene-reading generator plugin for that other format. For example, if you had a RIB file `myscene.rib` and a RIB-reading plugin (say, `rib.generator.so`), then you could convert the scene file from RIB to Pyg as follows:

```
topyg myscene.rib > myscene.pyg
```

Using `topyg` to view and/or "bake" the results of a procedural generator

Sometimes you may have a generator plugin that creates geometry on demand. In the process of debugging the generator, you may wish to see the equivalent Pyg of the geometry that the generator creates. Or, if the generator is expensive to run, you may simply wish to "bake" its output into a Pyg file rather than running it for every frame at runtime. Fortunately, you can use `topyg` to capture the function calls made by any generator.

For example, suppose you have a generator `grass.generator.so` that grows grass blades in a particular area of the scene. Perhaps this is invoked in your scene file as:

```
Input ("grass 0 0 0 14 0 28 0.001 3.96", (0,4,2,6,0,1))
```

Ordinarily, this would be executed dynamically every time the frame renders. You could capture the results of the procedural generator using `topyg` as follows (on the command line, not in the scene file):

```
topyg "grass 0 0 0 14 0 28 0.001 3.96" > somegrass.pyg
```

Notice that we simply take the full command, generator name as well as arguments, and give it to topyg as a single input by enclosing it as double quotes, and redirect the resulting Pyg to a file.

4 Gelato Attributes and Commands

This chapter documents all of the attributes recognized by Gelato for use with the `Camera`, `Output`, and `Attribute` API functions, and all types of commands recognized by the `Command` function.

4.1 Camera Attributes

These attributes set camera and image properties, and are normally set as optional parameters to the `Camera` command. They may also be set by `Attribute`, but in that case apply to the next camera that will be declared.

`"string projection" [`*`projectionname`*`]`

Sets the type of projection used by the camera. Gelato recognizes the projections `"perspective"` and `"orthographic"`. The default is `"perspective"`.

`"float fov" [`*`angle`*`]`

Sets the vertical field of view used by the camera. This parameter only has an effect if the projection is `"perspective"`. The *angle* is measured in degrees. If not set, the default (for a perspective camera) is 90.

`"float[4] screen" [`*`xmin xmax ymin ymax`*`]`

Specifies the region of `"screen"` space (points projected onto the $z = 1$ plane in camera coordinates) that is mapped to the image area. The $x = xmin$ line in `"screen"` space corresponds to the left edge of the raster image, $x = xmax$ to the right edge, $y = ymin$ to the lower edge, and $y = ymax$ to the upper edge.

If this attribute is not set, the default values are:

$$(-frameaspectratio, frameaspectratio, -1, 1)$$

(The frame aspect ratio is defined as $xresolution/yresolution$.)

In other words, the default behavior is that the image is centered around the z axis, with the y dimension running from -1 to 1, and the x dimension determined by the frame aspect ratio.

`"int[2] resolution" [`*`xres yres`*`]`

Sets the full resolution (in pixels) of the image to be rendered. The values are integers giving the horizontal and vertical resolution, respectively. The default is [640 480].

`"float pixelaspect" [`*`ratio`*`]`

Specifies the aspect ratio (width/height) of the pixels. The default value, 1.0, indicates square pixels.

`"float[4] crop" [`*`xmin xmax ymin ymax`*`]`

Designates a subregion of the image pixels to be rendered, bounded by *xmin, xmax* horizontally and *ymin, ymax* vertically. The *xmin, xmax, ymin, ymax* arguments are floating point numbers, expressed in "NDC" coordinates (that is, with the origin in the upper left corner and with coordinates ranging from 0 to 1 across and down the image, respectively). The default is for the entire image to be rendered (`0 1 0 1`).

The pixels output will range in *x* from `ceil(xres*crop[0])` to `ceil(xres*crop[1]-1)`, and in *y* from `ceil(yres*crop[2])` to `ceil(yres*crop[3]-1)`, inclusive. The values are clamped to the range [0,xres-1] and [0,yres-1]. Geometry outside the region may be processed and rendered as necessary to ensure that adjacent nonoverlapping crop windows will exactly match up, including properly filtering across the boundary.

`"float near" [`*`n`*`]`
`"float far" [`*`f`*`]`

Sets the near and far clipping planes. Geometry whose *z* coordinate in camera space is less than *near* or greater than *far* will not be visible. There are also some computations in which `"camera"` space *z* values are normalized using the clip plane values (for example, `"screen"` space *z* or the return value of the shading language `depth()` function). The default is near = `0.1`, far = `1.0e6`.

`"float[2] shutter" [`*`open close`*`]`

Specifies the time range in which the camera's shutter is open, allowing moving objects to form a blurred image. Unlike a real camera, longer shutter times will not increase the amount of light exposure or change the brightness of the image. If $open = close$, the scene will be rendered with no motion blur. The default, if no `"shutter"` attribute is given, is for the scene to be rendered at time 0.0 with no motion blur.

`"float fstop" [`*`fstop`*`]`
`"float focallength" [`*`focallength`*`]`
`"float focaldistance" [`*`focaldistance`*`]`

Collectively, these attributes specify the parameters of the camera that lead to "depth of field" effects, which simulates a camera lens with a particular focal length and f/stop, focused on objects at a given distance. The *focallength* and *focaldistance* parameters

are measured in units of "camera" space. If *fstop* is 1e30 (effectively infinity), a pinhole camera will be used, resulting in a perfectly sharp image at all distances (this is the default behavior if no "fstop" attribute is specified).

Consider the Pyg example:

```
Camera ("maincam", "fstop", 8, "focallength", 4, "focaldistance", 200)
```

If the scene was modeled such that "camera" space had units of centimeters, the command above sets up an f/8, 40mm lens focused on objects 200 cm from the camera.

"int[2] spatialquality" [*x y*]

Sets the quality level of the spatial antialiasing for geometric edges to a minimum of $x \times y$ subregions per pixel. The default is (4, 4).

"int temporalquality" [*n*]

Sets the quality level of the temporal antialiasing (motion blur) to a minimum of *n* different time values sampled. The default is 16 temporal samples.

"int dofquality" [*n*]

Sets the quality level of the depth of field to a minimum of *n* different lens values sampled. The default is 16 different lens samples.

"int[2] limits:bucketsize" [*x y*]

Sets the size (in *x* and *y* pixels) of the screen buckets that represent units of work for the renderer. The default is (32, 32). Ordinarily, there should be no reason to override the default, but advanced users may wish to tune performance on problematic scenes by adjusting this attribute.

"string bucketorder" [*direction*]

Sets the traversal order in which buckets are rendered on screen. Valid arguments are:

"horizontal"
: Top to bottom, and within each row, left to right, just like English text (this is the default).

"vertical"
: Left to right, and within each column, top to bottom. For some large scenes with a wide-screen aspect ratio, using "vertical" bucket order may render the scene faster and require much less memory.

"spiral"
: Start at the center of the screen and proceed outward in a spiral pattern. When rendering in "spiral" mode to a live iv display window, Shift-LeftMouse will re-center the spiral where you click the mouse.

"int limits:bucketblocksize"

For some bucket processing orders (particularly "horizontal" and "vertical"), buckets are clustered in order to increase coherence. The value of the argument is the number of buckets (both horizontally and vertically) that will be clustered into a batch. The default is 2, which means that groups of 4 adjacent buckets (in a 2×2 pattern) will be processed together. There should be no reason for users to ever modify this value, unless you are trying to adjust the number of buckets sent to each worker for network-parallel rendering.

"int limits:autopassreduction"

When this camera attribute is nonzero, Gelato will automatically lower the actual number of passes used for motion blur and depth of field in regions of the screen when there is less blur than would require the full temporalquality or dofquality. The default is 1, meaning that the quality levels will automatically be reduced when not needed. In theory, this should not be noticeable, but if it does produce visible artifacts, setting this camera attribute to 0 will ensure that the full temporal and DOF quality values are used throughout the image.

"float preview" [*p*]

Activates preview mode, in which several parameters are automatically changed to faster, lower-quality settings. The shading quality is set to the argument *p*, and other attributes are changed to their preview mode settings, which are described below. The default is 0, indicating that preview mode is not active.

When preview mode is enabled, the following additional camera attributes control various behaviors, at times overriding the similarly-named attributes documented elsewhere in this manual. These attributes are not used when the "preview" attribute is zero.

"int[2] preview:spatialquality" [*x y*]
Alternate spatial quality to use when in preview mode (default: 1,1).

"int preview:temporalquality" [*s*]
Alternate temporal quality for preview mode (default: 1, i.e., no motion blur).

"int preview:dofquality" [*s*]
Alternate DOF quality for preview mode (default: 1, i.e., no depth of field).

"int[2] preview:bucketsize" [*x y*]
Alternate bucket size for preview mode. The default is (2048,2048), which means that for most resolutions, the whole scene will be rendered as one bucket.

"float preview:dicecurvature" [*d*]
Alternate "dice:curvature" attribute when in preview mode (default: 90).

"int preview:thincurves" [*n*]

For large `Curves` calls, only one in every *n* individual curves will be drawn (this makes it draw faster while reducing visual clutter). The default is 10.

"int preview:thinpoints" [*n*]

For large `Points` calls, only one in every *n* individual points will be drawn (this makes it draw faster while reducing visual clutter). The default is 10.

"float preview:limitpixelsize" [*p*]

Objects whose bounding box covers less than this number of pixels of pixels covered) are drawn only as a bounding box. The default is 0, which means that all objects will be drawn as their real geometry rather than as bounding boxes.

"float preview:limitdistance" [*d*]

Objects farther from the camera than this distance are drawn only as a bounding box. The default is 1.0e38, which means that all objects will be drawn as their real geometry rather than as bounding boxes.

"string hider" [*name*]

Changes to an alternate hider. Currently, the only choice is `"default"` (the usual hider).

"string shadertype" [*name*]

Selects an alternate shading system. Currently supported options are `"gsl"` (the default – perform full GSL shading), `"defaultsurface"` (draw everything like the `"defaultsurface"` shader), and `"keyfillrim"` (draw with three automatically-placed, unshadowed lights). The non-GSL options may render previews very quickly by bypassing the usual shading system, and may also be used in conjunction with preview mode.

```
"float stereo:separation" [0]
"float stereo:convergence" [0]
"string stereo:projection" ["off-axis"]
"string stereo:shade" ["center"]
```

When the `"stereo:separation"` camera parameter is nonzero, the camera will render *two* views and output two images. The two views will be separated along the camera's *x*-axis by the amount given by the *separation* (in `"camera"` space units).

NEW!

If the projection is `"off-axis"` (the default), the cameras will face parallel directions but with their frusta constrained to share the same viewing area at $z = convergence$ (in camera space units); if `"parallel"`, the camera views will be parallel with regular on-axis frusta (convergence is ignored); if `"toe-in"`, the two cameras will be rotated about their *y* axes to converge at $z = convergence$. Figure 7.9 illustrates these three modes.

When rendering stereo views, Gelato still only computes tessellation rates and shading from a particular view, given by the `"stereo:shade"` attribute, which may have any of the values: `"left"` (shade correctly for the left view), `"right"` (shade correctly for the

right view), or "center" (shade correctly for the center axis between the two views). Shading from the center (the default choice) is most accurate, although shading from the left or right view is slightly faster and may be preferable if the parallax error is not noticeable.

4.2 Output Attributes

This section describes output image attributes, which may be set per output image as optional parameters to the `Output` command.

`"string filter" [`*`filtername`*`]`
`"float[2] filterwidth" [`*`xw yw`*`]`

Final image pixels are produced by taking a weighted average of the contribution of nearby subregions, including those from other pixels. The weights are determined by a pixel filter. The default is to use a `"gaussian"` filter with width 2 in each direction. This should be adequate for most images, but you can override with a custom filter and width (which may be different for each `Output` specified).

The filter shape may be specified by a combination of the `"filter"` parameter, which takes a string giving the name of the filter, and `"filterwidth"`, which takes an array of two floats specifying the x and y support widths of the filter.

For ordinary data (color, etc.), the filters supported are: `"gaussian"`, `"box"`, `"triangle"`, `"catmull-rom"`, `"sinc"`, `"blackman-harris"`, `"mitchell"`, and `"b-spline"`. For depth (z) data only, you may use the filters `"min"`, `"max"`, or `"average"`.

Note that `"gaussian"`, `"mitchell"`, `"box"`, `"triangle"`, `"b-spline"`, and `"blackman-harris"` actually get "wider" (and thus blurrier) as the filter width increase. The `"sinc"` and `"catmull-rom"` filters use the width to "window" the existing function, without changing the shape.

For certain filters, only particular widths make sense — the `"catmull-rom"` filter should always have width 4, and `"sinc"` should have a whole-number width (4 is a good value).

`"float gamma" [`*`gamma`*`]`
`"float gain" [`*`gain`*`]`

Before passing pixels to the image display driver (and thus prior to any quantization), the renderer transforms all `color` data according to following formula:

$$color = (color \cdot gain)^{1/gamma}$$

The default is $gain = 1, gamma = 1$.

`"int[4] quantize" [`*`zero one min max`*`]`
`"float dither" [`*`ditheramp`*`]`

All renderer data is computed and sent to the image driver as floating-point data. For those image formats that must store pixel values as integers, the image driver is expected to perform the conversion according to the following formula:

$$\begin{aligned} pixelval &= \text{round}(zero + (one - zero) * floatval + ditheramplitude * \text{random}()) \\ pixelval &= \text{clamp}(pixelval, min, max) \end{aligned}$$

This has the effect of scaling the values so that a value of 0.0 gets an integer output value of *zero* and a value of 1.0 gets an integer output value of *one*, a pseudo-random dither of amplitude *ditheramplitude* is added to eliminate banding artifacts, and the integer value is clamped to lie between *min* and *max*.

The values *zero*, *one*, *min*, and *max* are specified, respectively, as an array of 4 ints passed as the `"quantize"` parameter to `Output`. The *ditheramplitude* value is passed as the float argument to the `"dither"` parameter.

It is expected that image drivers honor the convention of using the range of *min* and *max* to determine the bit depth of the resulting output image: if both are ≤ 255, 8-bit integer (per channel); if both are ≤ 65535, 16-bit integer; otherwise 32-bit integer. If all four numeric parameters are 0, then no integer quantization is performed and a floating-point image is output. If *ditheramplitude* is 0 (as it should be for floating-point images), no dithering is performed. If the behavior of any particular image driver deviates from this convention, it should carefully document its behavior.

The default quantization is `(0, 255, 0, 255)` and the default dither is `0.5`, meaning that output will be 8 bits per channel.

`"string stereo:left" [""]`
`"string stereo:right" [""]`

When a camera is outputting two stereo views (see Sections 4.1 and 7.6), by default the files will be named "*filename*-left.*ext*" and "*filename*-right.*ext*" (where "*filename*.*ext*" is the output filename given to the single `Output` command). The `"stereo:left"` and `"stereo:right"` `Output` parameters allow you to override this automatic choice of filenames for the left and right views. These have no effect when not rendering a stereo camera (i.e., when the camera's `"stereo:separation"` parameter is zero).

`"int zchannels" [3]`

When the data being output is `"volz"`, indicating a volume shadow, the `"int zchannels"` parameter specifies how many z values to write for each pixel in the output file. The value must be at least 2, and defaults to 3. For volumes of very uneven density, more z channels may be used to improve accuracy of representing the volumetric density, but the resulting file size is proportional to the number of channels. Please see Section 10.1.3 for detailed information about generating and using volume shadows.

`"float opaquewidth" [3]`

When creating `"volz"` volumetric shadow depth images, this parameter gives a hint about the depth range over which which the opaque z's within a pixel are considered "the same object" when multiple samples per pixel are combined into a single output pixel for the volume depth map. The default value is 0.05. This is somewhat similar to "bias" for ordinary shadow maps — if the `"opaquewidth"` is too small, slanted opaque objects may appear to incorrectly self-shadow; but if it is too large, opaque objects that are too close (in depth) may fail to correctly shadow each other.

`"int dynamic"`

When nonzero, designates the output as *dynamic*, that is, no disk file will be written, but rather parts of the image will be rendered on-demand as required by the scene. The common use of this is *dynamic shadow maps* (see Section 10.1.5), in which shadow map tiles are generated “on the fly.”

4.3 Scene-wide Attributes

The attributes described in this section apply to the entire frame being rendered. They should be set only before the `World` call, since they cannot be changed for each object.

4.3.1 Pass naming

`"string pass" [""]`

Provides a name to the current render "pass." This is largely reserved for future support, but of course it may be queried by `GetAttribute` from the main API or `getattribute()` inside a shader. The default is the empty string, `""`.

4.3.2 Units

`"string units:length" [""]`
`"float units:lengthscale" [1]`

Explains the size of `"common"` space measurements, in terms of physical units of length. Valid unit names for `"units:length"` include `"mm"`, `"cm"`, `"m"`, `"km"`, `"in"`, `"ft"`, `"mi"`. The scale determines how many common units equal one physical unit. For example, to explain to the renderer that a distance of 1 in `"common"` space corresponds to 0.5 cm:

```
Attribute ("string units:length", "cm")
Attribute ("float units:lengthscale", 0.5)
```

Note that this merely allows the renderer to make correspondences between physical units and scene measurements (primarily via the GSL `transformu` function); setting these attributes do not actually alter the current transformation.

`"float units:fps" [24]`

Describes the animation rate, or frames per second. This establishes the conversion between frames and seconds. The default is 24, which is correct for motion picture film or for 24p HD.

`"string units:time" [""]`
`"float units:timescale" [1]`

Explains the time units used for the values passed to `Motion` and the `Camera` `"shutter"` attribute. Valid unit names for `"units:time"` include `"s"` and `"frames"`. The scale determines how many common time units (i.e., the `Motion` scale) equal one second or frame.

For example, to explain to the renderer that we are rendering 29.997 frames per second (NTSC) and that common (i.e., `Motion`) units are frames:

```
Attribute ("string units:time", "frames")
Attribute ("float units:timescale", 1.0)
Attribute ("float units:fps", 29.997)
```

4.3.3 Rendering and Re-rendering Control

`"int rerender" [0]`

When not in rerender mode (i.e., when this attribute has its default value of 0), only one call to `Render` is expected.

When this attribute is nonzero, the renderer prepares for *re-render mode*, in which multiple `Render` calls are legal, certain scene elements may be respecified for subsequent calls to `Render`, and the renderer will attempt to perform only the subset of computations necessary to rerender the scene given the specific changes.

`"int rerender:memory" [0]`

Specifies the amount of memory (in KB) used by the rerender system. If the value is zero (the default), the rerender system uses as much memory as there is available without causing excessive swapping.

`"int rerender:reshaderays" [1]`

When this attribute is nonzero (the default), an object will re-shade any reflection or refraction rays if objects visible in the reflection may have been changed by light updates (even if the reflective object itself does not have any changed lights shining on it). If this attribute is zero, any rays traced from the surface will not be re-traced or re-shaded when lights are updated. Avoiding reshading rays (by setting the attribute to 0) allows much faster re-rendering, although the resulting image may have incorrect reflections.

`"string rerender:filepattern" [".*"]`

The value of this attribute is a regular expression that describes files (such as texture maps, shadows, and spatial databases) that should be automatically checked for update when re-rendering. Such files that have been written since the last re-render will need to be re-read, whereas files that have not changed on disk since the last re-render may have their values completely cached and will avoid recomputation. The string `".*"` (which is the default) means that all files should be checked for update.

Checking the update time of files can be very expensive, especially if there are thousands of files and they are being read over a network. The purpose of this attribute is to restrict the files checked to a small subset of all files read during rendering, and assuming that all other files are unchanged between re-renders. For example, an application that allows relighting and thus wants automatic recomputation of lights (but does not expect ordinary textures to change between relighting) may restrict the update checks to only shadow maps with:

```
Attribute ("string rerender:filepattern", ".*\.sm")
```

4.3.4 Performance/Memory/Quality Tradeoff Attributes

`"int limits:threads" [1]`

Sets the maximum number of parallel execution threads that will be used to render the frame, on a single machine. If more than one thread is used, we say that it is *multi-threaded*. On a machine with multiple processors or with multi-core processors, using more than one thread can speed up rendering by performing many tasks in parallel.

Multithreading still consumes only a single renderer license per frame, NOT one license per thread, no matter how many threads you use. Note that multithreading (using more than one thread or processor on a single machine) is different from network-parallel rendering (using multiple separate computers), which is covered in Sections 4.3.11 and 6.6.

`"int limits:gridsize" [`*g*`]`

Sets the maximum number of surface points that will be shaded at one time. Primitives that are estimated to dice into more pieces than this limit will be parametrically split to form primitives of a more manageable size. The default is 256.

`"int limits:texturememory" [`*m*`]`

Sets the maximum amount of memory for a cache of texture from disk. When this amount is exceeded, texture tiles that have not been recently accessed will be freed to make room for new texture that is needed. The value is the number of kilobytes to use for the cache. The default is 20480, meaning that the renderer should use a 20MB texture cache.

`"int limits:texturefiles" [`*f*`]`

Sets the maximum number of file handles that the texture system will leave simultaneously open. When this limit is exceeded, the texture system will close files that have not recently been accessed (they may be reopened again later if more texture is needed from those files). The default is 1000 on Linux and 100 on Windows. Larger numbers will give somewhat better texturing performance, but many operating systems will have an inherent limit or will degrade performance as the number of open file handles increases. On some systems, setting this limit too high may cause the renderer to exceed the total number of available handles allowed by the operating system, leaving it unable to open more texture files.

`"float trimcurve:quality" [1]`

An overall scale on the quality of the approximation of trim curves. If trimmed regions have ragged-looking edges, increasing this number may improve them.

`"int limits:trimcurvememory" [`*m*`]`

Sets the maximum amount of memory for a cache of trim curve computations. The value is the number of kilobytes to use for the cache. The default is 10240, meaning that the renderer should use a 10MB trim cache. You shouldn't ever need to adjust this number, but you might try increasing it if you have many different trim curves in your scene and you believe performance is suffering because of this.

`"int limits:transparentlayers" [-1]`

NEW!

Sets the maximum number of transparent layers. Transparent surfaces deeper than this limit in any pixel will be ignored. Note that transparent surfaces may be ignored even if they are within this limit if they are hidden by closer surfaces whose cumulative opacity exceeds `"limits:opacitythreshold"`. The default for this attribute is -1, meaning that there is no limit to the number of transparent layers.

`"color limits:opacitythreshold" [`*r g b*`]`

Sets the threshold for which accumulated opacity is considered fully opaque (the renderer will not look "behind" the object). By default, this is (1,1,1). In environments with many stacked transparent objects, making the threshold slightly less than 1 can make rendering much more efficient. This threshold both stops camera views as well as shadow and reflection rays. Note that transparent surfaces may be ignored even if closer surfaces are less than the opacity threshold, if the number of closer transparent layers exceeds `"limits:transparentlayers"`.

`"color shadow:opacitythreshold" [`*r g b*`]`

Sets the threshold for which opacity is considered opaque for the purposes of rendering geometry into a shadow map. Shaded geometry will appear in a shadow depth map only where its opacity is greater than this value (in any of the channels). By default, this is (1,1,1), meaning that only fully opaque objects will appear in depth maps.

`"int limits:transparentgrids" [`*numlayers*`]`

The renderer batches the processing of transparent grids, and this value gives the number of adjacent layers it should handle per batch (the default is 32). This should never change the image appearance, but for scenes with many layers of transparency, tuning this parameter may improve performance.

`"string dice:fixeduname" ["u"]`
`"string dice:fixedvname" ["v"]`

When fixed dicing is used (Attribute `"dice:fixed"`, see Section 4.4.5), these scene-wide attributes allow you to override the choice of parametric *u* and *v* and instead specify an alternate parameter to which fixed dicing rates are tied. It is assumed that the named parameters are `"linear float"` or `"vertex float"` parameters attached to geometry. Any geometric primitives that do not have the named parameters attached will measure their fixed dicing (if they use fixed dicing) relative to *u* and *v*.

These attributes are useful when *baking* into 2D texture maps. Since parametric (u,v) are not necessarily an appropriate baking space, these attributes let you measure your fixed dicing rates in an alternate space that probably corresponds to the baking space.

4.3.5 Ray Tracing and Global Illumination

`"int ray:maxdepth" [`*`d`*`]`

Sets the maximum ray tracing recursion depth for rays spawned by `environment()` calls in shaders. A value of 0 means that no rays will ever be traced by `environment()`, even if the shaders request it; 1 means that you can see reflections, but not reflections of reflections; 2 means that you can see reflections of reflections, but not reflections of reflections of reflections; and so on. The default value is 2.

`"int ray:maxdepthcolor" [`*`c`*`]`

Sets the color for rays spawned by `environment()` calls but that do not actually trace because they are beyond the maximum ray depth. The default is black (0,0,0).

4.3.6 Search Paths

Various external files may be needed as the renderer is running, and unless they are specified as fully-qualified file paths, the renderer will need to search through directories to find those files. There exist attributes to set the directories in which to search for these files.

`"string path:input" [`*`pathlist`*`]`
`"string path:texture" [`*`pathlist`*`]`
`"string path:shader" [`*`pathlist`*`]`
`"string path:generator" [`*`pathlist`*`]`
`"string path:imageio" [`*`pathlist`*`]`

Sets the search path that the renderer will use for files that are needed at runtime.

The different search paths recognized by Gelato are:

`"input"`	scene files for `Input` calls, defaults to `".:$GELATOHOME/inputs"`.
`"texture"`	texture, shadow, and environment maps, defaults to `".:$GELATOHOME/textures"`.
`"shader"`	compiled shaders, defaults to `".:$GELATOHOME/shaders"`.
`"generator"`	DSO's/DLL's for `Input` calls, defaults to `".:$GELATOHOME/lib"`.
`"imageio"`	DSO's/DLL's for custom image format input/output plugins, defaults to `".:$GELATOHOME/lib"`.

Search path types in Gelato are specified as colon-separated lists of directory names (much like an execution path for shell commands). There are two special strings that have special meaning in Gelato's search paths:

`&` is replaced with the *previous* search path (i.e., what was the search path before this statement).

- `$VAR`, `${VAR}`, `$(VAR)`, and `%VAR%` are replaced by the value of environment variable `VAR`, if it exists (for any environment variable).

For example, you may set your generator path as follows (using Pyg):

```
Attribute ("string path:generator", "$HOME/lib/$ARCH:&")
```

The above statement will cause the renderer to find generator DSO's by first looking in a directory that is dependent on the architecture, then wherever the default (or previously set) path indicated.

4.3.7 Statistics, Debugging, and other Output

`"int verbosity" [1]`

Determines the amount of non-error status messages printed during routine rendering. Setting verbosity to 0 causes *nothing* to print other than error messages (this is equivalent to running Gelato with the `-silent` option). The default value of 1 also prints warnings, occasional information, and, if it is displaying on a terminal, a status update. A value of 2 causes lots of additional information to print, such as full paths to all shaders as they are opened.

`"string error:filename" [""]`

Gives the name of the file to write the error log. If set to the empty string `""` (the default), statistics will be written to `stderr`. It's probably not smart to do so, but it's possible to completely supress the printing of errors by redirecting them to `"/dev/null"` (on Unix-like systems).

`"int statistics:level" [0]`

Sets the amount of statistics to print when rendering is complete. The default, 0, prints no statistics. Increasing numbers print increasingly detailed statistics.

`"string statistics:filename" [""]`

Gives the name of the file to write the statistics. If set to the empty string `""` (the default), statistics will be written to `stdout`.

`"int debug:dso" [0]`

When nonzero, cause huge amounts of debugging information to print about all attempts to examine and load DSO's or DLL's (such as ImageIO, Shadeop, and Generator plugins). This can be useful when debugging why your DSO's are not being found or are not loading properly. The default is 0.

`"string debug:filesread" [""]`

When not an empty string, specifies the name of a file to which the renderer will write the filenames of all files read during the course of rendering (textures, SDB's, plug-ins, input files, etc.). This can be useful when debugging a scene to see exactly which texture, plug-in, etc., was read. The list may also be used to enumerate all resources that must be gathered in order to replicate rendering of the frame. The files read will be written one per line, in the format that can be used with the utility `tar -T`. The default is the empty string, meaning that no such list will be written.

Plug-ins that read files can inform the renderer of those files using `Command ("fileread")`, described in Section 4.5.

Note: this only works reliably for single frame renders on a single machine. For network rendering or Sorbetto, this list is not expected to be complete at this time.

`"int debug:shadernan" [0]`

When nonzero, extra debugging code will execute that will check for NaN (not a number) values while your shaders execute, printing error messages whenever your shaders generate a NaN value, alerting you to which shader, with source filename and line number. The default is 0 (no NaN detection). Note that turning on this debugging option is likely to severely decrease shading performance, so it should only be done when you are trying to debug a shader problem that you suspect is NaN-related.

`"string griddump:filename" [""]`

Gives the name of the file to write information about every shaded grid that is seen by the camera. Grid dumps are primarily helpful if you would like to capture all of the shaded grids processed by the renderer, to do further processing on them or to use them as input to another renderer or other tool. Be careful — these files can be huge! The default value (the empty string `""`) indicates that Gelato should not write out this information.

Grids are dumped as Pyg files, with each grid as a `Patch` call (i.e., as a bilinear patch mesh). Positions are in `"camera"` space, and the grids may be motion-blurred. It is possible to take those shaded grids and re-insert them into a future render (or generate your own shaded grids using another application and hand them to Gelato) by simply setting the surface shader to `"nullsurface"` and ensuring that there is no displacement shader on the surface (that signals that the colors on the bilinear `Patch` mesh should be used directly as a shaded grid). For example, if a previous render had dumped `"camera"` space grids into the file `"grids.pyg"`, the following idiom will insert them into a future render as pre-shaded grids:

```
PushAttributes ()
Shader ("displacement")
Shader ("surface", "nullsurface")
SetTransform ("camera")
Input ("grids.pyg")
PopAttributes ()
```

"int griddump:binary" [0]

If nonzero, any resulting grid dump will be use the binary Pyg extensions (see Section 3.7), resulting in grid dump files that are about half the size and may be read 2–5 times faster than ASCII grid dump files. The default is 0, meaning that the grid dump files will be ASCII.

4.3.8 Texture Attributes

"string texture:substtexture" [*texname*]
"string texture:substenv" [*envname*]

If non-empty, specifies the name of a substitute texture to use for all texture or environment map lookups.

There are two handy uses for this feature: (1) if you are trying to render a scene for which you don't have any textures (e.g., for debugging purposes), this lets you substitute a single texture or environment map rather than getting endless error messages about missing textures; (2) by making all textures access to the same map (particularly a very low-resolution one), you can confirm or deny that texture access or memory is a performance bottleneck in your scene.

"int texture:automipmap" [1]

If nonzero, allows Gelato to automatically create MIP-map textures (in a temporary directory, deleted upon completion of the render) any time it encounters a requested texture that is not a tiled MIP-map. A warning will be printed to the console or log file when this happens.

The default is 1, meaning that it will auto-generate MIP-maps. Note that this is much less efficient than pre-generating MIP-maps with `maketx` (see Section 9.1). Set this attribute to 0 in order to print an error message about non-MIP-map textures (and simply refuse to use the textures at all).

4.3.9 Spatial Database Attributes

"string spatialdb:write" [dbname]

Causes the named spatial database to be written to disk after rendering is completed.

"string spatialdb:read" [dbname]

Causes the named spatial database to be loaded from disk the first time it is queried by `spatialdbquery()`, `indirect()`, `occlusion()`, or `subsurface()`.

"string spatialdb:readonly" [dbname]

Causes the named spatial database to be loaded from disk the first time it is queried, and furthermore, `spatialdbquery()` will *always* succeed. If the named spatial database is one used by `indirect()` or `occlusion()`, then the `indirect()` and `occlusion()` will always interpolate values from the pre-loaded database rather than computing new ones.

"string spatialdb:writeonly" [dbname]

Causes the named spatial database to be written from disk, and also guarantees that it need not be queried during rendering. This constraint means that the renderer is free to write records to disk as they are added to the database, without having to accumulate the entire database in memory.

4.3.10 Renderer Information and Status Attributes

The following attributes store information about the renderer itself, or about its status while rendering a frame. These attributes are all *read-only*, that is, they may be queried by `GetAttribute` (or in GSL by `getattribute()`) but may not be set by `Attribute`.

"string renderer:name"

Returns the brand name of the renderer (default is "NVIDIA Gelato"). (This new name replaces the old attribute "renderer".)

"int[4] renderer:version"

Returns the major, minor, release, and patch numbers of the renderer (e.g., { 2, 1, 0, 0 }). (This new name replaces the old attribute "version".)

"string renderer:versionstring"

Returns the major, minor, release, and patch numbers of the renderer expressed as a string (e.g., "2.1.0.0"). (This new name replaces the old attribute "versionstring".)

"float render:progress"

This returns a value between 0.0 and 1.0 indicating how much of the scene has been rendered. A value of exactly 1.0 indicates that the render has finished (nearly finished renders will not accidentally be rounded up to 1.0 if the render is not yet done).

"int render:success"

NEW!

This returns a value of 1 if the last render was successul, i.e., rendered all pixels in the frame. A value of 0 indicates that not all pixels rendered because an error or user interruption stopped the render before the entire image was done. This is generally used in conjunction with "render:progress", and its value is reliable only when "render:progress" returns 1.0, indicating that the render is not still progressing.

4.3.11 Network-parallel Rendering Attributes

When performing network-parallel rendering with the `gelato` command-line renderer, you should simply use the `-net` command-line option (see Section 6.6). There is no need to set any of the attributes below if you are using the command-line `gelato`.

However, if you are writing a custom application that embeds Gelato in its library form, you will need to set the following attributes in order to do network-parallel rendering.

Please note that *network-parallel* refers to many different computers contributing to rendering a single frame. This is somewhat different to *multithreaded* rendering, which is about using multiple execution threads on a single machine (that presumably has more than one processor). For information on multithreading, please refer to Sections 4.3.4 and 6.6.

`"string net:workers"`

If this attribute is set, the string argument should contain a list of server names, separated by spaces, that will be the workers used for network-parallel rendering of the frame. Unless the list contains the name of the local machine (or its synonyms `"local"` or `"localhost"`), the local machine will not be one of the workers.

The default is the empty string, which indicates that all rendering should happen on the local machine.

`"string net:workercmd"`

This attribute sets the name of the program (and arguments) that should be executed to start rendering on each of the worker machines, when network-parallel rendering is used.

`"int net:tint"`

This is a debugging tool — when set to nonzero, each rendering bucket will be tinted, using a different tint color for each network rendering worker. The value also serves as a random number seed to control the color selection for tinting. This allows you to easily visualize how the frame is divided up among the workers, to be sure that network rendering is really happening and that it is doing a decent job of load balancing. The default is 0, which indicates that bucket tinting should not be used.

4.3.12 Scene-writing Attributes

These attributes that do not alter rendering at all, but rather control archive output (such as when using `libgelato` to output Pyg for later rendering, via `CreateRenderer("pyg")`). For such output libraries, these attributes control the output library but do not produce an `Attribute` statement in the output itself. These attributes are ignored entirely by "live" renderers.

`"int pyg:binary" [0]`
`"int format:binary" [0]`

When outputting a scene file, this attribute chooses a binary encoding, if available, when the argument is nonzero. A live renderer, or a library that does not support binary encodings, will simply ignore this attribute. The *pyg:binary* attribute is meant specifically

for the Pyg outputting library, and *format:binary* is a generic synonym that should be honored by other output formats, if they support binary output.

This is primarily used when using `libgelato` to output Pyg for later rendering, in order to trigger binary encoding of large arrays, which makes the resulting files smaller and allows them to be both written and read much more quickly than if the arrays were encoded as ASCII strings.

`"int pyg:separateparams" [0]`

The Pyg formats gives a choice between outputting parameters with the `Parameter` call, or embedding parameters as variable-length lists in the commands themselves (such as `Camera`, `Output`, `Shader`, and geometric primitives). This attribute controls which of the two methods are used when outputting Pyg, with a value of 0 (the default) causing parameters to be embedded in the comand, and a nonzero value causing parameters to be output separately via `Parameter` prior to the main command. It is purely an aesthetic choice and has no semantic difference in the resulting Pyg file. A live renderer will simply ignore this attribute.

`"int pyg:indent" [0]`

When outputting a scene file as Pyg, this attribute alters the indentation level (which starts out at 0 when a new Pyg file starts writing). A positive argument to this attribute will *increase* the indentation by that number of space characters, while a negative argument will *decrease* the indentation. The indentation will apply to every Pyg statement output by the library. Note that this is a global attribute, so the indentation level is not affected by `PopAttributes`. When setting the indentation with `Attribute`, the value is added to the indentation level (i.e., the argument is relative to the current surrounding indentation level), but if `GetAttribute` is used to retrieve the attribute, the actual current indentation level will be returned.

`"int limits:inputlevels" [0]`

NEW!

When outputting to a Pyg file, a call to `Input(Generator *, ...)` by default does not actually execute the generator, but rather just puts a comment in the Pyg stream indicating that this is where a Generator would be invoked. When set to a value greater than zero, this attribute controls the number of recursive levels for actually executing the Generator objects passed to `Input`.

4.4 Per-object Attributes

The attributes described in this section apply to individual objects, and may be set at any time (before or after `World`). They may be changed for each geometric primitive, and may be saved and restored with `PushAttributes`, `PopAttributes`, `SaveAttributes`, and `RestoreAttributes`.

4.4.1 Current Transformation

`"matrix transform"`

When retrieved by `GetAttribute`, this returns the current transformation matrix (CTM) that transforms points from local to world coordinates. When the CTM is motion-blurred, the matrix returned will be the transformation at shutter open time. If the attribute is queried before any camera is declared, the matrix will be for time 0. This attribute may be retrieved with `GetAttribute`, but may not be set by `Attribute`.

NEW!

4.4.2 Name

`"string name" [""]`

Tells the render a user-chosen name for subsequent geometric objects. This allows the renderer to print the object name when reporting errors, and also allows `Modify` to alter per-object attributes of states identified by name.

4.4.3 Geometry Sets

All geometric primitives are in one or more named *geometry sets*. Objects will only be visible to a particular camera if it is present in the geometry set that has the same name as the camera, or if it is present in a geometry set named `"camera"` (meaning visible to all cameras). The semantics of all other geometry sets are user defined — the most common use is to specify a group of primitives to ray-trace against, or a set of geometric primitives that comprises an area light source.

Camera geometry sets are special — any objects that are only in geometry sets corresponding to `Camera` calls will be culled if offscreen, occluded by opaque objects, or when rendering a different camera, and will have their memory reclaimed immediately upon completion of rendering of the pixels that they cover. Objects in geometry sets of other names (not corresponding to any cameras) will have a much more conservative reclamation strategy, since they may be ray traced. Therefore, you should not use the name of any camera as the name of a geometry set to raytrace, since by definition they are not raytraceable.

`"string geometryset" [`*`setmod`*`]`

This attribute changes the list of active geometry sets — that is, which geometry lists subsequently-declared primitives will be added to. The argument, *setmod*, is a comma-separated list of named geometry sets. A + character leading the name of a geometry set

will cause the set to become active. A leading - character will cause the named set to become inactive. No + or - will cause only the named sets to be active, and all others inactive.

EXAMPLES:

```
// Make geometry also appear in the "reflections" set (in addition
// to whichever sets it was in before)
r->Attribute ("geometryset", "+reflections");

// Make geometry NOT appear in the "shadow" set (but still in all other
// sets it was in before)
r->Attribute ("geometryset", "-shadow");

// Make geometry appear only to camera "maincam" (no other sets)
r->Attribute ("geometryset", "maincam");
```

4.4.4 Surface Appearance Attributes

"color C" [*r g b*]

Sets the default surface color C that will be available in the shader (if it is not overridden by supplying a value on the geometric primitive). The default is [1 1 1], indicating that subsequent surfaces are white.

EXAMPLE:

```
float C[3] = { 1, 0.5, 0.5 };
r->Attribute ("C", &C);
```

"color opacity" [*r g b*]

Sets the default surface opacity that will be available in the shader (if it is not overridden by supplying a value on the geometric primitive). The default is [1 1 1], indicating that subsequent surfaces are completely opaque.

"int holdout" [0]

If nonzero, marks subsequent objects as a "hold-out matte." Holdout objects block the view of objects behind them, leaving a "hole" in the final image. The argument value determines the holdout mode:

NEW!

0 ordinary geometry – shaders will determine color and opacity; objects "behind" will be shown to the extent that opacity < 1. (default)

1 holdout matte – color and opacity are 0 (shaders not run), yet blocks the view of other objects "behind" the surface.

2 shaded holdout matte – shaders are run to determine color and opacity, yet blocks the view of other objects "behind" the surface (even if opacity < 1).

`"string orientation" [orient]`

Alters the current orientation (that is, the rule that determines which of the two possible directions is chosen for the surface normal). The *orient* parameter is a string indicating one of five possible settings:

`"outside"`	same as the coordinate system's handedness (default)
`"inside"`	opposite the coordinate system's handedness
`"lh"`	left handed orientation (regardless of CTM handedness)
`"rh"`	right handed orientation (regardless of CTM handedness)
`"reverse"`	change to the opposite of the previous orientation

EXAMPLE:

```
char *orient = "outside";
r->Attribute ("orientation", &orient);
```

`"int twosided" [onoff]`

A nonzero value of *onoff* (the default) indicates that subsequent geometry should be visible from both sides.

A value of zero for *onoff* indicates that the renderer may discard subsequent backfacing geometry (i.e., those whose normals point away from the camera). For closed, opaque geometry whose surface normals always point to the outside of the object (or whichever side the camera will be on), backfacing geometry can be culled without changing the appearance of the image, and thus may be used as an optimization hint.

`"float shadingquality" [rate]`

Determines the spacing of shading calculations performed on surfaces, so that approximately *rate* shading samples occur per raster unit distance. For example, the default value of 1.0 indicates that shading points should be about one pixel apart from each other, i.e., surfaces should be shaded approximately once per pixel. A value of 2.0 would indicate that adjacent shading points should be about 1/2 pixel apart from each other, or about 4 shading samples per pixel area.

Higher `shadingquality` values are more expensive, but better quality. We strongly recommend using the default value of 1.0 for all ordinary high-quality rendering; you are unlikely to see improvements with higher quality settings in most situations. Values smaller than 1 will render more quickly, but will have less detail and may appear blurry or chunky (but this may be desired for faster previews).

4.4.5 Dicing and Tessellation Attributes

`"int dice:binary" [1]`

When nonzero, causes dicing rates for patch shapes to be rounded up to the next highest power of 2, which helps to eliminate cracking at patch boundaries.

`"float dice:curvature" [10]`

Sets the curvature threshold for dicing to capture the shape of objects. The argument is the deviation angle, in degrees, that forces additional tessellation.

`"float dice:highcurvature" [120]`

Sets the curvature threshold for dicing to capture the shape or shading of objects. The argument is the deviation angle, in degrees, that forces additional tessellation, no matter how small (in arc length) the object actually is.

`"float dice:motionfactor" [1]`

A scale factor controlling objects' tendency to tessellate more coarsely and run shaders less frequently, the faster they are moving (since, when blurred, you won't be able to see fine detail). This can be a big performance improvement for objects that are moving rapidly. The default value is `1`. Larger values shade even more coarsely when objects move rapidly, smaller values reduce this effect. A value of `0` indicates that no adjustment to the shading rate is made for motion-blurred objects. Objects that are not moving with respect to the camera are not affected at all by this attribute.

`"int dice:keepcreases" [0]`

When nonzero, special care is taken to ensure that objects with creases (e.g., multiple knot values for NURBS) force a geometric split to always capture the position of the sharp crease exactly. The default, 0, does not do this — if a split must occur for other reasons, it will happen at the crease, but the crease may be "diced over," softening it somewhat. Turning this attribute on will slow down rendering somewhat, but will guarantee that the creases are rendered perfectly. (Like all other attributes, you may turn it on and off on an object-by-object basis.)

`"int dice:rasterorient" [1]`

When set to 1 (the default), takes orientation into account when dicing geometry. If set to zero, this attribute forces subsequent geometry to dice at rates as if it were perpendicular to the camera. This can occasionally be useful for preserving fine dicing at grazing angles (such as a tricky ground plane on which the bumps are disappearing).

`"float[2] dice:fixed" [256 256]`

Dicing is usually adaptive, achieving a particular dicing rate (facet size in terms of screen area, determined by the other dicing attributes described in this section). But sometimes it's helpful to explicitly set a specific tessellation rate. This attribute completely locks down the facets per unit of u and v regardless of distance or orientation. This can be expensive, and it can significantly over- or under-dice — let the user beware! Fixed dicing rates, once on, can be disabled by setting the fixed dicing rate to (0, 0).

Note that the scene-wide attributes `"dice:fixeduname"` and `"dice:fixedvname"` (Section 4.3.4) allow you to substitute other user-supplied geometric parameters against which

fixed dicing rates are measured, overriding use of the default parametric *u* and *v* for dicing measurements.

`"float dice:thincurve" [1]`

`Curves` primitives whose width is less than this threshold (measured in pixels) will undergo optimizations based on the assumption that the viewer will not be able to discern color gradation across the width (i.e., *u*) direction of the curve. The default is 1, meaning that the optimizations apply to curves less than one pixel wide. Setting the value to 0 would turn off the optimization entirely.

`"int cull:occlusion" [1]`

When nonzero (the default is 1), subsequent primitives will be *occlusion culled*. That is, it will not be shaded if it is completely behind other opaque objects in the scene. If this attribute is zero, primitives will be shaded regardless of whether they are hidden by opaque foreground objects.

4.4.6 Displacement Attributes

```
"float displace:maxradius" [0]
"string displace:maxspace" ["common"]
```

Specifies the maximum amount that subsequent surfaces will displace: up to *maxradius* units as measured in the given *space*. This allows the renderer to expand the computed bounding boxes of subsequent geometry to include the "growth" of geometry due to displacement shaders.

4.4.7 Shading Attributes

`"string shading:view" ["camera"]`

NEW!

Selects among three methods of computing the view direction `I` for shaders executing on camera grids. Shading that is performed for ray tracing is not affected.

The default choice, `"camera"`, implements the usual meaning, where `I` is the vector from the camera position to the shading point. But when "baking," which is inherently view-independent, it's often useful to shade each surface point as if it were viewed from a point along the normal vector, as if from right above or below the surface. This can be achieved by using this attribute with the value `"outside"` (have `I` be `-Ng`) or `"inside"` (have `I` be `Ng`).

4.4.8 Trim Curve Control

`"string trimcurve:sense" ["inside"]`

Determines whether trim curves applied to `Patch` primitives will discard the geometry inside the trim region (if the value is `"inside"`, which is the default), or outside the trim region (if the value is `"outside"`).

`"float trimcurve:curvature" [10]`

Sets the curvature threshold for linearizing trim curves to capture their shape. The argument is the deviation angle, in degrees, that forces additional segments. If you see angular artifacts in your trim regions, you may want to reduce this number.

4.4.9 Ray Tracing Controls

`"int ray:opaqueshadows" [0]`

When nonzero, forces subsequent objects to be treated as opaque when computing ray-traced shadows. This results in faster ray-traced shadows, since the object need not be shaded to determine if it blocks all light along the shadow ray. If 0 (the default), the shader will be run at the hit point in order to determine the opacity of the object for ray traced shadow intersections. Shaders that do not set opacity and whose `Attribute "opacity"` is 1, as well as shaders that explicitly set opacity to 1, are automatically treated as opaque to shadow rays without having to actually run the shaders, even if the `"ray:opaqueshadows"` attribute is not set.

This attribute should be set to nonzero in two circumstances: (1) when the object is known to be opaque and you want to take advantages of the speedups for ray tracing opaque objects, but the shader is not one of the simple cases for which it can autodetect that it's opaque; or (2) the shader does not, in fact, make the object opaque, but you want it to cast a solid shadow anyway, without running the shader.

`"int ray:motion" [0]`

When zero (the default), rays do not consider the motion blur of objects. When nonzero, extra time and memory will be used to allow the rays to consider the motion of objects they hit. Note that this attribute applies to the *primary* object (the one that is reflective or that a shadow is cast upon), not the moving ray-traced object itself (the one visible in the reflection or casting the shadow).

The method that Gelato uses to calculate motion-blurred ray tracing works very well if the primary object is stationary with respect to the camera, but the ray-traced object is moving. It works somewhat less well when the relationships are reversed (that is, when the primary object is moving but the object visible to rays is not).

"int ray:displace" [1]

NEW!

Controls whether, for traced rays, the displacement shaders are run on entire grids (making many small pieces that need to be traced), or whether a bump-mapping approximation can be used. For objects where the amount of displacement is sufficiently small that a bump approximation is good enough (as seen in reflections or shadows), setting this attribute to 0 can *greatly* reduce the time and memory necessary to render the object. In either case, displacement shaders will truly move points to create ragged silhouettes as seen by the camera — this attribute only affects the appearance of objects in ray traced reflections or shadows.

"float shadow:bias" [*f*]

Sets the default bias used for shadow map lookups, ray-traced shadows, and (despite the name) ray-traced reflections. This may be overridden in the shader by passing an optional "bias" value to shadow(), environment(), occlusion, or other relevent function that requires bias. The default value is 0.01.

4.4.10 Indirect Illumination Attributes

"float indirect:maxerror" [0.25]

A maximum error metric for interpolation of indirect illumination samples. Smaller numbers result in higher quality, but longer rendering times. This value may be overridden by the optional "maxerror" argument to the GSL indirect() function.

"float indirect:maxpixeldist" [20]

The maximum distance allowed (measured in pixels) between indirect illumination samples. Smaller numbers result in higher quality, but longer rendering times. This value may be overridden by the optional "maxerror" argument to the GSL indirect() function.

"int indirect:minsamples" [3]

The minimum number of nearby samples required to interpolate sparsely-computed illumination samples. Larger numbers result in higher quality, but longer rendering times. Reducing the value of this parameter can speed up computation of indirect illumination, but may produce additional artifacts.

"string indirect:spatialdb" ["indirect.sdb"]

Name of the indirect illumination database to use. The name may be overridden by the optional "filename" argument to the GSL indirect() function.

4.4.11 Occlusion Query Attributes

`"float occlusion:maxerror" [0.25]`

A maximum error metric for interpolation of occlusion query samples. Smaller numbers result in higher quality, but longer rendering times. A value of 0 will cause full occlusion sampling to happen for every `occlusion()` call. This value may be overridden by the optional `"maxerror"` argument to the GSL `occlusion()` function.

`"float occlusion:maxpixeldist" [20]`

The maximum distance allowed (measured in pixels) between occlusion query samples. Smaller numbers result in higher quality, but longer rendering times. A value of 0 will cause full occlusion sampling to happen for every `occlusion()` call. This value may be overridden by the optional `"maxerror"` argument to the GSL `occlusion()` function.

`"int occlusion:minsamples" [3]`

The minimum number of nearby samples required to interpolate sparsely-computed occlusion samples. Larger numbers result in higher quality, but longer rendering times. Reducing the value of this parameter can speed up computation of occlusion, but may produce additional artifacts.

`"string occlusion:spatialdb" ["occlusion.sdb"]`

Name of the occlusion database to use. The name may be overridden by the optional `"filename"` argument to the GSL `occlusion()` function.

4.4.12 Re-rendering Attributes

`"int rerender:locked" [0]`

When this attribute is nonzero, re-rendering mode will not reshade subsequent primitives. In effect, it is a promise that the primitive will not change appearance, so re-rendering may reuse their original shading results, thus saving time. The default is 0, meaning that subsequent objects are subject to reshading if their shaders, lights, or attributes change during re-rendering.

`"int rerender:cachelights" [1]`

Inherited by any `Light` definition that succeeds it, this attribute controls whether the light source will have its results cached during re-rendering. By caching results, a light will avoid recomputation on a re-render unless one of the parameters to the light changed. A value of 0 indicates that a particular light should recompute on every re-render, even if that light has not had any of its parameters changed. The default is 1.

4.5 Commands

The `Command` function (see Section 2.3 for its formal specification) executes a "renderer command." The commands supported by Gelato are detailed below. Note that some of them are only sensible when issued in mid-render, which implies a non-blocking renderer such as is created by `CreateRenderer("remote")`.

`Command ("exit")`

Causes a renderer to terminate its process. It takes no parameters.

`Command ("return")`

When issued after a non-blocking `Render` (or during `Render` by, for example, a DSO), it causes the renderer to stop working on the frame as soon as it can do so safely. It is not guaranteed to stop instantaneously. but you can typically expect it to stop within a fraction of a second. It takes no parameters.

`Command ("wait")`

When issued after a non-blocking `Render`, it waits (blocks) until either the rendering is complete, or until a certain amount of time has passed, whichever occurs first. The amount of time is, by default, infinite (i.e., it will block until rendering is complete), but may be set by `Parameter("float time",`*seconds*`)`, giving a maximum amount of time to wait, in seconds. It is easy to detect whether `"wait"` has returned due to image completion or timeout by checking the `"render:progress"` attribute (see Section 4.3.10).

`Command ("system` *...command...*`")`
`Command ("system")`

Executes another program.

In the first form, in which a full command follows "system," the command is executed by spawning the default command shell (in the manner of C `system(3)` call. Because this is a shell command, wildcards will be expanded and other shell functionality may be used. For this form, no `Parameter` values are used.

In the second form (where only `"system"` is used as the command name), executes a program in a manner that is equivalent to the C `fork(2)` followed by `execv(3)`, spawning a new process. Two `Parameter` values are honored:

`"string[`*n*`] command"`
An array of *n* strings, the first of which contains the name of the program to execute, and the other elements contain the arguments to the program.

"int wait"

If nonzero, the Command will not return until the spawned program has completed. If zero, Command will return to the caller immediately while the spawned program runs "in the backround."

"int echo"

If nonzero, the command will also be echoed to the terminal (or to other designated error output) before being executed.

EXAMPLES:

```
Command ("system maketx foo.tif foo.tx")

Parameter ("int wait", 1)
Parameter ("string[4] command", ("iv", "-gamma", "2.2", "foo.tx"))
Command ("system")
```

Command ("mkdir")

NEW!

Crates a directory on the file system, if it does not already exist. Takes a single required Parameter:

"string name", (*dir*)

Specifies the directory path to create.

Command ("unlink")

NEW!

Deletes a file. This command takes a single required Parameter:

"string name", (*file*)

Specifies the file to delete.

Command ("bucketpriority")

For a blocking render, this command changes rendering priority of screen regions (buckets). It has no effect when active rendering is not occurring. There are two ways to re-organize bucket priority, depending on which of two possible Parameter values are also passed:

"int[2] center", (*x, y*)

Switches rendering order to a spiral pattern, centered upon the pixel coordinates passed as the parameter value.

`"int[4] rectangle"`, (*xmin, ymin, xmax, ymax*)
First renders the pixels within the given rectangle (in pixel coordinates), then returns to the previously active rendering pattern.

`Command ("fileread")`

Alerts the renderer that a file, given by the parameter `"name"`, has been read. This means that it will be reported if the attribute `"debug:filesread"` is used. The Command is useful for plug-ins that would like to add to the list of files read, when the renderer itself would be be unaware that the plug-in is reading the file.

`"string name"`, *filename*
Specifies the name of the file read.

5 Shading Language

5.1 Basics

5.1.1 Overall shader structure

A shader is organized as follows:

[optional function definitions]

shadertype shadername ([*params*])
{
    statements
}

Prior to the definition of the shader itself, there may be any number of functions (subroutines) declared. Those functions may then be called by the shader or by each other. A function may be called only after it is defined.

Every shader must have a *shadertype*. The shader types are described in Section 5.1.2. The use of shader parameters is described in Section 5.3. The various types of statements are described in Section 5.5.

5.1.2 Shader Types

The *shadertype* is the type of shader, one of `surface`, `displacement`, `volume`, `light`, or `shader` (indicating a generic shader). The shader's type determines which global variables it may access and which operations it may legally perform (for example, light shaders may not alter `P`, and volume shaders may not `emit` light). Generic shaders may be used as layers for any shader usage and are more lax in their compile-time checking of allowable operations, but Gelato will disallow use of a generic layer at render-time if it violates the rules of the shader usage (for example, a generic shader that writes `P` may only be used as a displacement layer).

5.1.3 Identifiers

Identifiers, used for names of variables, functions, and shaders, consist of one or more letters, numerals, or the underscore character. The first character of an identifier may not be a digit.

The following are valid identifiers in the shading language:

```
C
spec_color
hey19
```

Identifiers are *case sensitive*; in other words, the identifiers `variable`, `Variable`, and `VariAble` are three completely different variables.

5.1.4 Comments

As in C++, any text enclosed by `/*` and `*/` is considered a comment, as is any text following `//`, until the end of the line. Comments are completely ignored and have no effect on the meaning of your shader programs.

5.1.5 Preprocessor

Shaders are assumed to be filtered by a standard C preprocessor (such as `cpp` on Unix machines). The following preprocessor directives are supported and perform the same functionality as in the C or C++ languages:

```
#include
#define
#ifdef
#ifndef
#if
#endif
#else
```

5.2 Data Types

Gelato's shading language provides several built-in data types for performing computations inside your shader, detailed in the remainder of this section.

Though Gelato's shading language is superficially similar to C/C++, several important things are different. Several types, such as `color` and `point`, are provided that are not found in C because they make it more convenient to manipulate the graphical data that you need to manipulate when writing shaders. Although `float` will be familiar to C programmers, the shading language has no `double` or `int` types. In addition, the shading language does not support user-defined structures or pointers of any kind.

5.2.1 float

The `float` type is used for all scalar numeric values (including integer values), and is nearly identical to the equivalent data type in the C language. The `float` type is guaranteed to have precision at least equal to the IEEE 32-bit floating-point standard.

`float` constants are constructed the same way as in C, for example: `2`, `2.71828`, `1.0e-10`.

5.2.2 color

The `color` data type is used to represent 3-component spectral reflectivities and light energies (such as radiance). Colors are by default represented as RGB triples (`"rgb"` space). You can assemble a color out of three floats using a constructor:

```
color (.75, .5, .5)
```

A color constructor may also take an optional color space name:

```
color ("hsv", .2, .5, .63)
```

This constructs the color with coordinates (.2, .5, .63) in `"hsv"` space, and returns its RGB equivalent. Table 5.1 lists the color spaces that Gelato knows by name.

`"rgb"`	The color space that all colors start out in, and in which the renderer expects to find colors that are set by your shader (such as `C`, `opacity`, and `Cl`).
`"hsv"`	hue, saturation, and value.
`"hsl"`	hue, saturation, and lightness.
`"YIQ"`	the color space used for the NTSC television standard.
`"xyz"`	CIE *XYZ* coordinates.

Table 5.1: Names of the color spaces that Gelato recognizes.

The `+`, `-`, `*`, and `/` operators can be applied to two colors, performing the operations component-by-component. Colors may be also compared using the `==` and `!=` boolean operators. All of these operations may be performed between a `color` and a `float`, treating the `float` as if it were a `color` with all three components being identical.

Individual components of color variables may be set or accessed using array notation, for example,

```
color C;
float r = C[0];    // Get the 0th/red component of C
C[2] = 0;          // Set the 2nd/blue component of C
```

Component indices run from 0 through 2, inclusive. If a color is in `"rgb"` space, components are 0 for red, 1 for green, and 2 for blue. Non-integer indices will be rounded down to an integer value. It is an error to access a color component with an index outside the [0..2] range.

5.2.3 point, vector, normal

A `point` is a position in 3D space. A `vector` has a length and direction, but does not exist in a particular location. A `normal` is a vector that is perpendicular to a surface, and thus describes the surface's orientation. All three of these types are internally represented by three floating-point numbers.

This manual will often refer to these types collectively as “point-like” types (or, without loss of generality, simply as points). Note that points, vectors, and normals transform between coordinate systems using different transformation rules, so it is important that you choose the right types and use the right transformation routines on them.

All points, vectors, and normals are described relative to some coordinate system. All data provided to a shader (surface information, graphics state, parameters, and vertex data) are relative to one particular coordinate system that we call the "common" coordinate system. The "common" coordinate system is one that is convenient for the renderer's shading calculations.

Just like with color, the point-like types may be assembled from three floats using a constructor, for example:

```
point (0, 2.3, 1)
vector (a, b, c)
normal (0, 0, 1)
```

These expressions are interpreted as a point, vector, and normal whose three components are the floats given, relative to "common" space .

Also as with colors, an optional string argument to the constructor allows you to specify the coordinates relative to some other coordinate system:

```
Q = point ("object", 0, 0, 0);
```

This example assigns to Q the point at the origin of "object" space. However, this statement does *not* set the components of Q to (0,0,0)! Rather, Q will contain the "common" space coordinates of the point that is at the same location as the origin of "object" space. In other words, the point constructor that specifies a space name implicitly specifies a transformation to "common" space. This type of constructor also can be used for vectors and normals.

Table 5.2 lists the coordinate systems that Gelato recognizes by name. The API command SaveAttributes (see Section 2.4) may be used to give additional names to user-defined coordinate systems. These names may also be referenced inside your shader to designate transformations.

The +, -, *, and / operators can be applied to two point-like values, performing the operations component-by-component. Point-like values may be compared using the == and != boolean operators. All of these operations may be performed between a point-like value and a float, treating the float as if it were a point with all three components being identical.

Individual components of point, vector, and normal variables may be set or accessed using array notation, for example,

```
point P;
float x = P[0];    // Get the 0th/x component of P
P[2] = 0;          // Set the 2nd/z component of P
```

Component indices are 0 for x, 1 for y, and 2 for z. Non-integer indices will be rounded down to an integer value. It is an error to access a component with an index outside the [0..2] range.

"common"	The coordinate system that all spatial values are converted to prior to execution of shaders.
"object"	For each geometric primitive separately, the local coordinate system that was active when the geometric primitive was declared.
"shader"	For each shader separately, the coordinate system that was active when the shader was declared.
"world"	The coordinate system active at `World`.
"camera"	The coordinate system with its origin at the center of the camera lens, x-axis pointing right, y-axis pointing up, and z-axis pointing into the screen.
"screen"	The coordinate system of the camera's image plane (after perspective projection, if one is specified). Coordinate (0,0) in the center of the screen.
"raster"	2D pixel coordinates. The upper left corner of the image in `"raster"` space is (0,0), and the lower right corner is $(xres, yres)$.
"NDC"	2D Normalized Device Coordinates. The upper left corner of the image in `"NDC"` space is (0,0), and the lower right corner is (1,1).

Table 5.2: Names of predeclared geometric spaces.

5.2.4 matrix

The `matrix` type represents a matrix required to transform points and vectors between one coordinate system and another. Matrices are represented internally by 16 floats (a 4×4 homogeneous transformation matrix).

A `matrix` can be constructed from a single float or 16 floats. For example:

```
matrix zero = 0;   /* makes a matrix with all 0 components */
matrix ident = 1;  /* makes the identity matrix */

matrix I = matrix (1);   /* Also makes identity matrix */

/* Construct a matrix from 16 floats */
matrix m = matrix (m00, m01, m02, m03, m10, m11, m12, m13,
                   m20, m21, m22, m23, m30, m31, m32, m33);
```

Assigning a single floating-point number x to a matrix will result in a matrix with diagonal components all being x and other components being zero (i.e., x times the identity matrix). Constructing a matrix with 16 floats will create the matrix whose components are those floats, in row-major order.

Similar to point-like types, a `matrix` may be constructed in reference to a named space:

```
/* Construct matrices relative to something other than "common" */
matrix q = matrix ("shader", 1);
matrix m = matrix ("world", m00, m01, m02, m03, m10, m11, m12, m13,
                            m20, m21, m22, m23, m30, m31, m32, m33);
```

The first form creates the matrix that transforms points from `"shader"` space to `"common"` space. Transforming points by this matrix is identical to calling `transform("shader","common",...)`. The second form appends the world-to-common transformation matrix onto the 4×4 matrix with components $m_{0,0} ... m_{3,3}$. Note that although we have used `"shader"` and `"world"` space in our examples, any named space is acceptable.

Matrix values can be compared with `==` and `!=`. Also, the `*` operator between matrices denotes matrix multiplication, while `m1 / m2` denotes multiplying `m1` by the inverse of matrix `m2`. Thus, a matrix can be inverted by writing `1/m`.

Individual components of a matrix variable may be set or accessed using array notation, for example,

```
matrix M;
float r = M[row][col];
M[row][col] = 0;
```

Component indices run from 0 through 3, inclusive, for both row and column. Non-integer indices will be rounded down to an integer value. It is an error to access a matrix component with either a row or column index outside the [0..3] range.

5.2.5 string

The `string` type may hold character strings. The main application of strings is to provide the names of files where textures may be found. Strings can be compared using `==` and `!=`.

String constants are denoted by surrounding the characters with double quotes, as in `"I am a string literal"`. A string that contains no characters (`""`) is called an *empty* string.

As in C programs, strings may contain special escape sequences that begin with the backslash (‘`\`’) character: ‘`\n`’ (newline), ‘`\r`’ (carriage return), ‘`\t`’ (tab), ‘`\\`’ (backslash character), ‘`\"`’ (double quote character).

5.3 Shader Parameters

5.3.1 Shader Parameter Syntax

As described in Section 5.1.1, shaders have an optional list of parameter declarations with the following syntax:

type paramname = *defaultvalue* { , *moreparams* }

The *type* is one of the shading language data types (described in Section 5.2). The *paramname* is the name of the parameter, and it is given a *defaultvalue*, which will be the value of the parameter if another value is not supplied in the scene file. Several parameters may be declared in the parameter list, separated by commas (just like function definitions in C or C++).

Shader parameters may also be 1D arrays. The syntax of array usage is largely similar to C. Array parameters declared as follows:

type paramname [*arraylen*] = { *init0*, *init1* ... }

NEW!

An empty *arraylen* (that is, no number between the brackets giving the length) indicates an array parameter of *variable length*, meaning that the length is tentatively set to the number of initializers, but may be overridden at run time if the `Shader` (or `Light`) call passes a different number of elements, or if `ConnectShaders` hooks up to a different number of elements.

Shader parameters are treated as read-only — your shader may not modify their values — unless you declare them using the `output` keyword. Declaring a parameter as `output` makes it writable, and by writing new values your shader can communicate with other shaders attached to the same geometry or specify new data to be incorporated into the image output.

Below is an shader code highlighting parameter declarations:

```
surface bricks (
  /* Initializing parameters to constant values */
    float Ka = 1,
    float Kd = 1,
    float brickwidth = .28,
    color mortarcolor = color (.6,.6,.6),
    string bricktexture = "bricks.tx",
  /* Initializing parameters to other variables or computed values */
    float s = u,
    float t = v,
    point Pref = P,
    float brickheight = brickwidth / 4,
  /* Example of an ordinary array parameter */
    color brickcolors[3] = { color (.6,.1,.1),
                            color (.4,.15,.12), color (.65,.12,.11) },
  /* Example of an array parameter of variable length */
    float brickpattern[] = { 1, 2, 2, 2 },
  /* Example of an output parameter */
    output float in_mortar = 0)
{
  ...
}
```

The above example illustrates several parameter types including array parameters. The `in_mortar` parameter is an `output` parameter, and it is thus expected that the shader set its value somewhere in the body of the shader (but in this case will take on the value 0 if it is not set by the shader). Note that several parameters (`s`, `t`, and `Pref`) set their defaults to arbitrary expressions (in this case, the values of other variables) rather than strictly to constant values known at compile time.

5.3.2 How Shader Parameters Get Their Values

Before a shader starts executing, its parameters must be bound (that is, given particular values). These values can come from several places. In order of increasing-to-decreasing priority:

- If the geometry itself (that is, the `Patch`, `Mesh`, etc., call) had "primitive data" matching the type and name of a parameter, the primitive data will be interpolated and used as the value of the parameter of the same name. Otherwise...

- If the shader is one layer in a group and a `ConnectShaders` statement explicitly connects the parameter to an output of an earlier layer, then the earlier layer's output value will be used as this shader's input value according to the `ConnectShaders` arguments. If the parameter is a variable-length array, the true length will be set to the length of the connected array, or if elements are connected individually, the length will be set large enough to accomodate the largest element that is connected to. Otherwise...
- If the `Shader` or `Light` statement contained a parameter argument whose name and type matches the shader parameter, the value passed to `Shader` or `Light` will be used. If the parameter is a variable-length array, the true length will be set to the size of the data that's passed, rather than the original number of initializers in the shader.
- If not supplied by any of the above, the parameter's value will be as described by the default value or expression in parameter declaration in the shader source code. If the parameter is a variable-length array, the true length will be set to the number of element initializers in the shader.

5.4 Shader Metadata

A shader may optionally include *metadata*[1] consisting of hints or extra data about the shader. Metadata allows shader writers to annotate shader source code with extra information about the shader or its parameters that will be compiled into the .gso and can be queried by applications. The most important use of this is for the shader writer to specify user interface hints about shader parameters — for example, that a particular string parameter is intended to be a filename, or that a particular float parameter should be a checkbox (zero/nonzero) instead of a slider.

Metadata, if supplied, is denoted by the double brackets `[[` and `]]` enclosing a comma-separated list of parameter declarations. Each metadatum is given an ordinary data type and name, as with any other variable or parameter. However, metadata value initializers must be numeric or string constants (unlike the shader parameters, which may use full expressions for their default initializers).

Metadata about the shader as a whole is placed in the shader declaration, between the shader name and the parameter list.

Metadata about individual shader parameters is placed immediately after the parameter's initializing expression, but before the comma or closing parenthesis that terminates the parameter description.

The formal language specification of metadata can be found in Section 5.9, but use of metadata can best be explained by example. Below is an shader declaration highlighting the use of shader and parameter metadata:

```
surface bricks
        [[ string description = "Very realistic brick shader" ]]
   (
     float Ka = 1
              [[ string description = "Ambient scale",
```

[1]The term *metadata* refers to data about data. In this case, the shader code is data. The metadata is data about the shader code.

```
                        float UImin = 0, float UImax = 1 ]],
        float Kd = 1
                    [[ string description = "Diffuse scale",
                        float UImin = 0, float UImax = 1 ]],
        string bricktexture = "bricks.tx"
                    [[ string description = "Texture map for brick detail",
                        string UItype = "filename" ]],
        ...
      )
  {
    ...
  }
```

At present, no shader or parameter metadata are semantically meaningful. That is, they are completely arbitrary and do not affect the meaning or execution of the shader. (If future releases of this specification include semantically-meaningful metadata, a table of the reserved names and meanings will appear here.)

Most metadata therefore exist only to be embedded in the compiled shader and able to be queried by other applications. For example, in the shader fragment above, the metadata give additional information about the shader and each parameter such as a description and hints for user interface, presumably for a shader adjustment GUI that would customize itself based on these hints. The choice of metadata and their meaning is completely up to the shader writer and/or modeling system. However, we propose some conventions in Section 8.4.

5.5 Language Syntax

The bodies of shaders and shader functions are composed of *statements*. Statements in the shading language are terminated by semicolons (`;`). The types of statements allowed are:

Variable Declaration A local variable may be declared, and optionally, an initial value may be assigned to it.

Assignment A previously declared variable may have its value set to the evaluation of an expression.

Function call A previously declared function or a built-in function may be invoked.

Conditional The shading language supports `if` and `if-else` statements, just like the C language.

Loop The shading language supports `for`, `while`, and `do` loops, just like the C language.

Loop Modifier The `break` statement causes a loop to terminate, and the `continue` statement skips to the next iteration. Both statements may be called only within a loop.

Return statement Functions may terminate execution and/or return values with the `return` statement. A shader may also `return`.

Lighting Statement The shading language allows light shaders to generate light with the `emit` statement, and for surface and volume shaders to gather light with the `lights` statement.

Function Declaration A new function may be declared and defined, able to be called and invoked later within the same lexical scope.

New scope A new naming scope may be declared by enclosing statements with curly braces: { }

5.5.1 Variable Declarations and Assignments

Local variables are those that you, the shader writer, declare for your own use. They are analogous to local variables in C or any other general-purpose programming language. The syntax for declaring a variable in the shading language is:

type variablename
type variablename = *initializer*

where

- *type* is one of the basic data types.
- *variablename* is the name of the variable you are declaring.
- If you wish to give your variable an initial value, you may do so by assigning an *initializer*.

Arrays are also supported, declared as follows:

type variablename [*arraylen*]
type variablename [*arraylen*] = { *init0*, *init1* ... }

Arrays must have a constant length; they may not be dynamically sized. Also, only 1D arrays are allowed. Other than that, however, the syntax of array usage is largely similar to C.

Some examples of variable declarations are

```
float a, b;       /* Declare; current values are undefined */
float c = 1;      /* Declare and assign */
float d = b*a;    /* Another declaration and assignment */
float e[10];      /* The variable e is an array */
float f[3] = {42, 1.2, 5};  /* Initialize the elements */
```

5.5.2 Procedure Calls

Just like in C and many other programming languages, you may call functions by specifying their name, followed by parentheses, optionally with a comma-separated argument list:

functionname ( )

functionname (*arg1* , ... , *argn*)

If the function does not return any value (a `void` function) or if you wish to ignore the return value, a bare procedure call as above is a valid program statement. The function's return value (if it has one) may also be assigned to a variable, used in an expression, or passed as an argument to another function.

5.5.3 Conditionals

Conditionals in Gelato's shading language work exactly as in C:

```
if ( condition )
    truestatement
```

and

```
if ( condition )
    truestatement
else
    falsestatement
```

The statements can also be entire blocks, surrounded by curly braces. The condition may be one of the following Boolean operators: `==`, `!=` (equality and inequality); $<$, $<=$, $>$, $>=$ (less-than, less-than or equal, greater-than, greater-than or equal). Conditions may be combined using the logical operators: `&&` (and), `||` (or), `!` (not).

If the condition is a numerical value (`float`, `color`, etc.), then the condition will be evaluated as true if the numerical quantity is not equal to 0. If the condition is a `string`, then it will evaluate to true if the string is not the empty string (`""`).

5.5.4 Loops

Three types of loop constructs work identically to their equivalents in C. Repeated execution of statements for as long as a condition is true is possible with a `while` statement:

```
while ( condition )
    statement
```

Statements may be repeated with the test after the statements using:

```
do
    statement
while ( condition ) ;
```

Also, C++-like `for` loops are also allowed:

```
for ( init-statement ; condition ; expression )
    statement
```

Just like with C++, a `for` statement's initialization statement may contain variable initializations, which are scoped locally to `for` loop itself. For example,

```
for (float i = 0;  i < 3;  ++i) {
    ...
}
```

As with `if` statements, loop conditions may either be relations, numerical quantities (which are "true" if nonzero), or strings (which are "true" if they are not empty).

Loops of all three types may have `break` or `continue` statements. The `break` statement exits the loop immediately. The `continue` statement skips the rest of the body and proceeds to the next iteration.

5.5.5 Lighting Statements

The shading language has some special syntactic structures for emitting light in light shaders, gathering light in surface or volume shaders, and for tracing rays.

Emission of Light: `emit`

Within light shaders, the `emit` statement is available to control the emission of light from infinitely far and finite positions. Both statements set the `L` variable appropriately, and automatically set `Cl` to zero. It is expected that the code in the body of the `emit` statement will set `Cl` to the amount of light arriving at surface position `Ps`. There are several modes of the `emit` statement with different behaviors, as explained below.

For lights that have a definite, finitely close position:

```
emit ( point from ) {
     statements;
}

emit ( point from, vector axis, float angle ) {
     statements;
}
```

The first form of the `emit` statement indicates that light is emitted from position `from`, and is radiated in all directions. The `emit` statement implicitly sets `L = from - Ps`. The second form of `emit` also specifies a particular cone of light emission, given by an axis and angle. If `Ps` does not fall within the cone, the body of the `emit` statement will not be executed.

The `emit` statement may also be used to indicate that light is emitted from infinitely far away:

```
emit ( vector axis, float spreadangle ) {
     statements;
}
```

The effect of the above statement is to send light to every `Ps` from the same direction, given by *axis*. The `emit` statement sets the `L` variable to `-axis`. The result is that the light behaves as if the source was effectively infinitely far away (like the sun).

If *spreadangle* is zero, the rays are truly parallel. If *spreadangle* is is nonzero, light rays arrive at each `Ps` from a range of directions, instead of a single direction. Such lights are known as *broad solar lights* and are analogous to very distant but very large area lights (for example, the sun actually subtends a 1/2 degree angle when seen from Earth).

A special no-argument form of `emit` exists:

```
emit ( ) {
    statements;
}
```

When `emit` is not given any arguments, `L` will be set to the reflection direction of the surface that caused the light shader to be run. This is useful for environmental reflections that are rigged to appear as if they were light sources.

Gathering of light: `lights`

Surface and volume shaders may gather available light using another statement: `lights`.

```
lights ( point position ) {
    statements;
}

lights ( point position, vector axis, float angle ) {
    statements;
}
```

The `lights` statement loops over all non-ambient light sources (that is, those lights whose shaders contain an `emit` statement) visible from a particular *position*. In the first form, all lights are considered, and in the second form, only those lights whose directions are within *angle* of *axis* (typically, *angle*=$\pi/2$ and *axis*=`N`, which indicates that all light sources in the visible hemisphere from `P` should be considered). For each light source, the *statements* are executed, during which two additional variables are defined: `L` is the vector that points to the light source, and `Cl` is the color representing the incoming energy from that light source.

```
lights ( string category, point position ) {
    statements;
}

lights ( string category, point position, vector axis, float angle ) {
    statements;
```

```
}
```

If an optional *category* is given as the first argument to `lights`, then the *statements* will only be executed for those light sources whose light shader has a string parameter called `"__category"` and for which *category* matches one of the comma-separated names in `"__category"`.

The *category* may be a simple “boolean” expression of words separated by `&` or `|`. If any word is immediately preceeded by a `!`, it will only match if *not* found in the light shader’s category list. Associativity of `&` and `|` is simple left-to-right (*not* with differing precedence, as in the C language) and `!` has higher precedence.

For example, the *category* `"blue&!red"` matches lights that have `"blue"`, but lack `"red"`, in their comma-separated `"__category"` list. Also, a *category* of `"*"` matches every light with non-null categories, and `"-*"` matches only lights with no category. Also note that comma `,` is a synonym for `|`, and `-` is a synonym for `!`.

Tracing rays: trace

The `trace` construct may be used to generate and trace a bundle of rays, then execute code for each ray.

```
trace (string geom, point position,
         vector axis, float angle, float samples, ...)  {
     statements;
}
```

The `trace` construct casts *samples* rays at primitives in the named geometry set. The rays will originate from approximately *position* in a cone centered around *axis* and with a half-angle of *angle* (in radians). For each ray, statements in the loop body will be executed.

Optional arguments may follow *samples*, and are conveyed with alternating names and values (much like with an `environment()` call). Optional arguments include:

"bias", <float>

> Ignores ray hits closer than this value in order to prevent incorrect self-reflection of surfaces. If no `"bias"` is supplied, or if its value is less than 0, the global shadow bias will be used.

"maxhitdist", <float>

> Ignores ray hits farther than this value (measured in `"common"` space units). If no `"maxhitdist"` is supplied or if the value is ≤ 0, there is no maximum hit distance.

"distribution", <string>

> Specifies the sampling distribution of the rays within the cone. Currently supported distributions include `"cosine"` (the default), or `"uniform"`.

"shade", <float>

If the argument is zero, the object hit will not be shaded. This can be useful if you are casing rays only to detect if something is hit, or the distance to hit objects, but do not need the fully-shaded color of the hit object, and therefore would like to avoid the cost of shading it. The default is to shade objects that are hit by rays.

Inside the body of the trace loop, you can retrieve information with about the ray and the object it hit using getmessage("trace", *name, variable*). The following "trace" messages are supported:

- "hit", <float> nonzero if the ray hit anything, 0 if nothing hit
- "hitdist", <float> distance to nearest hit point, or 1e30 if no hit
- "L", <vector> actual ray direction
- "Ps", <point> actual ray position (may differ from *position*)
- "Cl", <color> color returned by the ray (full ray composite)
- "Ol", <color> opacity returned by the ray (full ray composite)
- "P", <point> hit point (undefined if no hit)
- *other* retrieve the named global or parameter of the closest object hit (including C, N, opacity, etc.)

The shader spawning the trace can also pass information to the object being hit, by using setmessage(), and then the hit object may do a getmessage("parent").

trace loops may be nested. The messages retrieved by getmessage("trace") are for the immediately-surrounding trace loop.

It is not necessary to use the trace construct for all ray tracing. Most common uses of ray tracing can be accomplished using environment(), shadow(), indirect(), or occlusion().

5.5.6 Scoping and Function Definitions

Even though the shading language provides many useful functions, you will probably want to write your own, just as you would in any other programming language. Defining your own functions is similar to doing it in C:

```
returntype functionname ( params )
{
    ... do some computations ...
    return return_value ;
}
```

However, in many ways shader function definitions are not quite like C:

- All function parameters are *passed by reference*. In other words, unlike C, the function does not have private copies of its parameters that can be modified without affecting their originals. Rather, function parameters are merely new names for the original variables (much like using the & reference parameters in C++).

- Recursion is not allowed. A function may not call itself, or call another function that will cause the caller to be called again.
- You may not compile functions separately from the body of your shader. The functions must be declared prior to use and in the same compilation pass as the rest of your shader (though you may place them in a separate file and use the `#include` mechanism).

Valid return types for functions are the same as variable declarations: `float`, `color`, `point`, `vector`, `normal`, `matrix`, `string`. You may declare a function as `void`, indicating that it does not return a value. You may not have a function that returns an array.

Functions obey standard variable lexical scope rules. Functions may be declared outside the scope of the shader itself, as you do in C, or within another function (somewhat like in Pascal). Variable and function names will always refer to the identically named objects in the *innermost containing scope*.

Following is an example:

```
float outerfunction (...)
{
    float x, y;

    float myfunc (float f)
    {
        float x;        /* local hides the one in the outer scope */
        P = 0;          /* refers to the globally-scoped P */
        return x+y;     /* x is local, y is from the outer scope*/
    }
    ...
}
```

5.5.7 `return` statement

Within a function, the `return` statement exits from the function, optionally returning a value (for non-`void` functions).

A `return` statement within the main body of a shader is equivalent to calling the built-in `exit()` function — that is, it takes an early exit from the shader without executing any additional code, even if the `return` statement is within a loop or conditional.

5.6 Expressions

The expressions available in the shading language include the following:

- constants: floating-point (e.g. `1.0`, `3`, `-2.35e4`) and string literals (e.g., `"hello"`).
- `color`, `point`, `vector`, `normal`, or `matrix` constructors, for example:

```
point ("world",1,2,3)
vector (1)
color (0, 0.25, 1)
```

- variable references (by name)
- array element references of variables (using `[]`)
- component references of `color`, `point`, `vector`, or `normal` variables (using `[]`), or of `matrix` variables (using `[][]`)
- prefix and postfix increment and decrement operators, just like C:

`++` *varref*	(pre-increment)
varref `++`	(post-increment)
`--` *varref*	(pre-decrement)
varref `--`	(post-decrement)

- unary and binary operators on other expressions, for example (in order of precedence):

`-` *expr*	(negation)
expr `/` *expr*	(division)
expr `*` *expr*	(multiplication)
expr `+` *expr*	(addition)
expr `-` *expr*	(subtraction)

The operators `+`, `-`, `*`, `/`, and the unary `-` (negation) may be used on any of the numeric types. For multi-component types (colors, vectors, matrices), these operators combine their arguments on a component-by-component basis.

The only operators that may be applied to the `matrix` type are `*` and `/`, which respectively denote matrix-matrix multiplication and matrix multiplication by the inverse of another matrix.

- relations between variables (all lower precedence than the numeric operators):

expr `==` *expr*	(equal)
expr `!=` *expr*	(not equal)
expr `<` *expr*	(less than)
expr `<=` *expr*	(less than or equal to)
expr `>` *expr*	(greater than)
expr `>=` *expr*	(greater than or equal to)

The `==` and `!=` comparisons may be performed between any two values of equal type, and are performed component-by-component for multi-component types. The relative ordering comparisons (such as `<` and `>`) may only be performed between two `float`'s.

Note that relations produce Boolean (true/false) values. Boolean values may be assigned to numeric variables, in which case it is evaluated as 0 if false, 1 if true.

- logical combinations of Boolean expressions:

expr && *expr*	(and)
expr || *expr*	(or)
! *expr*	(not)

- another expression enclosed in parenthesis: `( )`. Parenthesis can be used to guarantee associativity of operations.
- function calls
- assignment expressions, any of:

  var = *expr*
  var *= *expr*
  var /= *expr*
  var += *expr*
  var −= *expr*

- ternary operator, just like C: *condition* ? *expr1* : *expr2*
  If *condition* is true, *expr1* is returned, but if *condition* is false, *expr2* is returned.
- type casts, specified by simply having the type name in parenthesis in front of the value to cast:

```
(vector) P          /* cast a point to a vector */
(point) f           /* cast a float to a point */
(color) P           /* cast a point to a color */
```

  The three-component types (`point`, `vector`, `normal`, `color`) may be cast to other three-component types. A `float` may be cast to any of the three-component types (by placing the float in all three components) or to a `matrix` (which makes a matrix with all diagonal components being the `float`). Obviously, there are some type casts that are not allowed because they make no sense, like casting a `point` to a `float`, or casting a `string` to a numerical type.

5.7 Global variables

A variety of state information is available through pre-declared "global" variables. Somewhat different sets of global variables are available in different types of shaders. Table 5.3 lists all of the predefined global variables and their meanings, while Table 5.4 details which variables are accessible in each shader type.

Variable	Description
`point` **P**	Surface position — the point being shaded.
`vector` **I**	Incident ray direction — points from viewing position to `P`.
`normal` **N**	"Shading" normal of the surface at `P`.
`normal` **Ng**	True geometric normal of the surface at `P`.
`vector` **dPdu**, **dPdv**	Partial derivatives $\partial P/\partial u$ and $\partial P/\partial v$ of the surface at `P`.
`float` **u**, **v**	Geometric parameters of the surface at `P`.
`vector` **L**	Direction of incoming light (points from `Ps` to `Pl`).
`color` **Cl**	Incoming light color at P.
`point` **Ps**	Position at which the light is being queried (i.e., the position passed to `lights`).
`normal` **Ns**	Axis about which the light is being queried (i.e., the axis passed to `lights`), or the surface's `N` if no axis was passed to `lights`).
`point` **Pl**	Position of the light source.
`normal` **Nl**	Normal on the light (area light sources only).
`color` **C**	Incident ray color — the color of the light leaving the surface from `P` in the direction `-I`.
`color` **opacity**	Incident ray opacity — the degree to which the viewer can "see through" the surface at `P`.
`float` **time**	Current shutter time
`float` **dtime**	The amount of time covered by this shading sample.
`vector` **dPdtime**	How the surface position P is changing per unit time.

Table 5.3: Shader "Global" Variables summary table.

Variable	surface	displacement	volume	light
P	R	RW	R	R
I	R		R[1]	R[1]
N	RW	RW	R[1]	R[1]
Ng	R	R	R[1]	R[1]
dPdu, dPdv	R	R	R[1]	R[1]
u, v	R	R	R[1]	R[1]
L	R[2]		R[2]	RW
Cl	R[2]		R[2]	RW
Ps				R
Ns				R
Pl	R[2]		R[2]	R
Nl	R[2]		R[2]	R
C	RW		RW	
opacity	RW		RW	
time	R	R	R	R
dtime	R	R	R	R
dPdtime	R	R	R	R

[1]Certain surface variables (including N, Ng, I, etc.) are readable in light or volume shaders, but in that case store the values for the surface whose shading triggered execution of the shader. Use caution: light may be gathered from a completely different place (for example, when stepping through a volume).
[2]Light variables may be read in surface and volume shaders, but only inside lights loops, and they refer to the current light being examined by the loop.

Table 5.4: Accessibility of variables by shader type

5.8 Standard Library Functions

Gelato's shading language provides a standard library of functions. Note that some functions are *polymorphic*, that is, they can take arguments of several different types. In some cases we use the shorthand *ptype* to indicate a type that could be any of the point-like types `point`, `vector`, or `normal`.

5.8.1 Mathematical Constants

```
M_PI
PI
M_PI_2
M_E
M_LN2
M_LN10
M_SQRT2
M_SQRT1_2
```

Several mathematical constants are defined as part of the standard library: `M_PI` = π, `M_PI_2` = $\pi/2$, `M_E` = e (Euler's number, the base of natural logarithms), `M_LN2` = $\ln 2$, `M_LN10` = $\ln 10$, `M_SQRT2` = $\sqrt{2}$, `M_SQRT1_2` = $1/\sqrt{2}$, `PI` is a synonym for `M_PI`.

5.8.2 Mathematical Functions

float **radians** (float d)
float **degrees** (float r)

Convert degrees to radians, and radians to degrees, respectively.

float **sin** (float x)
float **cos** (float x)
float **tan** (float x)

Basic trigonometry. Angles, as in C, are assumed to be expressed in radians.

float **asin** (float y)
float **acos** (float x)
float **atan** (float y, float x)
float **atan** (float y_over_x)

Inverse trigonometric functions, returning radians. The `asin` and `acos` functions clamp their input to $[-1,1]$, and thus avoid out-of-range exceptions.

`float` **`sinh`** `(float x)`
`float` **`cosh`** `(float x)`
`float` **`tanh`** `(float x)`

Hyperbolic sine, cosine, and tangent.

`float` **`pow`** `(float x, float y)`
`float` **`exp`** `(float x)`
`float` **`log`** `(float x)`
`float` **`log2`** `(float x)`
`float` **`log10`** `(float x)`
`float` **`log`** `(float x, float base)`

Exponentials and logarithms: `pow` returns x^y; `exp` returns e^x; `log` returns the natural (base e) log, $\ln x$; `log2` returns the base-2 log; and `log10` returns the base-10 log; `log(x,b)` calculates an arbitrary base logarithm, $\log_b x$.

`float` **`sqrt`** `(float x)`
`float` **`inversesqrt`** `(float x)`

Return $\sqrt{x}$ and $1/\sqrt{x}$, respectively.

`float` **`hypot`** `(float x, float y)`
`float` **`hypot`** `(float x, float y, float z)`

Return $\sqrt{x^2+y^2}$ and $\sqrt{x^2+y^2+z^2}$, respectively.

`float` **`abs`** `(float x)`
`float` **`fabs`** `(float x)`

Return $|x|$. (The two functions are synonyms.)

`float` **`sign`** `(float x)`

Returns 1 if $x > 0$, -1 if $x < 0$, 0 if $x = 0$.

`float` **`floor`** `(float x)`
`float` **`ceil`** `(float x)`
`float` **`round`** `(float x)`
`float` **`trunc`** `(float x)`

`floor` returns the highest integer less than or equal to x. `ceil` returns the smallest integer greater than or equal to x. `round` returns the closest integer to x, rounding away from zero for the cases where x is halfway between two integers. `trunc` returns the integer part of x.

`float` **`mod`** `(float a, float b)`

Just like the *fmod* function in C, returns $a - b * \mathrm{floor}(a/b)$.

type **min** (*type* a, *type* b)
type **max** (*type* a, *type* b)

The `min` and `max` functions return the minimum or maximum, respectively, of two values of identical type (which may be any of `float`, `point`, `vector`, `normal`, or `color`). The variants that operate on colors or point-like objects operate on a component-by-component basis (i.e., separately for x, y, and z).

type **clamp** (*type* x, *type* minval, *type* maxval)

The `clamp` function returns

```
min(max(x,minval),maxval),
```

that is, the value x clamped to the specified range. The *type* may be any of `float`, `point`, `vector`, `normal`, or `color`. The variants that operate on colors or point-like objects operate on a component-by-component basis (i.e., separately for x, y, and z).

type **mix** (*type* x, *type* y, float alpha)

The `mix` function returns a linear blending of any simple numeric *type* (any of `float`, `color`, `point`, `vector`, `normal`, `matrix`): $x*(1-\alpha)+y*\alpha$

float **isnan** (float x)
float **isinf** (float x)
float **finite** (float x)

The `isnan` function returns 1 if x is `NaN` (not a valid IEEE floating point number), 0 otherwise. The `isinf` function returns 1 if x is `inf` (IEEE infinity), -1 if x is `-inf`, or 0 for any other number. The `finite` function returns 1 if x is not `NaN`, `Inf`, or `-Inf`, otherwise returns 0.

float **erf** (float x)
float **erfc** (float x)

The `erf` function returns the error function $\mathrm{erf}(x)=\frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}dt$. The `erfc` returns the complementary error function `1-erf(x)`.

5.8.3 Geometric Functions

ptype **ptype** (float f)
ptype **ptype** (float x, float y, float z)
ptype **ptype** (string space, float f)
ptype **ptype** (string space, float x, float y, float z)

Constructs a point-like value (*ptype* may be any of `point`, `vector`, or `normal`). When constructing from one float, all three components will be set to the single value.

If the optional `space` is supplied, the components are assumed to be relative to that coordinate system, and therefore the value is automatically transformed to `"common"` space. In other words,

```
p = point(myspace, x,y,z);
```

is equivalent to

```
p = transform (myspace, "common", point(x,y,z));
```

(And similarly for `vector`/`transformv` and `normal`/`transformn`.)

`float` **`dot`** `(vector a, vector b)`

Returns the dot product (a.k.a. inner product) of the two vectors (or normals), $a \cdot b$.

`vector` **`cross`** `(vector a, vector b)`

Returns the cross product of the two vectors (or normals), $a \times b$.

`float` **`length`** `(vector v)`

Returns the length of a vector or normal.

`float` **`distance`** `(point P0, point P1)`

Returns the distance between two points.

`float` **`distance`** `(point L0, point L1, point Q)`

Returns the distance from `Q` to the closest point on the line segment joining `L0` and `L1`.

`vector` **`normalize`** `(vector V)`
`vector` **`normalize`** `(normal V)`

Return a vector in the same direction as V but with length 1, that is, `V / length(V)`

`vector` **`faceforward`** `(vector N, vector I, vector Nref)`
`vector` **`faceforward`** `(vector N, vector I)`

If `dot(Nref,I)` < 0, returns `N`, otherwise returns `-N`. For the version with only two arguments, `Nref` is implicitly `Ng`, the true surface normal. The point of these routines is to return a version of `N` that faces towards the camera — in the direction "opposite" of `I`.

To further clarify the situation, here is the implementation of `faceforward` expressed in the shading language:

```
vector faceforward (vector N, vector I, vector Nref)
{
    return (dot(I,Nref) > 0) ?  -N : N;
}

vector faceforward (vector N, vector I)
{
    return faceforward (N, I, Ng);
}
```

vector **reflect** (vector I, vector N)

For incident vector `I` and surface orientation `N`, returns the reflection direction `R = I - 2*dot(N,I)*N`. Note that `N` must be normalized (unit length) for this formula to work properly.

vector **refract** (vector I, vector N, float eta)

For incident vector `I` and surface orientation `N`, returns the refraction direction using Snell's law. The `eta` parameter is the ratio of the index of refraction of the volume containing `I` divided by the index of refraction of the volume being entered.

```
void fresnel (vector I, normal N, float eta,
              output float Kr, output float Kt,
              output vector R, output vector T)
```

According to Snell's law and the Fresnel equations, `fresnel()` computes the reflection and transmission direction vectors `R` and `T`, respectively, as well as the scaling factors for reflected and transmitted light, `Kr` and `Kt`. The `I` parameter is the normalized incident ray, `N` is the normalized surface normal, and `eta` is the ratio of refractive index of the medium containing `I` to that on the opposite side of the surface.

point **transform** (matrix m, point p)
vector **transformv** (matrix m, vector v)
normal **transformn** (matrix m, normal n)

Transform a `point`, `vector`, or `normal` by the given 4x4 matrix.

point **transform** (string fromspacename, string tospacename, point pfrom)
vector **transformv** (string fromspacename, string tospacename, vector vfrom)
normal **transformn** (string fromspacename, string tospacename, normal nfrom)

Transform a `point`, `vector`, or `normal` (assumed to be represented by its `"fromspace"` coordinates) into the `tospacename` coordinate system.

`point` **transform** `(string tospacename, point p_common)`
`vector` **transformv** `(string tospacename, vector v_common)`
`normal` **transformn** `(string tospacename, normal n_common)`

Transform a `point`, `vector`, or `normal` (assumed to be in `"common"` space) into the `tospacename` coordinate system. This is equivalent to `transform("common",tospacename,p)`.

`point` **transform** `(string fromspacename, matrix tospace, point pfrom)`
`vector` **transformv** `(string fromspacename, matrix tospace, vector vfrom)`
`normal` **transformn** `(string fromspacename, matrix tospace, normal nfrom)`

`point` **transform** `(matrix fromspace, string tospacename, point pfrom)`
`vector` **transformv** `(matrix fromspace, string tospacename, vector vfrom)`
`normal` **transformn** `(matrix fromspace, string tospacename, normal nfrom)`

`point` **transform** `(matrix fromspace, matrix tospace, point pfrom)`
`vector` **transformv** `(maxtrix fromspace, matrix tospace, vector vfrom)`
`normal` **transformn** `(matrix fromspace, matrix tospace, normal nfrom)`

These routines work just like the ones that use the space names but instead use transformation matrices to specify the spaces to transform into and out of.

`float` **transformu** `(string tounits, float x)`
`float` **transformu** `(string fromunits, string tounits, float x)`

Transforms a measurement in *fromunits* (assumed to be `"common"` if not supplied) to its value expressed in *tounits*.

For length conversions, units may be any of `"mm"`, `"cm"`, `"m"`, `"km"` `"in"`, `"ft"`, `"mi"`, or the name of any coordinate system, including `"common"`, `"world"`, `"shader"`, or any coordinate system named by `SaveAttributes`.

For time conversions, units may be any of `"s"`, `"frames"`, or `"common"` (which indicates the timing scale used by `Motion` and the `"shutter"` camera parameter).

The units of `"common"` space, and by extension the relationships between all physical units and the scene, are determined by the unit attributes described in Section 4.3.2. It is invalid to convert between units of dissimilar category (for example, `"m"` may not be converted to `"s"`).

`point` **rotate** `(point Q, float angle, point P0, point P1)`

Returns the point computed by rotating point `Q` by `angle` radians about the axis that passes from point `P0` to `P1`.

5.8.4 Color Functions

`color` **color** `(float f)`
`color` **color** `(float r, float g, float b)`
`color` **color** `(string colorspace, float f)`
`color` **color** `(string colorspace, float a, float b, float c)`

Constructs a color. When constructing from one float, all three color components will be set to the single value.

If the optional `colorspace` is supplied, the components are assumed to be relative to that color space, and therefore the color is automatically transformed to `"rgb"` space. In other words,

```
c = color(myspace, a,b,c);
```

is equivalent to

```
c = transformc (myspace, "rgb", color(a,b,c));
```

Table 5.1 lists the color spaces that Gelato knows by name.

`float` **`luminance`** `(color c)`

Return the equivalent gray value that is perceptually equally bright as color *c*, assuming that *c* is in `"rgb"` space with standard RGB chromaticity.

`color` **`transformc`** `(string tospacename, color c_rgb)`
`color` **`transformc`** `(string fromspacename, string tospacename, color c_from)`

Transform a `color` from one color space to another. The first form assumes that `c_rgb` is already an "rgb" color and transforms it to another named color space. The second form transforms a color between two named color spaces.

5.8.5 Matrix Functions

`matrix` **`matrix`** `(float f)`
`matrix` **`matrix`** `(float f0, ... float f15)`

Constructs a matrix. When constructing from one float, the matrix will have all diagonal components equal to f, and all other components will be 0. When constructing from 16 floats, the values are filled into the matrix in row-major order.

`matrix` **`matrix`** `(string space, float f)`
`matrix` **`matrix`** `(string space, float f0, ... float f15)`

Constructs a matrix relative to the named space, automatically multiplying it by the `space`-to-`common` transformation matrix. Note that

```
m = matrix(myspace, 1);
```

returns the myspace-to-common transformation matrix.

`float` **`determinant`** `(matrix M)`

Returns the determinant of matrix `M`.

`matrix` **`transpose`** `(matrix M)`

Returns the transpose (exchange rows and columns) of matrix `M`.

`matrix` **`translate`** `(matrix M, point t)`
`matrix` **`rotate`** `(matrix M, float angle, vector axis)`
`matrix` **`scale`** `(matrix M, point t)`

Return a matrix that is the result of appending simple transformations onto the matrix `M`. These functions are similar to the `GelatoAPI Translate`, `Rotate`, and `Scale` commands, except that the rotation angle in `rotate()` is in radians, not in degrees.

5.8.6 Pattern Generation Functions

`float` **`step`** `(float edge, float x)`

Returns 0 if $x < edge$ and 1 if $x \geq edge$.

`float` **`smoothstep`** `(float edge0, float edge1, float x)`

Returns 0 if $x \leq edge0$, and 1 if $x \geq edge1$, and performs a smooth Hermite interpolation between 0 and 1 when $edge0 < x < edge1$. This is useful in cases where you would want a thresholding function with a smooth transition.

`type` **`noise`** `(float u)`
`type` **`noise`** `(float u, float v)`
`type` **`noise`** `(point p)`
`type` **`noise`** `(point p, float t)`

The `noise()` function returns a continuous, pseudo-random (but repeatable) scalar field defined on a 1- (`float`), 2- (2 `float`'s), 3- (`point`), or 4-dimensional (`point` and `float`) domain. The function always lies between 0 and 1, with a large-scale average value of 0.5, is fairly isotropic, has no easily detectable periodicity, and has most of its energy at frequencies between 0.5–1. This makes it ideal to use as a basis function for producing complex natural-looking patterns.

The return *type* may be any of `float`, `color`, `point`, `vector`, or `normal`, depending on the type of the variable the result is assigned to (or based on an explicit type cast). For multi-component return types (e.g., `point`), each component is an uncorrelated `float` noise function.

`type` **`snoise`** `(float u)`
`type` **`snoise`** `(float u, float v)`
`type` **`snoise`** `(point p)`
`type` **`snoise`** `(point p, float t)`

The `snoise()` function returns a continuous, pseudo-random (but repeatable) scalar field defined on a 1- (`float`), 2- (2 `float`'s), 3- (`point`), or 4-dimensional (`point` and `float`) domain. The function always lies between -1 and 1, with a large-scale average value of 0, is fairly isotropic, has no easily detectable periodicity, and has most of its energy at frequencies between 0.5–1. This makes it ideal to use as a basis function for producing complex natural-looking patterns.

The return *type* may be any of `float`, `color`, `point`, `vector`, or `normal`, depending on the type of the variable the result is assigned to (or based on an explicit type cast). For multi-component return types (e.g., `point`), each component is an uncorrelated `float` noise function.

Note that `snoise()` and `noise()` are essentially the same function, except scaled to have a different range and average.

type **cnoise** (float u)
type **cnoise** (float u, float v)
type **cnoise** (point p)
type **cnoise** (point p, float t)

The `cnoise()` function returns a discrete pseudo-random (but repeatable) scalar field defined on a 1- (`float`), 2- (2 `float`'s), 3- (`point`), or 4-dimensional (`point` and `float`) domain. The function is uniformly distributed on $(0,1)$, and has no easily detectable periodicity. The function is constant between integer values and discontinuous just before integer values — in other words, $cnoise(x) = cnoise(\text{floor}(x))$.

The return *type* may be any of `float`, `color`, `point`, `vector`, or `normal`, depending on the type of the variable the result is assigned to (or based on an explicit type cast). For multi-component return types (e.g., `point`), each component is an uncorrelated `float` cnoise function.

type **pnoise** (float u, float uperiod)
type **pnoise** (float u, float v, float uperiod, float vperiod)
type **pnoise** (point p, point pperiod)
type **pnoise** (point p, float t, point pperiod; float tperiod)

The `pnoise()` function is just like `noise()`, but repeats with the specified periodicity (which is rounded down to the nearest integer). Other than the user-set periodicity, the appearance and statistical properties of `pnoise` are identical to those of `noise()`.

type **psnoise** (float u, float uperiod)
type **psnoise** (float u, float v, float uperiod, float vperiod)
type **psnoise** (point p, point pperiod)
type **psnoise** (point p, float t, point pperiod; float tperiod)

The `psnoise()` function is just like `snoise()`, but repeats with the specified periodicity (which is rounded down to the nearest integer). Other than the user-set periodicity, the appearance and statistical properties of `psnoise` are identical to those of `snoise()`.

type **random** ( )
type **random** (float x)
type **random** (point p)

Returns a random value uniformly distributed between 0 and 1. The return *type* may be any of `float`, `color`, `point`, `vector`, or `normal`, depending on the type of the variable the result is assigned to (or based on an explicit type cast). For multi-component return types (e.g., `point`), each component is a separate uniformly distributed random number.

If `random()` is called with no arguments, a truly "random" result is returned, and it is not repeatable. If either a `float` or `point` (or, technically, a `vector` or `normal`) is passed as an argument, the `random()` function returns a seemingly random number that is really a *hash* of its argument (and is therefore repeatable if passed the exact same argument again).

type **randomgrid** ( )

Returns a random value uniformly distributed between 0 and 1, but which has the same value at all points being shaded at the same time (whereas the value `random()` is different at every point being shaded).

The return *type* may be any of `float`, `color`, `point`, `vector`, or `normal`, depending on the type of the variable the result is assigned to (or based on an explicit type cast). For multi-component return types (e.g., `point`), each component is a separate uniformly distributed random number.

type **spline** (float x, *type* y_0, y_1, ... y_{n-1})
type **spline** (string basis, float x, *type* y_0, *type* y_1, ... *type* y_{n-1})
type **spline** (float x, *type* y[])
type **spline** (string basis, float x, *type* y[])

Fit a spline to uniformly spaced data values $y_0 ... y_{n-1}$, returning the interpolated value at *x* (which will be clamped to lie on $[0,1]$). Instead of *n* data values, you may pass an array. The `spline` function will determine the length of the array.

The return *type* may be any of `float`, `color`, `point`, `vector`, or `normal`, depending on the type of the data values `y`. For multi-component splines (e.g., `color` splines), each component is interpolated separately.

Optionally, a spline basis may be specified by name. Valid basis names are: `"catmull-rom"`, `"bezier"`, `"bspline"`, `"hermite"`, or `"linear"`. If no basis name is supplied, `"catmull-rom"` is used. The number of data values must be $4n+3$ for Bezier splines, and $4n+2$ Hermite splines. Any number of knots may be used for Catmull-Rom or Linear splines, but the number of knots must be ≥ 4. To maintain consistency with the other spline types, linear splines will ignore the first and last data value, interpolating piecewise-linearly between y_1 and y_{n-2}.

The following basis names are also supported: `"solvecatmull-rom"`, `"solvebezier"`, `"solvebspline"`, `"solvehermite"`, `"solvelinear"`. For any of the `"solve"` spline types, whose data values may only be `float`, the spline's *inverse* is computed. That is,

spline() returns the lookup value for which the spline's result would be *x*. Results are undefined and likely unstable if the knots do not determine a monotonically increasing spline.

5.8.7 Derivatives, Area Operators, and Antialiased Functions

float **deltau** (float x), **deltav** (float x)
vector **deltau** (point x), **deltav** (point x)
vector **deltau** (vector x), **deltav** (vector x)
color **deltau** (color x), **deltav** (color x)

Compute differentials of the argument. deltau and deltav compute the approximate change in their argument expressions between adjacent shading sample in each of two principal directions on the surface. For most surfaces, these principal directions correspond to the parametric surface *u* and *v*, but this is not guaranteed.

float **filterwidth** (float x)
float **filterwidth** (point x)
float **filterwidth** (vector x)
float **filterwidth** (color x)

Computes the approximate difference to be expected between adjacent shading samples (in any direction) of the expression argument. This is very handy for antialiasing of shader patterns.

normal **surfacenormal** (point p)

Returns a vector perpendicular to the surface that is defined by point p (as p is computed at all surface points). This can be approximated by cross(deltau(p),deltav(p)), though the implementation of surfacenormal also takes surface orientation into consideration, and handles difficult cases where the tangents vanish.

float **samplearea** (point p)

Returns the differential area corresponding to the position. This roughly corresponds to length(deltau(p)) * length(deltav(p)), except that the value of samplearea uses the *true* differentials, not the deltas that are "smoothed" to ensure that derivatives are smooth across grid boundaries.

float **aastep** (float edge, float s, ...)
float **aastep** (float edge, float s, float ds, ...)

The aastep function provides an analytically antialiased step function. In its two-argument form, it takes parameters identical to step, but returns a result that is filtered over the area

of the surface element being shaded. In its three-argument form, the `aastep` function is filtered over the range [`s-ds/2`, `s+ds/2`].

In both forms, an optional parameter list provides control over the filter function, and may include the following parameters: `"width"`, the amount to "overfilter" in `s`; `"filter"`, the name of the filter kernel to apply. The filter may be any of the following: `"catmull-rom"` (the default), `"box"`, `"triangle"`, or `"gaussian"`.

5.8.8 Grid-wise Reduction Operators and Grid-Aware Functions

Although shaders are usually written as if one point on the surface is being shaded in isolation, generally many points (called a *grid*) are being shaded simultaneously. Gelato provides a few functions that give information about the grid dimensions or that reduce all the point-by-point values of a variable into a single value for the entire grid.

`float` **`gridn`**`(),` **`gridnu`**`(),` **`gridnv`**`()`
`float` **`gridindex`**`()`

Returns information about the dimensions of the grid being shaded. `gridn()` returns the total number of points in the grid. `gridnu()` and `gridnv()` return the number of grid points in the u and v directions, respectively (assuming a rectangular grid). `gridindex()` is the index of the point within the grid (on the range $[0..n-1]$). Caveat: shading grids are not guaranteed to be rectangular, so it is possible that $nu \times nv \neq n$.

`float` **`gridany`** `(`*`type`*` x)`

Returns 1 if the value of x is nonzero *anywhere* on the grid of points being shaded simultaneously, returns 0 only if x is zero at all of the points being shaded simultaneously. If x is a string, then `gridany()` returns 1 if the string is nonempty (i.e., not `""`) anywhere on the grid. If x is a boolean value (like the result of < or ==, then `gridany()` returns 1 if the condition is true anywhere on the grid.

`type` **`gridmin`** `(`*`type`*` x)`
`type` **`gridmax`** `(`*`type`*` x)`

Returns the single minimum or maximum, respectively, of the value of x across the entire grid of points being simultaneously shaded. The variants that operate on colors or point-like objects operate on a component-by-component basis (i.e., separately for x, y, and z).

5.8.9 String Functions

`string` **`format`** `(string template, ...)`

The `format` function takes a template and an argument list, and returns a formatted string (much like C's `printf` function). Where the template contains the characters `%f`, `%c`, `%p`, `%m`, and `%s`, `printf` will substitute arguments, in order, from the argument list (assuming

that the argument types are `float`, `color`, point-like, `matrix`, and `string`, respectively). In addition, `%d` and `%i` will also print `float`'s, truncating and printing them as if they were integers.

`format()` is much like C's `sprintf()` function, except that it returns the formatted string, rather than supplying its address as an argument.

`void` **`printf`** `(string template, ...)`

Much as in C, `printf` takes a template and an argument list, and prints the resulting formatted string to the console. Where the template contains the characters `%f`, `%c`, `%p`, `%m`, and `%s`, `printf` will substitute arguments, in order, from the argument list (assuming that the argument types are `float`, `color`, point-like, `matrix`, and `string`, respectively). In addition, `%d` and `%i` will also print `float`'s, truncating and printing them as if they were integers.

`void` **`error`** `(string template, ...)`

The `error` function is just like `printf`, but is printed as a renderer error message, including information about the name of the shader and the object being shaded.

`void` **`fprintf`** `(string filename, string template, ...)`

The `fprintf` function is just like `printf`, but it concatenates the output onto the end of the text file with the given `filename`.

`string` **`concat`** `(string s1, ..., string sN)`

Concatenates a list of strings, returning the aggregate string.

`float` **`match`** `(string pattern, string subject)`

Does a string pattern match on `subject`. Returns 1 if the `pattern` exists anywhere within `subject` and 0 if the `pattern` does not exist within `subject`. The `pattern` can be a standard Unix regular expression. Note that the pattern does not need to start in the first character of the `subject` string, unless the pattern begins with the ^ (beginning of string) character.

`string` **`substr`** `(string s, float start, float length)`
`string` **`substr`** `(string s, float start)`

Return *length* characters from *s*, starting with the character indexed by *start* (indexing, like in C, begins with 0). If *length* is omitted, return the rest of *s*, starting with *start*. If *start* is negative, it counts backwards from the end of the string (for example, `substr(s,-1)` returns just the last character of *s*).

5.8.10 Texture, Reflection, and Shadow Access Functions

```
type texture (string filename, float s, float t, ...params...)
type texture (string filename, float s, float t,
         float dsdx, float dtdx, float dsdy, float dtdy, ...params...)
```

Perform an antialiased lookup into an image file, indexed by 2D coordinates. The lookup is centered at (s,t) and the differentials *dsdx, dtdx, dsty, dtdy* define a region to filter over for antialiasing; if the differentials are not supplied, they will be automatically computed by taking derivatives of *s* and *t*.

The `texture()` function may perform either a single-channel lookup (returning a `float`) or a three-channel lookup (returning a `color`), depending on what kind of variable the results are assigned to (or whether an explicit type cast is made).

The 2D lookup coordinate(s) may be followed by optional token/value arguments that control the behavior of `texture()`:

"blur", <float>

Specifies additional blur when looking up the texture value. The blur amount is relative to the size of the texture (i.e., 0.5 blurs by a kernel half the width and height of the texture). The default value is 0.

The blur may be specified separately in the *s* and *t* directions by using the "sblur" and "tblur" parameters, respectively.

"width", <float>

Scales the size of the filter specified by coordinates, or the filter area that the renderer estimates from examining the derivatives of the coordinate. The default value is 1.

The width value may be specified separately in the *s* and *t* directions by using the "swidth" and "twidth" parameters, respectively.

"wrap", <string>

Overrides the texture file's wrap mode with one of: "black", "periodic", "clamp", "mirror". The default value is "default", which indicates that the renderer should use the default wrap modes that are in the texture file itself.

The wrap modes may be specified separately in the *s* and *t* directions by using the "swrap" and "twrap" parameters, respectively.

"firstchannel", <float>

Specifies the first channel to look up from the texture map. The default is 0.

"fill", <float>

Specifies the value to return for any channels that are not present in the texture file. The default is 0.

"alpha", <floatvariable>

The alpha channel (i.e., the next channel after the channels returned by the function call) will be stored in the variable specified. This allows for RGBA lookups in a single call to `texture()`.

type **texture3d** (string filename, point p, *...params...*)

Perform an antialiased lookup into a volume image file, indexed by 3D coordinates. The version with a single 3D *p* coordinate automatically examines how *p* varies over the surface in order to antialias the lookup.

The `texture3d()` function may perform either a single-channel lookup (returning a `float`) or a three-channel lookup (returning a `color`), depending on what kind of variable the results are assigned to (or whether an explicit type cast is made).

The 3D lookup coordinate(s) may be followed by optional token/value arguments that control the behavior of `texture3d`. All of the optional arguments to `texture()` are also respected in analogous ways for `texture3d`.

type **environment** (string name, vector R, *...params...*)
type **environment** (string name, point P, vector R, *...params...*)
type **environment** (string filename, vector R,
    vector dRdx, vector dRdy, *...params...*)
type **environment** (string filename, point P, vector R,
    vector dRdx, vector dRdy, *...params...*)

Computes the environmental reflections from position *P* in the direction *R*. If the optional *P* is not supplied, it is assumed to be the surface position `P`. The optional differential vectors *dRdx* and *dRdy* define a solid angle to filter over for antialiasing; if not supplied, they are automatically generated by taking derivatives of *R*.

If *name* is the name of an environment map file, that environment map will be used for the reflections. If *name* is the name of a geometry set (see Section 4.4.3), Gelato will compute the reflection by ray tracing. Thus, a single `environment()` call may perform either environment mapping or ray tracing, depending on the name of the map (which can be passed as an argument to the shader). Note that when ray tracing, `environment()` will only "see" objects that are in the named geometry set (with `Attribute ("geometryset",...)`).

The `environment()` function may perform either a single-channel lookup (returning a `float`) or a three-channel lookup (returning a `color`), depending on what kind of variable the results are assigned to (or whether an explicit type cast is made).

The directional coordinate(s) may be followed by optional token/value arguments that control the behavior of `environment()`:

"angle", <float>

Specifies additional blur when looking up the environment value, expressed as the half-angle of the lookup cone, in radians. The "angle" supercedes any "blur"

value that may also be given. If neither "angle" nor "blur" values are given, no additional blur will be supplied (i.e., reflections will be sharp).

"blur", <float>

Specifies additional blur when looking up the environment value. The default value is 0. A value of 1 blurs 90 degrees. Note that if the newer "angle" parameter is given, it will override any "blur" value supplied.

The blur may be specified separately in the *s* and *t* directions by using the "sblur" and "tblur" parameters, respectively.

"width", <float>

Scales the size of the filter specified by coordinates, or the filter area that the renderer estimates from examining the derivatives of the coordinate. The default value is 1.

The width value may be specified separately in the *s* and *t* directions by using the "swidth" and "twidth" parameters, respectively.

"firstchannel", <float>

Specifies the first channel to look up from the texture map. The default is 0.

"fill", <float>

Specifies the value to return for any channels that are not present in the texture file. The default is 0.

"Kr", <float>

Is a hint about how the results of environment() will be scaled, which can allow environment() to sample selectively and especially to eliminate work if the eventual multiplier is 0. Note that this is just a hint, and does not in itself cause the results to be scaled.

"space", <string>

Specifies the name of the coordinate system in which the environment map was created. The P and R parameters will be automatically transformed to this space for the map lookup. If no "space" parameter is specified, no transformation will be performed (i.e., "common" space is assumed, which is almost always incorrect for environment maps). This parameter is ignored for ray tracing, which is always performed in "common" space.

"radius", <float>

Specifies a radius for the environment map, allowing some (fake, but convincing) parallax to be incorporated into the environment lookup. If no "radius" parameter is supplied, or its value is 0, parallax will not be considered. This parameter is ignored for ray tracing.

"up", <string>

NEW!

For lookups into lat-long environment maps, a value of "y" or "z" forces the map to be interpreted in a "$+y$ is up" or "$+z$ is up" convention, respectively. If this option is not supplied or is the empty string, the convention will be determined by the map itself (which has a default orientation). This option has no effect on cube-face environment maps or ray traced reflections.

"samples", <float>

For ray tracing, specifies how many sample rays to average to give the final result. This parameter is ignored for environment map lookups.

"bias", <float>

Ignores ray hits closer than this value in order to prevent incorrect self-reflection of surfaces. If no "bias" is supplied, or if its value is less than 0, the global shadow bias will be used. This parameter is ignored for environment map lookups.

"maxhitdist", <float>

Ignores ray hits farther than this value (measured in "common" space units). If no "maxhitdist" is supplied or if the value is ≤ 0, there is no maximum hit distance. This parameter is ignored for environment map lookups.

"shade", <float>

If the argument is zero and the environment() call is tracing rays, the object hit will not be shaded. This can be useful if you are casing rays only to detect if something is hit, or the distance to hit objects, but do not need tthe fully-shaded color of the hit object, and therefore would like to avoid the cost of shading it. The default is to shade objects that are hit by rays. This parameter is ignored for environment map lookups.

"alpha", <floatvariable>

The alpha channel will be stored in the variable specified. This allows for RGBA lookups in a single call to environment(). For environment map lookups, the alpha channel is assumed to be the next channel after the channels returned by the function call.

For ray tracing, the alpha returned is the average opacity of the rays traced to determine the reflection. An alpha value of 0 indicates that the sample rays hit no objects in the scene.

"hitdist", <floatvariable>

For ray tracing, the value stored in the parameter variable is the average distance to hits for all the sample rays that actually hit objects (or 1e38 if none of the sample rays hit any objects). For environment maps, the value stored will always be 1e38.

"Phit", <pointvariable>

For ray tracing, the value stored in the parameter variable is the average position of hits for all the sample rays that actually hit objects. For environment map lookups, the variable is unchanged.

"Nhit", <vectorvariable>

For ray tracing, the value stored in the parameter variable is the average normal at the hit points for all the sample rays that actually hit objects. For environment map lookups, the variable is unchanged.

```
type shadow (string name, point P, ...params... )
type shadow (string shadowname, point P,
        vector dPdx, vector dPdy, ...params...)
```

Calculate shadows, returning the amount of occlusion at point *P* (0 means unoccluded, 1 means completely occluded). Differential vectors *dPdx* and *dPdy* define a region to filter over for antialiasing; if not supplied, they are computed automatically from *P*.

If *name* is the name of a shadow map file, shadow depth mapping will be used to compute the occlusion (the source position of the light is stored in the shadow map itself). If *name* is the name of a geometry set (see Section 4.4.3), Gelato computes the occlusion by ray tracing from *P* to the light's position (as passed to the surrounding `emit` statement). Thus, a single `shadow()` call may perform either shadow mapping or ray tracing, depending on the name of the map (which can be passed as an argument to the shader). Note that when ray tracing, `shadow()` will only "see" objects that are in the named geometry set (with `Attribute ("geometryset",...)`).

The `shadow()` function may return either a `float` or a `color`, depending on what kind of variable the results are assigned to (or whether an explicit type cast is made). When performing a shadow map lookup into a single-channel depth map, the `color shadow()` function replicates the shadow value in all three channels. When performing a multi-channel shadow lookup (including ray traced shadow probes), the `float shadow()` function uses just the red channel. We therefore recommend always using the `color shadow()` function, which performs as expected for both ray tracing as well as shadow map lookups.

The positional coordinate(s) may be followed by optional token/value arguments that control the behavior of the `shadow()` function:

"blur", <float>

Specifies Additional blur when looking up the shadow value. The default value is 0 (sharp shadows). For maps, the amount is expressed as a fraction of the shadow map view angle; for ray tracing, it is scaled so that a blur of 1.0 indicates a sampling angle of 90 degrees.

The blur may be specified separately in the *s* and *t* directions by using the `"sblur"` and `"tblur"` parameters, respectively.

`"width", <float>`

Scales the size of the filter specified by coordinates, or the filter area that the renderer estimates from examining the derivatives of the coordinates. The default value is 1. The width value may be specified separately in the *s* and *t* directions by using the `"swidth"` and `"twidth"` parameters, respectively.

`"samples", <float>`

Specifies how many samples (or number of rays, when ray tracing) are used to compute the shadow lookup. For shadow map lookups (but not for ray traced shadows), a minimum of 16 samples are taken.

`"bias", <float>`

Specifies the amount to add to lookups in order to prevent incorrect self-shadowing of surfaces. For values less than 0, the global bias (`Attribute "shadow:bias"`) is used instead. For ray tracing, this causes ray hits closer than the bias distance to be ignored.

5.8.11 Lighting Functions

`color` **`ambient`** `()`

Returns the contribution of *ambient* light sources. Ambient lights come from no specific location or direction, and thus their shaders have no `emit` statement.

`color` **`diffuse`** `(vector N)`

Calculates light reflected using Lambert's law:

$$\sum_{i=1}^{\text{nlights}} Cl_i \max(0, N \cdot L_i)$$

where for each of the non-ambient light sources (any light source whose light shader contains an `emit` statement is *non-ambient*), L_i is the unit vector pointing toward the light, Cl_i is the light color, and N is the unit normal of the surface. The max function ensures that lights with $N \cdot L_i < 0$ (i.e., those *behind* the surface) do not contribute to the calculation. The `N` vector is assumed to be of unit length.

For lights that have a parameter (or `output` parameter) called `__nondiffuse`, the `diffuse()` function scales the contribution of that light by `(1-__nondiffuse)`.

`color` **`specular`** `(vector N, vector V, float roughness)`

Computes *specular* highlights resulting from the narrow scattering of light off the surface, summing the contribution of all non-ambient light sources (any light source whose light shader contains an `emit` statement is *non-ambient*). The `N` and `V` vectors are assumed to be of unit length.

For lights that have a parameter (or `output` parameter) called `__nonspecular`, the `specular()` function scales the contribution of that light by `(1-__nonspecular)`.

`color` **`specularbrdf`** `(vector L, vector N, vector V, float roughness)`

Computes the attenuation of light coming from `L` and bouncing off a surface with normal `N` and given `roughness`, as viewed from direction `V`. All of `L`, `N`, and `V` are assumed to be of unit length.

`color` **`indirect`** `(string name, point Pos, vector R, ...)`

Use ray tracing to probe the geometry set specified by *name* at position *Pos* in the hemisphere centered around *R*. The return value is the total irradiance gathered over the hemisphere. The `indirect()` function will only "see" objects that are in the named geometry set (see Section 4.4.3).

The irradiance information is computed sparsely and interpolated. As samples in this "irradiance cache" are computed, they are stored in a spatial database. It is possible to

save the database to disk (“bake” them), or read a database from disk to “seed” the in-memory database (including a read-only mode where no new values will be computed).

The interpolation and use of the irradiance cache is controlled by several attributes described in Section 4.4.10, as well as the following optional arguments:

`"samples", <float>`

Specifies how many rays to use to sample the hemisphere. The default is 1, which is woefully inadequate.

`"adaptive", <float>`

Controls adaptive sampling of the hemisphere. If $adaptive = 0$, the full *samples* will be used. For nonzero values, fewer than *samples* rays may be cast in some areas. The default value is 1, with larger values more aggressively reducing samples in some areas, and smaller values tending to use the full *samples* more of the time.

`"maxerror", <float>`

A maximum error metric for interpolation of irradiance samples. Smaller numbers result in higher quality, but longer rendering times. A value of 0 will cause full sampling to happen for every `indirect()` call. If no `"maxerror"` parameter is passed, or its value is negative, *maxerror* will take its value from the `"indirect:maxerror"` Attribute (see Section 4.4.10).

`"maxpixeldist", <float>`

The maximum distance allowed (measured in pixels) between irradiance samples. Smaller numbers result in higher quality, but longer rendering times. A value of 0 will cause full sampling to happen for every `indirect()` call. If no `"maxerror"` parameter is passed, or its value is negative, *maxpixeldist* will take its value from the `"indirect:maxpixeldist"` Attribute (see Section 4.4.10).

`"bias", <float>`

Ignores ray hits closer than this value in order to prevent incorrect self-reflection of surfaces. If no `"bias"` is supplied, or if its value is less than 0, the global shadow bias will be used.

`"maxhitdist", <float>`

Ignores ray hits farther than this value (measured in `"common"` space units). If no `"maxhitdist"` is supplied or if the value is ≤ 0, there is no maximum hit distance.

`"falloff", <float>`
`"falloffmode", <float>`

Controls how distance of objects affects the amount of indirect illumination. If $falloff = 0$ (the default), there is no distance falloff of indirect radiance (this is the physically correct behavior). If $falloff > 0$ and $falloffmode = 0$, the influence of indirect illumination will fall off as $e^{-falloff/dist}$. If $falloff > 0$ and $falloffmode = 1$, indirect illumination will fall off as $(1 - dist/maxhitdist)^{falloff}$.

"filemode", <string>

Specifies the file operations that will be performed with the indirect spatial database, augmenting or overriding the various "spatialdb" Attributes. Choices for the string value include:

"r" Reads the database from a disk file, if such a file exists and has not already been read, even if there was never a Attribute ("spatialdb:read") for this database.

"w" Marks the database to be written to disk after rendering, even if there was never an Attribute("spatialdb:write") command for that file.

"rw" Reads the database from disk (if not already done), and marks it to be written to disk after rendering, even if neither option was specified as an Attribute.

"ro" Forces a read of the database from disk (if not already performed), designates the database as not to be written to disk, and forces all database queries to succeed so that new entries are not formed. This is equivalent to the "spatialdb:readonly" attribute.

"wo" Sets "write-only" mode, in which entries are written to disk as they are added to the database, but the in-memory database does not retain the entries, thus keeping extremely low memory usage. This is deal for a pass in which you are "baking" data to a spatial database on disk, but have no need to use the data on the current rendering pass.

"" An empty string indicates the default action, which is to obey the various "spatialdb" attributes or any previous file modes in effect for this database.

The file mode overwrites any previously-specified file mode for this database, so beware of giving contradictory modes to the same database at different times during a single render.

```
float occlusion (string name, point Pos, vector R, float angle,
                 [output vector Nunocc,] ...)
```

Use ray tracing to probe the geometry set specified by *name* with a cone of rays having apex *Pos*, in direction *R* and with half-angle *angle*. The return value is the portion of the solid angle that is occluded (1 if all sample rays hit objects, 0 if all sample rays miss all objects). If the optional *Nunocc* is supplied, it will have its value replaced by the average direction of the probe rays that were not occluded. The occlusion() function will only "see" objects that are in the named geometry set (see Section 4.4.3).

The occlusion information is computed sparsely and interpolated, much like indirect illumination. As the samples in this "occlusion cache" are computed, they are stored in a spatial database. It is possible to save the database to disk ("bake" them), or read a database from disk to "seed" the in-memory database (including a read-only mode where no new values will be computed).

The interpolation and use of the occlusion cache are controlled by several attributes described in Section 4.4.11, as well as the following optional arguments:

`"samples", <float>`

Specifies how many rays to use to sample the hemisphere. The default is 1, which is woefully inadequate.

`"adaptive", <float>`

Controls adaptive sampling of the hemisphere. If $adaptive = 0$, the full *samples* will be used. For nonzero values, fewer than *samples* rays may be cast in some areas. The default value is 1, with larger values more aggressively reducing samples in some areas, and smaller values tending to use the full *samples* more of the time.

`"maxerror", <float>`

A maximum error metric for interpolation of occlusion query samples. Smaller numbers result in higher quality, but longer rendering times. A value of 0 will cause full occlusion sampling to happen for every `occlusion()` call. If no `"maxerror"` parameter is passed, or its value is negative, *maxerror* will take its value from the `"occlusion:maxerror"` Attribute (see Section 4.4.11).

`"maxpixeldist", <float>`

The maximum distance allowed (measured in pixels) between occlusion query samples. Smaller numbers result in higher quality, but longer rendering times. A value of 0 will cause full occlusion sampling to happen for every `occlusion()` call. If no `"maxerror"` parameter is passed, or its value is negative, *maxpixeldist* will take its value from the `"occlusion:maxpixeldist"` Attribute (see Section 4.4.11).

`"bias", <float>`

Ignores ray hits closer than this value in order to prevent incorrect self-reflection of surfaces. If no `"bias"` is supplied, or if its value is less than 0, the global shadow bias will be used.

`"maxhitdist", <float>`

Ignores ray hits farther than this value (measured in `"common"` space units). If no `"maxhitdist"` is supplied or if the value is ≤ 0, there is no maximum hit distance.

`"falloff", <float>`
`"falloffmode", <float>`

Controls how distance of occluding objects affects the amount of occlusion. If $falloff = 0$ (the default), all ray hits are "equally occluding." If $falloff > 0$ and $falloffmode = 0$, the amount of occlusion will be $e^{-falloff/dist}$. If $falloff > 0$ and $falloffmode = 1$, the amount of occlusion will be $(1 - dist/maxhitdist)^{falloff}$.

"filemode", <string>

Specifies the file operations that will be performed with the occlusion spatial database, augmenting or overriding the various "spatialdb" Attributes. Choices for the string value include:

"r" Reads the database from a disk file, if such a file exists and has not already been read, even if there was never a Attribute ("spatialdb:read") for this database.

"w" Marks the database to be written to disk after rendering, even if there was never an Attribute("spatialdb:write") command for that file.

"rw" Reads the database from disk (if not already done), and marks it to be written to disk after rendering, even if neither option was specified as an Attribute.

"ro" Forces a read of the database from disk (if not already performed), designates the database as not to be written to disk, and forces all database queries to succeed so that new entries are not formed. This is equivalent to the "spatialdb:readonly" attribute.

"wo" Sets "write-only" mode, in which entries are written to disk as they are added to the database, but the in-memory database does not retain the entries, thus keeping extremely low memory usage. This is deal for a pass in which you are "baking" data to a spatial database on disk, but have no need to use the data on the current rendering pass.

"" An empty string indicates the default action, which is to obey the various "spatialdb" attributes or any previous file modes in effect for this database.

The file mode overwrites any previously-specified file mode for this database, so beware of giving contradictory modes to the same database at different times during a single render.

color **subsurface** (string dbname, point p, normal n, ...)

Returns the illumination due to subsurface scattering. Optional arguments set the material properties. This function depends on the presence of a spatial database (named by *dbname*) that contains baked diffuse illumination, scaled by surface area. The database file will automatically be read from disk; there is no need for a separate Attribute ("spatialdb:read") command in the scene. It is important that p and n are in the same coordinate system as the values that were baked into the spatial database.

Optional token/value arguments may follow n, and may include any of the following:

"scattering", <color>

The reduced scattering coefficient (σ_s') of the material. The default is (2.19, 2.62, 3.0), which is a measured σ_s' for marble, in units of mm^{-1}. Higher numbers indicate more scattering (changes in the direction of light through the material), lower numbers indicate less scattering within the material. All components should be ≥ 0.

"absorption", <color>

The absorption coefficient (σ_a) of the material. The default is `(.0021, .0041, .0071)`, which is a measured σ_a for marble, in units of mm^{-1}. Higher numbers indicate that light is absorbed quickly, lower numbers indicate that light travels quite far through the material. All components should be ≥ 0.

"eta", <float>

The index of refraction (η) of the material. The default is 1.5, which is a measured η for marble. The value should be ≥ 1.

"maxsolidangle", <float>

An error metric for reconstructing the subsurface illumination. The default is 1. Larger values render faster, but less accurately; smaller values render more accurately, but slower. The value should be > 0.

5.8.12 Displacement Functions

void **displace** (float amp)
void **displace** (string space, float amp)

Displace the surface in the direction of the surface normal `N` by *amp* units, as measured in the named *space*, or in `"common"` space (if no *space* is specified). This function properly accounts for interpolated normals on faceted geometry, ensuring that the surface still looks smooth after displacement.

void **displace** (vector offset)

A more general version of the above, displace the surface by an arbitrary vector *offset* (which does not have to be in the direction of `N`).

void **bump** (float amp)
void **bump** (string space, float amp)
void **bump** (vector offset)

These routines are analogous to `displace()`, except that they only modify `N` as if the surface was displaced, but without actually moving the surface `P`.

5.8.13 Spatial Databases

Shaders may enter arbitrary data into sparse spatial databases, and look up that data later (interpolating between nearby samples, within error tolerances).

A variety of additional attributes controlling the behavior of spatial databases are documented in Section 4.3.9.

void **spatialdbsave** (string name, float s, float t, *type* data, ...)
void **spatialdbsave** (string name, point p, *type* data, ...)
void **spatialdbsave** (string name, point p, normal n, *type* data, ...)

Adds a new record to the named spatial database, indexed by a 2-D coordinate (s,t), a 3D position (p), or a 3D position and normal direction (p,n). The type of *data* may be `float`, `color`, `point`, `vector`, or `normal`.

Optional token/value arguments may follow *data*, and may include any of the following:

`"interpolate", <float>`

If zero, the default, a spatial database entry will be written for every shading vertex. If nonzero, a spatial database entry will be written at the center of shaded facet (averaging the four vertex values for position, normal, and data for that quadrilateral facet). Using interpolation in this manner can help avoid duplicate spatial database entries on grid edges and corners, which would happen due to abutting grids on the other side of the edge. Caveat: don't call `spatialdbsave` with `"interpolate"` 1 from inside a "varying if" – it can't properly interpolate if the values it's interpolating aren't well-defined everywhere.

`"filemode", <string>`

Specifies the file operations that will be performed with the spatial database, augmenting or overriding the various `"spatialdb"` Attributes. Choices for the string value include:

- `"w"` Marks the database to be written to disk after rendering, even if there was never an `Attribute("spatialdb:write")` command for that file.
- `"rw"` Reads the database from disk (if not already done), and marks it to be written to disk after rendering, even if neither option was specified as an `Attribute`.
- `"wo"` Sets "write-only" mode, in which entries are written to disk as they are added to the database, but the in-memory database does not retain the entries, thus keeping extremely low memory usage. This is deal for a pass in which you are "baking" data to a spatial database on disk, but have no need to use the data on the current rendering pass.
- `""` An empty string indicates the default action, which is to obey the various `"spatialdb"` attributes or any previous file modes in effect for this database.

`"resolution", <float[2]>`

If a 2D spatial database is selected to be written to disk (file mode `"w"`, `"rw"`, or `"wo"`, or the equivalent `Attribute`), and the file name has an extension that corresponds to an ImageIO plugin, the SDB will be filtered and written as a 2D image file. In this case, the optional `"resolution"` parameter, which is passed a `float[2]`, specifies the resolution of the image file.

`"filedataformat", <string>`

If a 2D spatial database is selected to be written to disk (file mode `"w"`, `"rw"`, or `"wo"`, or the equivalent `Attribute`), and the file name has an extension that corresponds to an ImageIO plugin, the SDB will be filtered and written as a 2D image file. In this case, the optional `"filedataformat"` parameter, which is passed a `string`,

specifies the data format used for the file. The value may be one of: "8" (8-bit integer), "16" (16-bit integer), "half" (16-bit float), or "float" (32-bit float). If the requested data format is not supported by the image file format, a format-specific default will be used instead.

"writefilterwidth", <float>

If a 2D spatial database is written as a 2D image (i.e., the file name has an extension that corresponds to an ImageIO plugin), the SDB will be filtered using a filter that is 2 pixels wide in the baked texture image but may grow up to this width in areas with low sample density. The default is 2 pixels, which means a uniform pixel width everywhere. Smaller values have no effect – the minimum filter size is 2 pixels. A larger "writefilterwidth" can help to avoid gaps the baked image in areas of low sampling density, but may result in the baked image having a wider "border" of blurry extrapolated values outside the true parameter range of the samples.

```
float spatialdbquery (string name, float s, float t, output type data, ...)
float spatialdbquery (string name, point p, output type data, ...)
float spatialdbquery (string name, point p, normal n, output type data, ...)
```

Queries the named spatial database, indexing by a 2-D coordinate (s,t), a 3D position (p), or a 3D position and normal direction (p,n). The type of *data* may be float, color, point, vector, or normal. If it can do so within error tolerances, an interpolated value from nearby entries in the database will be written to *data* and spatialdbquery will return 1. If there are not enough nearby samples to safely interpolate, *data* will not be modified, and the return value will be 0.

Optional token/value arguments may follow *data*, and may include any of the following:

"maxerror", <float>

Sets error tolerances. The default is 0.25. Larger numbers will allow interpolation of more distant samples.

"maxdist", <float>

Sets the maximum distance that a sample in the database can be from p in order to be considered. The default is 1. Larger numbers will allow interpolation of more distant samples, which may be faster, but may result in a "blurrier" lookup from the database.

"minsamples", <float>

Sets the minimum number of nearby samples required to interpolate a value. The default is 3. Larger numbers may lead to higher quality interpolation but be more expensive, smaller numbers may execute faster but may have blocky artifacts.

"filter", <string>

The default interpolation method (if no filter is specified, or if the filter is "value") is to interpolate the values in the spatial database. If the filter name is "areadensity", the value returned will instead be a value-weighted *density estimate* of the values in the spatial database. Area densities are used for photon maps.

"filemode", <string>

Specifies the file operations that will be performed with the spatial database, augmenting or overriding the various "spatialdb" Attributes. Choices for the string value include:

"r" Reads the database from a disk file, if such a file exists and has not already been read, even if there was never a Attribute ("spatialdb:read") for this database.

"rw" Reads the database from disk (if not already done), and marks it to be written to disk after rendering, even if neither option was specified as an Attribute.

"ro" Forces a read of the database from disk (if not already performed) and forces all database queries to succeed so that new entries will not need to be created. This is equivalent to the "spatialdb:readonly" attribute.

"" An empty string indicates the default action, which is to obey the various "spatialdb" attributes or any previous file modes in effect for this database.

5.8.14 Renderer State Queries

float **getattribute** (string name, output *type* destination)

Retrieve the renderer attribute with the given *name* (i.e., any of the attributes described for Attribute or GetAttribute in Section 2.4, including user attributes). If the attribute exists and has the same type as *destination*, the value of the attribute will be stored in *destination* and the getattribute() function will return 1.0. If the name is not recognized or if its type does not match that of *destination*, then *destination* will not be modified and getattribute() will return 0. Since GSL does not have an integer type, you may use float variables to retrieve int attributes, and the appropriate type conversions will happen automatically.

void **setmessage** (string messagename, *type* value)

Stores data (a "message") in an area where it can later be retrieved by name by other shaders attached to the same object. Existing messages with the identical name will be replaced.

```
float getmessage (string sourcename, string messagename,
                  output type destination)
```

Retrieve data (a "message") from another shader attached to the same object. The source of the message is determined by the *sourcename* parameter, which may be one of the following:

`"surface", "displacement", "volume"`

The message is retrieved from the shader of that type that is present on the same object being shaded.

`"light"`

If the `getmessage()` call is within a `lights` loop, the message is retrieved from the active light source being examined in that particular iteration of the `lights` loop.

`"parent"`

The message will be retrieved from the shader (potentially on another object) that spawned the ray that is viewing this object.

`"trace"`

If the `getmessage()` call is within a `trace` loop, the message is retrieved from the closest object hit by the ray. Also, `getmessage("trace",...)` may be used to get information about the ray sample itself (see Section 5.5.5).

`"primitive"`

The value will be retrieved by interpolating a user variable attached to the geometric primitive itself (regardless of whether it is also a name of a shader parameter). NEW!

If *messagename* is the name of a message that was set by a call to `setmessage` by the given source shader, and the type of *destination* matches the type of the message, then the message's value will be stored in *destination* and `getmessage()` will return `1`.

If no matching message was set via `setmessage`, then the parameters (including `output` parameters) of the source shader will be searched. If *messagename* is the name of a parameter of the given shader and has the same type as *destination*, the value of the shader's parameter will be stored in *destination* and the `getmessage()` function will return `1`.

If the name (with matching type) is not found in either the set messages or the parameter list of the source shader, then then *destination* will not be modified and `getmessage()` will return `0`.

If the shader named by *sourcename* is actually a series of shaders (as specified by `ShaderGroupBegin` and `ShaderGroupEnd`), then the renderer will attempt to match and retrieve the message starting with the last executed shader in the sequence, proceeding to the first until a match is found.

```
float gettextureinfo (string texturename, string paramname,
                      output type destination)
```

Returns information about a particular stored texture file. If the file exists and the parameter is recognized and has the same type as *destination*, its value will be stored in *destination* and the `gettextureinfo()` function will return `1.0`. If the file does not exist, or the name is not recognized, or if its type does not match that of *destination*, then *destination* will not be modified and `gettextureinfo()` will return `0`. Table 5.5 lists the data names that Gelato understands.

Name	Type	Description
`"resolution"`	`float` [2]	The resolution of the highest resolution version of the image stored in the texture map.
`"type"`	`string`	Returns one of: `"texture"`, `"shadow"`, `"environment"`, `"volume"`.
`"textureformat"`	`string`	Returns one of: `"Plain Texture"`, `"Shadow"`, `"CubeFace Shadow"`, `"Volume Shadow"`, `"CubeFace Environment"`, `"LatLong Environment"`, `"Volume Texture"`. Note that this differs from `"type"` in that it specifically distinguishes between the different types of shadows and environment maps.
`"channels"`	`float`	The number of channels in the texture map.
`"viewingmatrix"`	`matrix`	(Shadow maps only) The matrix that transforms points from `"common"` space to the `"camera"` space from which the texture was created.
`"projectionmatrix"`	`matrix`	(Shadow maps only) The matrix that transforms points from `"common"` space to a 2D coordinate system where *x* and *y* range from -1 to 1.

Table 5.5: Names known to the `gettextureinfo` function.

```
float raylevel ( )
```

Returns `0` if the shader is computing the appearance of a surface directly visible from the camera, `1` if it is calculating the appearance of a one-bounce reflection ray, `2` if a two-bounce reflection ray, etc.

```
float isshadowray ( )
```

Returns `1` if the shader is being run to evaluate the opacity of an object for the purpose of a ray-traced shadow, otherwise returns `0`.

Note that shadow rays have the same ray level as the grid that spawned them. For example, the ray-traced shadows of a camera grid have raylevel 0, but `isshadowray()` will return 1.

`float` **`isindirectray`** `( )`

Returns `1` if the shader is being run to evaluate indirect illumination, otherwise returns `0`.

5.8.15 Miscellaneous Functions

This section documents miscellaneous built-in functions that defy easy categorization.

`float` **`arraylength`** `(`*type* `A[])`

Returns the length (number of elements) of the referenced array.

NEW!

`void` **`exit`** `()`

Exists the shader, without executing any more code (for the shader point currently being run), even if the `exit()` call is within a conditional, loop, or function body. In the main body of a shader, `exit()` is equivalent to `return`. But within a function body, `return` exits only the function, so `exit()` is provided to terminate execution of the entire shader.

5.9 Formal Language Syntax

This section gives the complete syntax of Gelato's shading language, in a BNF-like format. Note that any syntactical structures that have a name ending in "-opt" are optional. Structures surrounded by curly braces { } may be repeated 0 or more times. Text in `typewriter` face indicates literal text. The ε character is used to indicate that it is acceptable for there to be no token.

5.9.1 Lexical Elements

⟨*digit-sequence*⟩ ::= ⟨*digit*⟩ { ⟨*digit*⟩ }

⟨*integer*⟩ ::= ⟨*sign*⟩ ⟨*digit-sequence*⟩

⟨*floating-point*⟩ ::= ⟨*digit-sequence*⟩ ⟨*decimal-part-opt*⟩ ⟨*exponent-opt*⟩
| ⟨*decimal-part*⟩ ⟨*exponent-opt*⟩

⟨*decimal-part*⟩ ::= '`.`' { ⟨*digit*⟩ }

⟨*exponent*⟩ ::= '`e`' ⟨*sign*⟩ ⟨*digit-sequence*⟩

⟨*sign*⟩ ::= '`-`' | '`+`' | ε

⟨*number*⟩ ::= ⟨*integer*⟩
| ⟨*floating-point*⟩

⟨*char-sequence*⟩ ::= { ⟨*any-char*⟩ }

⟨*stringliteral*⟩ ::= '`"`' ⟨*char-sequence*⟩ '`"`'

⟨*identifier*⟩ ::= ⟨*letter-or-underscore*⟩ { ⟨*letter-or-underscore-or-digit*⟩ }

5.9.2 Overall Structure

⟨*shader-file*⟩ ::= ⟨*function-declarations-opt*⟩ ⟨*shader-declaration*⟩

⟨*function-declarations*⟩ ::= ⟨*function-declaration*⟩
| ⟨*function-declarations*⟩ ⟨*function-declaration*⟩

⟨*shader-declaration*⟩ ::=
⟨*shadertype*⟩ ⟨*identifier*⟩ ⟨*metadata-opt*⟩ `(` ⟨*shader-formal-params-opt*⟩ `)` `{` ⟨*statement-list*⟩
`}`

⟨*shadertype*⟩ ::= `surface` | `displacement` | `light` | `volume` | `shader`

⟨*shader-formal-params*⟩ ::= ⟨*shader-formal-param*⟩
| ⟨*shader-formal-params*⟩ `,` ⟨*shader-formal-param*⟩

⟨*shader-formal-param*⟩ ::= ⟨*outputspec*⟩ ⟨*typename*⟩ ⟨*identifier*⟩ ⟨*initializer*⟩ ⟨*metadata-opt*⟩
  | ⟨*outputspec*⟩ ⟨*typename*⟩ ⟨*identifier*⟩ ⟨*arrayspec*⟩ ⟨*array-initializer*⟩ ⟨*metadata-block-opt*⟩

⟨*metadata-block*⟩ ::= `[[` ⟨*metadata*⟩ `]]`

⟨*metadata*⟩ ::= ⟨*metadatum*⟩
  | ⟨*metadata*⟩ `,` ⟨*metadatum*⟩

⟨*metadatum*⟩ ::= ⟨*typename*⟩ ⟨*identifier*⟩ ⟨*initializer*⟩

⟨*function-declaration*⟩ ::=
  ⟨*typename*⟩ ⟨*identifier*⟩ `(` ⟨*function-formal-params-opt*⟩ `)` `{` ⟨*statement-list*⟩ `}`

⟨*function-formal-params*⟩ ::= ⟨*function-formal-param*⟩
  | ⟨*function-formal-params*⟩ `,` ⟨*function-formal-param*⟩

⟨*function-formal-param*⟩ ::= ⟨*outputspec*⟩ ⟨*typename*⟩ ⟨*identifier*⟩ ⟨*arrayspec-opt*⟩

⟨*outputspec*⟩ ::= `output` | ε

5.9.3 Declarations

⟨*declaration*⟩ ::= ⟨*function-declaration*⟩
  | ⟨*variable-declaration*⟩

⟨*arrayspec*⟩ ::= `[` ⟨*integer*⟩ `]`

⟨*variable-declaration*⟩ ::= ⟨*typename*⟩ ⟨*def-expressions*⟩ `;`

⟨*def-expressions*⟩ ::= ⟨*def-expression*⟩
  | ⟨*def-expressions*⟩ `,` ⟨*def-expression*⟩

⟨*def-expression*⟩ ::= ⟨*identifier*⟩ ⟨*initializer-opt*⟩
  | ⟨*identifier*⟩ ⟨*arrayspec*⟩ ⟨*array-initializer-opt*⟩

⟨*initializer*⟩ ::= `=` ⟨*expression*⟩

⟨*array-initializer*⟩ ::= `=` `{` ⟨*expression-list*⟩ `}`

⟨*typename*⟩ ::= `float` | `color` | `point` | `vector` | `normal` | `matrix` | `string` | `void`

5.9.4 Statements

⟨*statement-list*⟩ ::= ⟨*statement-list*⟩ ⟨*statement*⟩

⟨*statement*⟩ ::= ⟨*expression-opt*⟩ `;`
  | ⟨*scoped-statements*⟩
  | ⟨*declaration*⟩
  | ⟨*conditional-statement*⟩

| ⟨*loop-statement*⟩
| ⟨*loopmod-statement*⟩
| ⟨*lighting-statement*⟩
| ⟨*return-statement*⟩

⟨*scoped-statements*⟩ ::= `{` ⟨*statement-list-opt*⟩ `}`

⟨*conditional-statement*⟩ ::=
`if (` ⟨*expression*⟩ `)` ⟨*statement*⟩
| `if (` ⟨*expression*⟩ `)` ⟨*statement*⟩ `else` ⟨*statement*⟩

⟨*loop-statement*⟩ ::=
`while (` ⟨*expression*⟩ `)` ⟨*statement*⟩
| `do` ⟨*statement*⟩ `while (` ⟨*expression*⟩ `) ;`
| `for (` ⟨*for-init-statement-opt*⟩ ⟨*expression-opt*⟩ `;` ⟨*expression-opt*⟩ `)` ⟨*statement*⟩

⟨*for-init-statement*⟩ ::=
⟨*expression-opt*⟩ `;`
| ⟨*variable-declaration*⟩

⟨*loopmod-statement*⟩ ::= `break ;`
| `continue ;`

⟨*lighting-statement*⟩ ::=
`emit (` ⟨*expression-list-opt*⟩ `)` ⟨*statement*⟩
| `lights (` ⟨*expression-list-opt*⟩ `)` ⟨*statement*⟩
| `trace (` ⟨*expression-list-opt*⟩ `)` ⟨*statement*⟩

⟨*return-statement*⟩ ::= `return` ⟨*expression-opt*⟩ `;`

5.9.5 Expressions

⟨*expression-list*⟩ ::= ⟨*expression*⟩
| ⟨*expression-list*⟩ `,` ⟨*expression*⟩

⟨*expression*⟩ ::= ⟨*number*⟩
| ⟨*stringliteral*⟩
| ⟨*type-constructor*⟩
| ⟨*incdec-op*⟩ ⟨*variable-ref*⟩
| ⟨*expression*⟩ ⟨*binary-op*⟩ ⟨*expression*⟩
| ⟨*unary-op*⟩ ⟨*expression*⟩
| `(` ⟨*expression*⟩ `)`
| ⟨*function-call*⟩
| ⟨*assign-expression*⟩
| ⟨*ternary-expression*⟩
| ⟨*typecast-expression*⟩
| ⟨*variable-ref*⟩

⟨*variable-lvalue*⟩ ::= ⟨*identifier*⟩ ⟨*array-deref-opt*⟩ ⟨*component-deref-opt*⟩

⟨*variable-ref*⟩ ::= ⟨*identifier*⟩ ⟨*array-deref-opt*⟩ ⟨*component-deref-opt*⟩ ⟨*incdec-op-opt*⟩

⟨*array-deref*⟩ ::= `[` ⟨*expression*⟩ `]`

⟨*component-deref*⟩ ::= `[` ⟨*expression*⟩ `]`
| `[` ⟨*expression*⟩ `]` `[` ⟨*expression*⟩ `]`

⟨*binary-op*⟩ ::= `*` | `/`
| `+` | `-`
| `==` | `!=` | `>` | `>=` | `<` | `<=`
| `&&` | `||`

⟨*unary-op*⟩ ::= `-` | `!`

⟨*incdec-op*⟩ ::= `++` | `--`

⟨*type-constructor*⟩ ::= ⟨*typename*⟩ `(` ⟨*expression-list*⟩ `)`

⟨*function-call*⟩ ::= ⟨*identifier*⟩ `(` ⟨*function-args-opt*⟩ `)`

⟨*function-args*⟩ ::= ⟨*expression*⟩
| ⟨*function-args*⟩ `,` ⟨*expression*⟩

⟨*assign-expression*⟩ ::= ⟨*variable-lvalue*⟩ ⟨*assign-op*⟩ ⟨*expression*⟩

⟨*assign-op*⟩ ::= `=` | `*=` | `/=` | `+=` | `-=`

⟨*ternary-expression*⟩ ::= ⟨*expression*⟩ `?` ⟨*expression*⟩ `:` ⟨*expression*⟩

⟨*typecast-expression*⟩ ::= `(` ⟨*typename*⟩ `)` ⟨*expression*⟩

Keywords

The following are keywords, and may not be used as identifiers for variables or functions:

```
break color continue do else emit float for if lights matrix normal point output
return string trace vector void while
```

Reserved Words

The following words are reserved for possible future use, and may not be used as identifiers for variables or functions:

```
bool case char class const default double enum extern false friend inline int
long private protected public short signed sizeof static struct switch template
this true typedef uniform union unsigned varying virtual
```

Part II

Using Gelato

6 Running Gelato

6.1 `gelato` Command Line Operation

Invoking Gelato from the command line is done as follows:

`gelato [`*options*`]` *file1 ... filen*

Gelato will process any *options* first (regardless of where they appear on the command line), and then will read the *files* containing scene commands in sequence. Specifying multiple files is identical to specifying a single file that is a concatenation of all the files.

The remainder of this section explains the various command line options that may be used.

6.1.1 General options

`-help`

Prints a brief description of all the command line switches.

`-version`

Prints the version and compile date of Gelato.

`-statistics`

Print statistics after rendering the frame.

`-threads t`

Sets the number of threads (CPU's) simultaneously working on the frame. The default is 1.

`-net` *server*

Specifies servers for network-parallel rendering. See Section 6.6 for details.

`-display` *server*

Overrides the `$DISPLAY` environment variable.

`-cwd` *path*

Changes the working directory to *path* before beginning rendering.

`-env` *var=val*

Sets an environment variable before beginning rendering.

`-silent`

Supresses status and other messages (except error messages) from being displayed on the terminal.

`-preview` *p*

If *p* is nonzero, turns on preview mode and uses *p* as the override for `"shadingquality"`.

`-shade` *name*

Selects an alternate shading system. Currently supported options are `"gsl"` (the default – perform full GSL shading), `"defaultsurface"` (draw everything like the `"defaultsurface"` shader), and `"keyfillrim"` (draw with three automatically-placed, unshadowed lights). The non-GSL options may render previews very quickly by bypassing the usual shading system, and may also be used in conjunction with `-preview`.

6.1.2 Additional outputs

`-iv`

Causes an `Output` to be specified to write the image as it is rendered to an `iv` window, *in addition to* whatever outputs are specified in the scene. This displayed image will be the main RGBA output, unless the `-data` option is used to select a different channel to output.

`-o` *imagefile*

Causes an `Output` to be specified to write the final image to the given file, *in addition to* whatever outputs are specified in the scene. This saved image will be the main RGBA output, unless the `-data` option is used to select a different channel to output.

`-data` *channel*

Specifies which data (e.g., `rgba`) will be output in the extra image saved by the `-iv` or `-o` command. The `-data` command has no effect if neither `-iv` nor `-o` are used.

There are two special choices for *channel*: `"first"` causes the data from the first `Output` command to be saved or displayed; and `"all"` causes all output channels to be saved or displayed (as separate pages in `iv`).

6.2 Startup File

Upon startup, but before any user scene files are read, Gelato will read a file named `$GELATOHOME/.gelato.pyg` (where `$GELATOHOME` refers to the value of the environment variable of that name), if it exists, and then `$HOME/.gelato.pyg`, if it exists. These Pyg startup files may be used to set site-wide and user options for the renderer, including search paths, default image resolution and quality levels, etc.

6.3 Environment variables

Gelato's behavior is influenced by the following *environment variables* of the command shell:

`GELATOHOME`

This should point to the Gelato installation directory. The default search paths all refer to subdirectories within the installation directory — for example Gelato will search for certain shaders in `$GELATOHOME/shaders`. Most critically, the scene file format readers (for at least the startup file) must be in `$GELATOHOME/lib`.

`GELATOTEMP`

If set, this should point to a large, local directory where temporary files may be stored. If `GELATOTEMP` is not set, temporary files will be stored in `/usr/tmp` on Linux, or in `C:/temp` on Windows.

`LM_LICENSE_FILE`

Overrides the location of the Flexlm license file (but only if `$NVIDIAD_LICENSE_FILE` is not also set).

`NVIDIAD_LICENSE_FILE`

Overrides the location of the Flexlm license file (and also overrides `$LM_LICENSE_FILE`).

`GELATO_LICENSE_MODE`

NEW!

Gelato Pro

If set to `"basic"`, Gelato will run in basic mode without any overhead of checking for licenses. If set to `"pro"`, Gelato will only run in "pro" mode, and will refuse to render if no license is available (instead of reverting to "basic" mode).

If not set, or set to any other value, Gelato will use the default strategy of trying to check out a Gelato Pro license and reverting to "basic" mode if no license file or license server are available.

As a reminder, below we show how to set the `GELATOHOME` environment variable to `/opt/nvidia/gelato` using a variety of shells:

`csh`, `tcsh`:	`setenv GELATOHOME /opt/nvidia/gelato`
`ksh`, `bash`:	`export GELATOHOME=/opt/nvidia/gelato`

6.4 Remote display with `iv`

On Linux, if `gelato -iv` is started with the `DISPLAY` environment variable pointing to a remote machine's display, then `iv` is started on the remote machine, and completed pixels are sent there. This allows renders on a remote computer to be displayed from a user workstation. (Incidentally, `gelato` uses a local display connection to access the GPU, regardless of `$DISPLAY`.)

Remote display only works if `rsh` is set up between the two machines; the `gelato` process must be able to use `rsh` to run a command on the workstation machine without requiring a password. Furthermore, the environment of the `rsh`'d command must have `iv` on its path, and must have either no `DISPLAY` environment variable, or one that points to the workstation screen. When troubleshooting remote display, begin with a simple test of the first requirement. Start a shell on the rendering machine, and type:

```
rsh workstation echo hello
```

If permission is denied or a password is requested, you may need to modify `.rhosts` files on the workstation machine. If the command hangs, you may need to start the `rshd` daemon on the workstation machine, or modify security settings for `pam` or the firewall. Once this works, test the remote environment:

```
rsh workstation echo \$DISPLAY
```

The backslash is needed to ensure that `$DISPLAY` is expanded remotely. Finally, to check that `iv` can be run via `rsh`, copy an image file to a known place on the user workstation and then type (still in the shell on the rendering machine):

```
rsh workstation iv /usr/tmp/foo.tif
```

If this works, remote display should work from `gelato -iv`.

6.5 Preview Modes

Preview mode is a way to get Gelato to render scenes at a lower quality level, but very quickly (sometimes 100 times faster than usual). This is very useful for quickly viewing a scene to see how objects are placed, whether lights are on and positioned roughly correctly, etc. See Figure 6.1 for example images.

You can engage preview mode with the `-preview` *p* command line flag. When in preview mode, the value *p* is used as an alternate `"shadingquality"` and a number of alternate values are used other attributes (see Section 4.1 for details). For example,

```
gelato -iv -preview 0.1 myfile.pyg
```

Preview mode overrides the usual settings for several attributes including `"shadingquality"`, which in this example is set to 0.1. It's possible to change the values used for the other overrides, with the variety of `"preview:*"` attributes (see Section 4.1), but we think you should never need to change these values.

In the usual preview mode, Gelato is still using the full shaders, lights, and textures of the scene, but merely is computing them much less frequently. Another way of previewing a scene

with greatly reduced rendering cost is to avoid running the real lights and shaders in favor of a simple proxy shader on all objects. This can be accomplished using the `-shade`, which may take arguments of `"defaultsurface"` (shading all objects quickly as if lit by a single light at the camera position) and `"keyfillrim"` which lights all objects with a standard three-light setup. In both cases, the lights are unshadowed and all surfaces are essentially plain plastic. For example,

```
gelato -iv -shade defaultsurface myfile.pyg
```

The `-shade` and `-preview` options may be combined:

```
gelato -iv -shade defaultsurface -preview 0.25 myfile.pyg
```

The `-preview`, and `-shade` modes are orthogonal, and may be used together. It is also possible to control these modes via `Attribute` in the scene file (see Section 4.1).

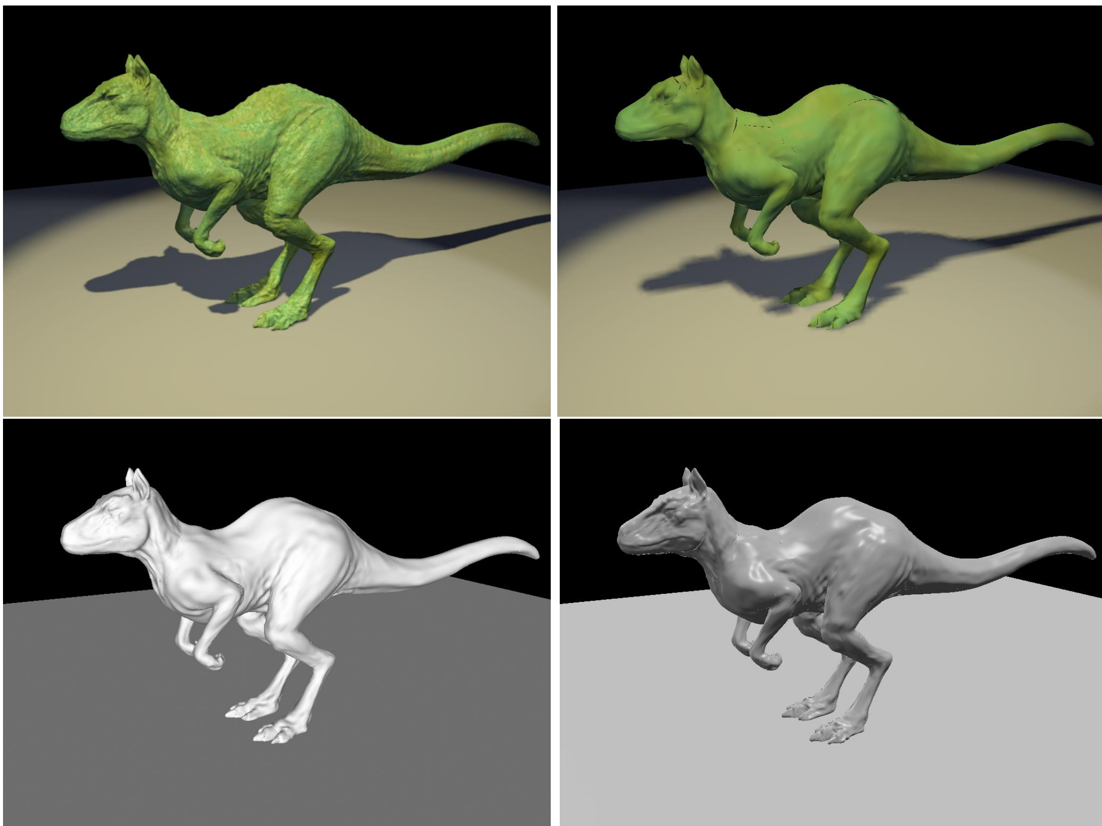

Figure 6.1: Example of preview modes. Top left: normal rendering, 16 seconds. Top right: `-preview 0.1`, 2 seconds. Note the blurrier shadows and texture, as well as some geometric artifacts. Bottom left: `-shade defaultsurface -preview 0.25`, 2 seconds. Bottom right: `-shade keyfillrim -preview 0.5`, 2 seconds.

6.6 Network-Parallel and Multi-Threaded Rendering

Gelato supports several methods for parallelizing rendering computations. Each has its strengths, weaknesses, and tradeoffs. The methods are:

- **Frame-parallel**: each processor or machine renders a different frame simultaneously.
- **Multi-threaded**: multiple processors or processor cores within a single computer contribute to rendering a single frame.
- **Network-parallel**: multiple processors or cores on a network (usually, but not exclusively on different computers) contribute to rendering a single frame.

Use and tradeoffs of each of these methods is described in the next three subsections.

Gelato Pro

NOTE: multi-threaded and network-parallel rendering require Gelato Pro.

6.6.1 Frame-parallel

With frame-parallel rendering, many frames render simultaneously on one or more computers. Each frame is completely separate from the others, and each simultaneously-rendering frame requires a separate Gelato license to be available.

Frame-parallel rendering is only fully efficient if you have many machines and many frames to render. Frame-parallel rendering cannot speed up the rendering of a single frame.

But if you do have many frames to render and many machines on which to do the rendering, frame-parallel rendering is the most efficient way to use your computing resources. It is "perfect" parallelism — if you have n machines, you can render n frames simultaneously in the same total time it would take you to render just one frame on a single machine. Because each frame is separate, there is no redundant work and no communication necessary between the processors rendering different frames.

Efficient and convenient frame-parallel rendering requires software for controlling which frames render on each machine (sometimes known as "render queue" or "render farm" management software). Gelato does not include such software; you will need to develop or license such software separately.

6.6.2 Multithreading

With multithreading, many processors (or processor cores, for "multi-core" processors) on one computer can be used by a single Gelato execution to render a frame. A Gelato execution in multithread mode requires only a single Gelato license, no matter how many threads or processors are being used.

Multithreaded rendering is much less efficient overall than frame-parallel rendering, but it has the advantage of being capable of speeding up rendering of a single frame. With multithreading, even though many processors are being used, there is still only a single copy of the scene in memory. Thus, it is much more memory-efficient than network rendering (even to the same machine). However, it often is not perfectly efficient – for example, 2 threads may only render a frame 1.5 times faster than a single thread – because of the overhead of careful coordination between the processors that are collaborating on the frame.

Multithreaded rendering is accomplished with the `-threads` command-line option to Gelato (or with `Attribute("int limits:threads")` in the scene file). To specify multiple threads, just use the `-threads` command line argument. For example:

```
gelato -threads 3 scene.pyg
```

6.6.3 Network-parallel

With network-parallel rendering, many processors ("servers") spread across a network (usually on separate computers) execute Gelato to render a single frame. Each simultaneous Gelato execution requires a separate Gelato license to be available. Fewer servers than you specify may be used if not enough licenses are available to check out.

Network-parallel rendering is much less efficient overall than frame-parallel rendering, but it has the advantage of being capable of speeding up rendering of a single frame.

Network-parallel rendering is accomplished with the `-net` command-line option to Gelato. To specify multiple servers, you can either use `-net` commands, or you may specify multiple server names within double quotes following `-net`. For example:

```
gelato -net server1 scene.pyg

gelato -net server1 -net server2 -net server3 scene.pyg

gelato -net "server1 server2 server3" scene.pyg
```

If a particular server has many processors, you may specify it more than once to indicate to run several simultaneous processes:

```
gelato -net "server1 server2 server2 server2" scene.pyg
```

Gelato's network rendering mode requires that all servers be able to find all the necessary scene files, shaders, textures, or other resources. The coordinating machine *does not* send those resources over the network.

By default, the local machine (that is, the one where you started Gelato) will not contribute to rendering the frame. That is, if you use `-net` at all, *only* the machines listed will contribute as "workers." If you want the local machine to also contribute, you must list its name, or `local`, for example:

```
gelato -net "local server1 server2" scene.pyg
```

While it is possible and sometimes useful to perform network-parallel renderering using many processors on a single machine, please be aware that each separate Gelato process will need to separately read the input files, and will each have its own copy of the scene. Thus, the memory efficiency of network-parallel rendering on a single machine is not nearly as good as with multithreading.

7 Cameras and Image Output

This chapter covers the basic details of how the CG camera is placed in the scene, and the various options that must be set to determine image resolution and framing, camera attributes, image quality, exactly what data are saved, and how you can determine the image file types and other properties.

7.1 The Camera

7.1.1 Positioning the Camera

Objects in the scene are positioned relative to `"world"` space or some other local coordinate system. This is the result of your having translated or rotated those objects to place them in the scene.

The camera also has a certain position and orientation relative to the world. A special coordinate system called `"camera"` space is centered about the camera, with the x axis pointing to the camera's right, the y axis pointing up, and the z axis pointing in the direction that the camera is looking. Note that this is a "left handed" camera coordinate system.

Gelato allows two ways to position the camera in the scene:

1. Assume that at the beginning of the frame that you start out in `"world"` space, set the camera position and orientation as you would with any other object (relative to `"world"` space), then make a `Camera` call. Upon hitting the `World` call, the coordinate system will be restored to `"world"` space. For example:

   ```
   # initially start in world space
   SetTransform (...)  # Position the camera
   Parameter (...)     # Set camera parameters
   Camera ("main")     # Instance a camera in the current position
   World ()            # Restore CTM to be world space again
   ...
   ```

   There may be multiple cameras in the scene, although at present, Gelato only will create images from the first `Camera` declared.

2. If *no* `Camera` call is made prior to the `World` call, then it is assumed that that the *initial* state was `"camera"` space, and that the CTM at the time of the `World` call represents `"world"` space. In other words, a lack of `Camera` statement implies that all the transformations prior to `World` are placing the world relative to the camera. Note that this is the

way that certain other API's (including OpenGL and RenderMan) naturally operate, so this mode allows for easy translation in cases where only one camera is needed.

If there is no `Camera` call in the scene (and thus, the renderer will implicitly create a camera once it hits `World`), all the optional camera parameters are also settable with the ordinary `Attribute` command.

```
# initially we are in camera space
SetTransform (...)  # Position the world relative to the camera
Attribute (...)     # Set camera parameters with Attribute
World ()            # Establishes world space
...
```

An early step in rendering is for objects to be transformed into `"camera"` space. This is illustrated figuratively in Figure 7.1.

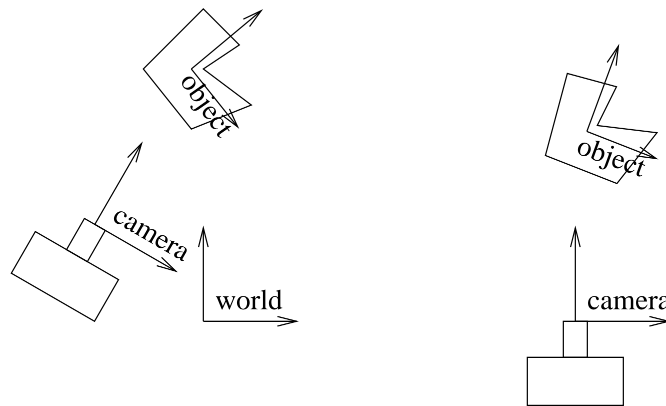

Figure 7.1: Transforming objects into camera space.

7.1.2 Camera Projection

A three-dimensional scene is reduced to a two-dimensional image by *projection*. In any projection, all points along a "line of sight" correspond to the same 2D location in the final image. Gelato supports both *perspective* (lines of sight converging at a point) and *orthographic* (parallel lines of sight) camera projections. Along any line of sight, the closest object to the camera will be the one seen (although if it is partially transparent, you may also see other objects behind it). This is illustrated by Figures 7.2 and 7.3.

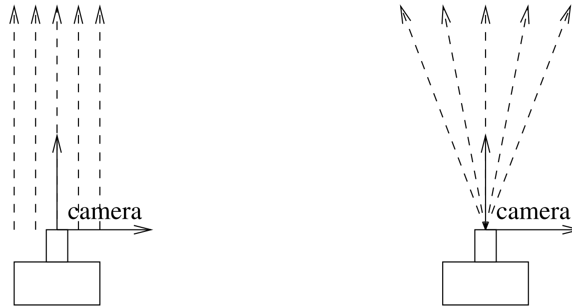

Figure 7.2: Left: Orthographic projections view along parallel rays. Right: Perspective projections view along rays that converge at the camera position.

The projection is selected using the "projection" camera attribute (see Section 4.1). By default, cameras use the "perspective" projection.

You may set the camera projection by specifying it as a parameter to the Camera call, as in this Pyg example:

```
Parameter ("string projection", "orthographic")
Camera ("main")
```

If no Camera command is used (implying that the camera was at the initial scene origin), then the projection may be set by an Attribute at any point prior to the World command:

```
Attribute ("string projection", "orthographic")
...
World ()
```

Perspective projection

Perspective projections are the default camera projection for Gelato. Perspective projections are similar to the projection used in an ordinary camera (though not exactly – real cameras always have additional distortions, but in the rare cases where this is important, it is usually corrected as an image-processing operation, not during rendering). With a perspective projection, an object will appear bigger when it is closer to the camera (see Figure 7.3).

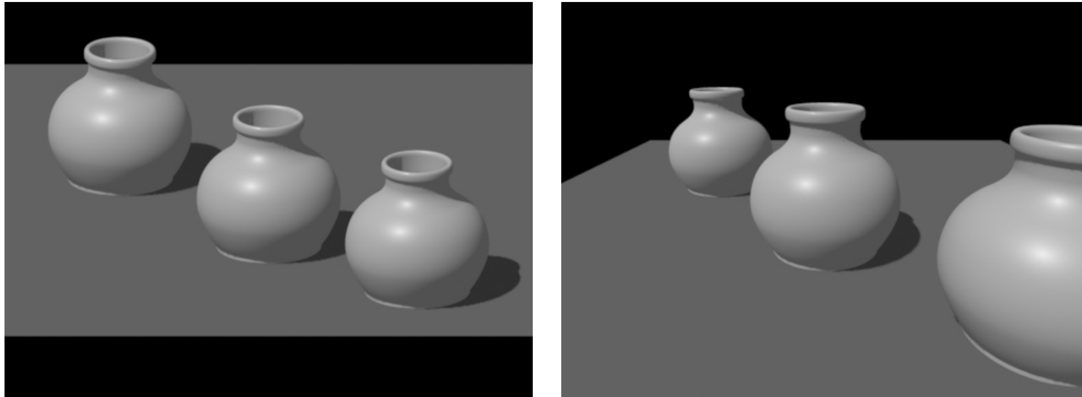

Figure 7.3: Example of an orthographic (left) and perspective (right) projections.

You will almost always want to use a perspective projection for “final” images from the main camera. Perspective projections should also be used when generating shadow maps for light sources that project light from a single point (like a spotlight).

The perspective projection also responds to the camera parameter `"fov"` which sets the field-of-view angle in degrees. For example, the following Pyg command sets a perspective projection with a 30 degree field of view:

```
Parameter ("string projection", "perspective")
Parameter ("float fov", 30)
Camera ("main")
```

Orthographic projection

Orthographic projections are primarily used for reproducing certain architectural or engineering drawing methods, and for creating shadow maps for “distant” light sources (those whose light emanates in parallel rays). With an orthographic projection, an object will appear the same size no matter what its depth from the camera (see Figure 7.3).

Orthographic projections do not respond to the `"fov"` parameter. The default orthographic projection is almost certainly too small a view, and you will need to adjust the `"screen"` parameter in order to correctly frame your scene for an orthographic view.

7.1.3 Motion Blur

Real cameras have a shutter that stays open for a certain amount of time to expose the film. The longer the shutter stays open, the more moving objects will appear as blurred streaks on the film (see Figure 7.4). This effect is critical to avoiding strobing when rendering frames for an animation. The shutter interval may be set with the `"shutter"` camera attribute, which takes the opening and closing times. For example,

```
Parameter ("float[2] shutter", (0, 1.0/48))
Camera ("main")
```

instructs the camera to open the shutter at time 0 and close it again 1/48 later.

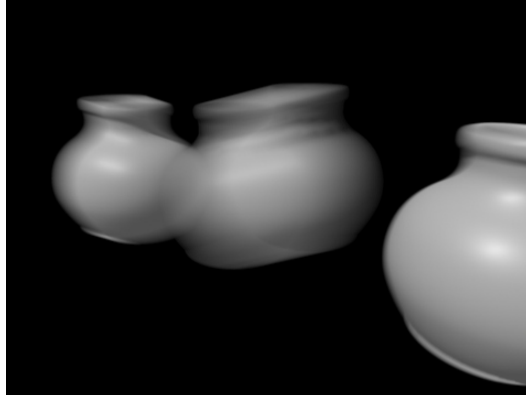

Figure 7.4: Motion blur.

The units (seconds, frames, etc.) do not matter, but they are expected to be calibrated to the same scale as the times specified by `Motion` for any moving or deforming objects.

Most motion picture film cameras leave the shutter open for approximately half of the inter-frame time. For example, for a 24 frame-per-second film, the actual shutter interval is typically 1/48 of a second.

For real cameras, the longer the shutter stays open, the more light strikes the film, and therefore the brighter the resulting image will be. This is not true for the synthetic camera — the image will be no brighter or dimmer, no matter what the `"shutter"` attribute specifies.

Moving objects in time

Individual objects may be blurred one of two ways: via *transformation* blur or *deformation* blur. For transformation blur, the object transformations themselves (the position and orientation of the object or part) are changing over time. For deformation blur, the positions of the vertices comprising the object move their position over time.

To explain how a transformation is blurred, consider a simple translation of 3 units in x:

```
Translate (3, 0, 0)
```

If, for example, the object should translate by 3 units at time 0 and by 3.5 units at time 1, the previous `Translate` would be replaced by the following *motion block*:

```
Motion (0, 1)
Translate (3, 0, 0)
Translate (3.5, 0, 0)
```

The arguments to the `Motion` function are a series of n time values (usually two, though it may be any number). It is then expected that the `Motion` function be followed by n API calls of the same name, but differing only in parameter values, each such call corresponding to one of the time values passed to the preceeding `Motion` call.

Any of the transformation routines (`Translate`, `Rotate`, `Scale`, `AppendTransform`, `SetTransform`) may be blurred in this manner.

It is also possible to blur the shape of the geometry itself by using a `Motion` call followed by *n* calls to a geometric function call (such as `Mesh`, `Patch`, etc.). Each geometric call must be the same function and "shape" (e.g., you may not "morph" a `Patch` into a `Sphere`, nor may you change the number of vertices or faces between subsequent `Mesh` calls within a motion block). For example:

```
Motion (0, 1)
Parameter ("vertex point P", (0,0,0, 0,0,1, 1,0,0, 1,0,1))
Patch ("linear", 2, 2)
Parameter ("vertex point P", (0,1,0, 0,0.5,1, 1,0,0, 1,0,1))
Patch ("linear", 2, 2)
```

In the example above, a bilinear batch is deforming over time. Note that the *n* time values passed to `Motion` are followed by *n* full geometric primitives – including both parameters as well as the geometric primitive itself.

Multi-segment motion blur

`Motion` calls usually specify two times, and therefore be followed by two transformation calls or geometric primitives. This results in *linear motion blur*, in which any particular point on the object moves in a straight line over the course of a frame. Considering that each final frame will be seen by viewers for only a fraction of a second, linear motion blur is almost always sufficient.

However, it is sometimes desirable to have an object trace out a more complex path over the course of a frame (see Figure 7.5). As a practical matter, this is only necessary for objects undergoing rapid rotation (think "helicopter rotor") for which linear motion would look obviously wrong. Of course, this is very simple in Gelato, merely requiring a larger number of times in the motion block. For example:

```
Motion (0, 0.25, 0.5, 0.75, 1)
Rotate (15, 0, 0, 1)
Rotate (30, 0, 0, 1)
Rotate (45, 0, 0, 1)
Rotate (60, 0, 0, 1)
Rotate (75, 0, 0, 1)
Input ("rotor.pyg")
```

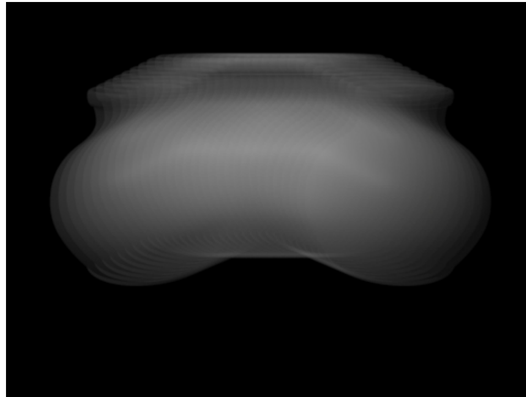

Figure 7.5: Multi-segment motion blur: the object can be seen moving in an arc, not a straight line.

7.1.4 Depth of Field

Depth of field refers to the way objects at a particular distance from the camera appear in sharp focus, while objects that are closer or farther away will appear blurred (see Figure 7.6). It is a physical phenomenon caused by the finite aperture of a camera, and other focusing attributes of the lens system.

By default, Gelato has depth of field turned off, meaning that all objects are in sharp focus, regardless of their depth in the scene. The depth of field effect can be turned on and adjusted with three camera attributes: `"fstop"`, `"focallength"`, and `"focaldistance"`.

The f/stop is the ratio of focal length to lens aperture, much as you would see f/stop settings on a real camera lens — it lets you control the aperture size. The focal length is the distance from the lens opening to the film plane. The focal distance is the depth from the camera at which objects appear in sharp focus. Both the focal length and focal distance are measured in the same units as `"camera"` space.

For example, if you had constructed your scene so that `"camera"` space units were meters, then the following command would specify an f/4 aperture on a 50mm lens, set to focus sharply objects that were 3.6 meters from the camera:

```
Parameter ("float fstop", 4)
Parameter ("float focallength", 0.05)
Parameter ("float focallength", 3.6)
Camera ("main")
```

For real cameras, the wider the aperture (i.e., the smaller the f/stop number), the more light enters the camera, and therefore the brighter the resulting image will be. This is not true for the synthetic camera — the image will be no brighter or dimmer, no matter what the depth of field settings.

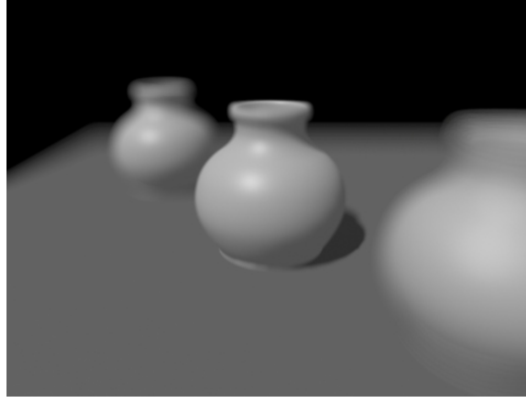

Figure 7.6: Depth of field.

7.1.5 Clipping

In addition to objects being not visible to the camera if it is too far to the right or left, top or bottom (that is, off-screen), you can also have the camera ignore objects that are too near to, or too far from the camera. This is something that obviously cannot be done with a real camera, but it can be very useful and often comes in handy with the CG camera. Objects are ignored if their `"camera"` space z values are less than the *near* plane, or if their `"camera"` space z values are greater than the *far* plane.

The z clipping planes can be set with the `"near"` and `"far"` camera attributes. For example, to set the hither plane to $z = 0.1$ and the yon plane to $z = 10,000$:

```
Parameter ("float near", 0.1)
Parameter ("float far", 10000)
Camera ("main")
```

There is some benefit to attempting to set the clipping carefully. Tightly bounding the depth of interest in your scene can preserve more computational precision in some parts of the rendering process.

7.2 Image Resolution and Framing

7.2.1 Image Resolution

The image *resolution* refers to the number and shape of the pixels in the final image. The `"resolution"` camera attribute (see Section 4.1) can be used to set the x and y resolution, which are whole numbers that give the size of the final image, in pixels. The `"pixelaspect"` camera attribute describes the ratio of the width to height of an individual pixel (the default value, 1.0, indicates square pixels). For example, to render an image with 640×480 square pixels:

```
Parameter ("int[2] resolution", (640, 480))
Parameter ("float pixelaspect", 1)
Camera ("main")
```

The aspect ratio of the frame is determined by the x and y resolution. Therefore, setting the `"resolution"` not only determines the number of pixels in the image, but also the "shape" of the resulting image.

7.2.2 The Screen Window

Once the scene is projected to a 2D plane, only a subset of the plane is actually turned into the image. That subset is called the *screen window*. This is directly settable by the `"screen"` camera attribute, which by default is

$$(-frameaspectratio, frameaspectratio, -1, 1)$$

where $frameaspectratio = xres/yres$.

Since the default is for the screen window to be centered and with the frame aspect ratio, it is usually not necessary to set the `"screen"` camera attribute. However, there are two instances where it is critical: (1) For an orthographic camera, the `"screen"` attribute is almost certainly required to ensure proper framing of the image. (2) the `"screen"` attribute can be used to distort or shear the image by making the screen window's aspect ratio not match the frame aspect ratio, or by using an off-center screen window.

7.2.3 Crop Window

It is often very useful to render a subset of image pixels, particularly if you are debugging or adjusting part of the scene and do not wish to wait for the entire image to rerender every time a tweak is made. This is easily accomplished by setting the `"crop"` camera attribute, which takes x minimum and maximum, and y minimum and maximum range, expressed as a portion of the total image (i.e., 0–1). For example, to render the upper-right quadrant of the image only:

```
Parameter ("float[4] crop", (0.5, 1, 0, 0.5))
Camera ("main")
```

Even though the crop window is expressed with floating-point numbers, it will be rounded in such a way as to result in a whole number of pixels. Gelato is careful to round and to filter at the edges so that adjacent crop windows will match up exactly without seams. For example, rendering one image with:

```
Parameter ("float[4] crop", (0, 1, 0, 0.5))
```

and a second image of the same scene with:

```
Parameter ("float[4] crop", (0, 1, 0.5, 1))
```

will *exactly* render all pixels, without repetition and without seams if the two images are assembled together.

7.3 Image Output

Once pixel values are derived by filtering the data from the pixel subregions, an image must be written to disk in some format, or displayed on some device.

7.3.1 Outputs and Channels

Upon rendering, Gelato produces one or more *image outputs*. You can think of each output as a separate image of the scene (from the same camera). Each output stream may contain different data — for example, one output stream may consist of color and alpha (RGBA), while another output stream may contain z depth information.

Each image output consists of one or more *channels*. A channel is a single "pane" of data, such as red or blue. A greyscale-only image is a 1-channel image, an ordinary color RGB image is a 3-channel image, and an RGBA image is a 4-channel image.

7.3.2 Image I/O Plugins

There are many different types of image file formats or devices on which to display images. Gelato uses programs called *Image I/O plugins* to handle image output and display.[1] There is one such plugin for each file format or display type. Gelato comes with image I/O plugins that understand how to write TIFF, OpenEXR, JPEG, and Targa files, and how to display the image on a computer screen (`"iv"`). Developers can expand Gelato's format repertoire by writing their own plugins, as explained in Chapter 13.

The basic way to specify what data goes to which file and in what format is with the `Output` command:

`Output` (*name, format, dataname, camera, ...params...*)

The *name* is the filename of the file, the *format* is the file format or display device (actually the name of the imageio plugin), *data* is the name of the data to write to the output stream, and *camera* is the name of the camera from which to image this output. The optional *params* (which may be passed beforehand using the `Parameter` call) control various aspects of the image output, including format-specific options.

As an example, to instruct Gelato to write RGB color data from camera `"main"` to a TIFF file named `"myfile.tif"`:

```
Output ("myfile.tif", "tiff", "rgb", "main")
```

To write an image with RGB and alpha (coverage) as a 4-channel image:

```
Output ("myfile.tif", "tiff", "rgba", "main")
```

To display the image "live" to a framebuffer display using Gelato's `iv` display tool:

```
Output ("myfile.tif", "iv", "rgba", "main")
```

7.3.3 Specifying an ImageOutput directly

NEW!

When linking against a live renderer (instead of generating Pyg, for example), it is also possible to directly use an existing `ImageOutput` object created by the application. Since there is not a version of `Output` that takes a `ImageOutput*`, this can be accomplished by passing a *format* argument to `Output` that consists of an ASCII string starting with `"0x"` followed by the hexidecimal address of the `ImageOutput` in memory. For example:

```
ImageOutput *myimageoutput = new MyImageOutput (...);
char addr[20];
// don't use %p - it is different on Windows and Linux
sprintf (addr, "%#lx", myimageoutput);
r->Output (dummy_filename, addr, "rgba", "maincamera");
```

As an example of how this can be useful, consider that you could create an ImageOutput subclass that directly displayed completed image pixels in a display window of the application (as opposed to a separate `iv` process) or otherwise uses the pixel data in the calling process.

7.3.4 Bit depth, quantization, and dither

Gelato computes pixel values with floating-point precision, but not all output formats support floating-point data. Therefore, the Image I/O plugin may need to convert the raw pixel data to an integer (whole number) representation. This process is known as *quantization*.

The `Output` command takes an optional parameter `"quantize"` that gives the quantization mapping. The `"quantize"` parameter takes an array of four integers that specify the *zero* level, *one* level, *min*, and *max* values.

When floating-point numbers are converted to integers, the number of bits per channel is known as the *bit depth*. The bit depth is computed automatically from the *max* quantization value: if $max \leq 255$, an 8-bit file is created; otherwise, if $max \leq 65535$, a 16-bit file is created; otherwise, a floating-point output file is created. Also, if all of *zero*, *one*, *min*, and *max* are 0, floating-point output will be selected.

For example, to write `"myfile.tif"` as 8-bit integers (this is a typical output format, and also the default):

```
Parameter ("int[4] quantize", (0, 255, 0, 255))
Output ("myfile.tif", "tiff", "rgb", "main")
```

If 8 bits per channel are not enough precision for your application, you could generate a 16 bit per channel image with the following command:

```
Parameter ("int[4] quantize", (0, 65535, 0, 65535))
Output ("myfile.tif", "tiff", "rgb", "main")
```

To aid in reducing artifacts that result from the float-to-integer conversion, you can add a random *dither* to the image. This is just noise that helps to soften the edges and reduce objectionable banding in the image. The dither amplitude can be set by `Output` using the

[1]Image I/O plugins also handle image *input* into the renderer.

optional parameter "dither". The default dither level is 0.5. The main reason to override this default is in the case of floating-point images, which do not need dither and therefore should have their dither set to 0. For example, to output color pixels with full floating-point precision (and no dither):

```
Parameter ("int[4] quantize", (0, 0, 0, 0))
Parameter ("float dither", 0)
Output ("myfile.tif", "tiff", "rgb", "main")
```

7.3.5 Filters

As described earlier in Section 7.5, selection of a pixel filter can be accomplished by setting the "filter" (which takes a string giving the filter name) and "filterwidth" (which takes two floats that specify the *x* and *y* support widths of the filter). For example, to use the Catmull-Rom filter with width 3:

```
Parameter ("string filter", "catmull-rom")
Parameter ("float[2] filterwidth", (3, 3))
Output ("myfile.tif", "tiff", "rgb", "main")
```

As another example, *z* depth images should not overlap pixel boundaries and must be floating point. Furthermore, we recommend using the "min" filter for ordinary depth maps, Therefore, the proper Output command to write a *z* depth file is:

```
Parameter ("int[4] quantize", (0, 0, 0, 0))
Parameter ("float dither", 0)
Parameter ("string filter", "min")
Parameter ("float[2] filterwidth", (1, 1))
Output ("myfile.sm", "shadow", "z", "main")
```

7.3.6 Arbitrary Output Variables

In addition to color, alpha, and depth, "global" shader variables (see Table 5.3) and any arbitrary data computed and stored in an output variable of the surface shader may be an image output. Outputs may contain ID tags for objects; separate ambient, diffuse, and specular lighting; normal (orientation) information of the surfaces — in short, anything you can compute in the shaders.

To specify multiple output streams, you can simply use multiple Output commands. Remember that Gelato allows you to use entirely different Image I/O plugins, formats, quantization levels, and even pixel filters for each output stream.

For example, to have one output stream containing RGBA quantized to 16 bits per channel, a second output stream with floating-point *z* depth data, a third output stream containing the surface normals, and a fourth output stream containing floating-point color value for just the specular highlight:

```
Parameter ("int[4] quantize", (0, 65535, 0, 65535))
Parameter ("float dither", 0.5)
Output ("myfile.tif", "tiff", "rgba", "main")
```

```
Parameter ("int[4] quantize", (0, 0, 0, 0))
Parameter ("float dither", 0)
Parameter ("string filter", "min")
Parameter ("float[2] filterwidth", (1, 1))
Output ("myfile.sm", "shadow", "z", "main")

Parameter ("int[4] quantize", (0, 0, 0, 0))
Parameter ("float dither", 0)
Output ("normals.tif", "tiff", "normal N", "main")

Parameter ("int[4] quantize", (0, 0, 0, 0))
Parameter ("float dither", 0)
Output ("specpass.tif", "tiff", "color Cspec", "main")
```

Notice that when outputs contain data computed by the shader, the data name is simply the name and type of the `output` variable of the shader. For shaders that do not contain the `output` variable with the correct name and type, the variable will have value 0.

7.4 Gelato's Bundled Image I/O Plugins

As discussed in Section 7.3, the `Output` command takes the name of a Image I/O plugin, which is a plugin that actually writes the pixels in a particular format. Gelato ships with several image I/O plugins (and hence, can write to those formats). Users or third parties may expand the formats by writing DSO's/DLL's, as described in Chapter 13. This section describes Gelato's bundled image I/O plugin types.

7.4.1 "tiff" plugin

The built-in `"tiff"` image I/O plugin writes output pixels as TIFF files. The plugin can write pixels as 8-bit unsigned integer, 16-bit unsigned integer, or 32-bit floating-point data, depending on the `"quantize"` parameter. The resulting output file is a totally standard scanline-oriented TIFF. Accepted optional parameters are:

Name	Type	Description
"quantize"	int[4]	Gives the conversion from floating point to integer, and indirectly, the data format of the TIFF file, as described in Section 7.3.4.
"dither"	float	The amplitude of random dither to add to output pixels, as described in Section 7.3.4.
"gain"	float	A constant multiplicative factor applied to pixel values prior to quantization.
"gamma"	float	A nonlinear gamma correction factor applied to pixel values prior to quantization.
"compression"	string	The name of the TIFF compression method — one of "none", "lzw", "zip", or "deflate" (synonym for zip).
"rowsperstrip"	int	The number of TIFF rows per strip (default: 32).
"ImageDescription"	string	Optional description of the image.
"DocumentName"	string	Optional document name.
"Artist"	string	Optional artist name.
"copyright"	string	Optional copyright notice.

If any of the optional "ImageDescription", "DocumentName", "Artist", or "copyright" parameters are given, their contents will be stored in the file as the TIFF tags with the same names.

7.4.2 "shadow" plugin

Requesting image type "shadow" actually results in a TIFF file, but automatically using tiled output, floating-point data, and storing the camera and screen matrices. This is Gelato's preferred shadow (depth) map format. Just as with ordinary "tiff" files, the "ImageDescription", "DocumentName", "Artist", and "copyright" parameters are accepted, and will be stored as TIFF tags in the resulting file.

7.4.3 "iv" plugin

The bundled "iv" image I/O plugin communicates its output images to an interactive session of the iv image viewing program. The operation of iv is described $GELATOHOME/doc/iv/iv.html. The "iv" image I/O plugin responds to the following optional Output parameters:

Name	Type	Description
"quantize"	int[4]	Gives the conversion from floating point to integer, and indirectly, the data format of the file, as described in Section 7.3.4.
"dither"	float	The amplitude of random dither to add to output pixels, as described in Section 7.3.4.
"gain"	float	A constant multiplicative factor applied to pixel values prior to quantization.
"gamma"	float	A nonlinear gamma correction factor applied to pixel values prior to quantization.
"foreground"	string	The name of a foreground image that will be composited over the rendered image when displayed.
"background"	string	The name of a background image that will be composited underneath the rendered image when displayed.
"fullscreen"	int	A nonzero value forces iv to start up in full-screen mode.
"zoom"	int	Starts up iv at the specified zoom level.

7.4.4 "OpenEXR" plugin

The bundled "OpenEXR" image I/O plugin writes output pixels as scanline-oriented OpenEXR files (see http://www.openexr.org). The plugin always writes image channels as 16-bit floating point ("half").

Because OpenEXR uses a floating-point format, not integers, there is no quantization step per se, so the "quantize", and "dither" parameters are ignored. Accepted optional parameters are:

Name	Type	Description
"gain"	float	A constant multiplicative factor applied to pixel values prior to quantization.
"gamma"	float	A nonlinear gamma correction factor applied to pixel values prior to quantization.
"compression"	string	The name of the compression method — one of "none", "zip", "deflate" (synonym for zip), "zips", "piz", "pxr24". See the OpenEXR documentation (www.openexr.org) for information on the compression methods. By default, Gelato's OpenEXR plugin uses "zip" compression.
"owner"	string	Optional notice (presumably the title or owner of the image.
"copyright"	string	Optional synonym for "owner".
"software"	string	Optional name of software.

7.4.5 "jpg" plugin

The bundled "jpg" image I/O plugin saves files in JPEG format. We generally do not recommend rendering images directly into JPEG format because JPEG is restricted to 8-bit per

channel, 3-channel images, and is "lossy." Accepted optional parameters are:

Name	Type	Description
"quantize"	int[4]	Gives the conversion from floating point to integer, and indirectly, the data format of the TIFF file, as described in Section 7.3.4.
"dither"	float	The amplitude of random dither to add to output pixels, as described in Section 7.3.4.
"gain"	float	A constant multiplicative factor applied to pixel values prior to quantization.
"gamma"	float	A nonlinear gamma correction factor applied to pixel values prior to quantization.

7.4.6 "iff" plugin

The bundled "iff" image I/O plugin saves files in Alias Maya's IFF format. It is capable of reading and writing 8- and 16-bit integer pixel values, with no more than 4 data channels.

The "iff" plugin ignores "quantize" and "dither", always quantizing 8-bit files to (0..255) and 0.5 dither, and always quantizing 16-bit files to (0..65535) and 0.5 dither. At this time, no other parameters have any effect.

7.4.7 "DevIL" plugin

The bundled "DevIL" image I/O plugin uses the open-source DevIL toolkit to read and write a variety of formats. The DevIL toolkit is distributed under the LGPL and is available for download at http://openil.sourceforge.net/.

The DevIL plugin can read and write any of the following file formats:

bmp	Microsoft Windows BMP
png	Portable Network Graphics (PNG)
sgi	SGI Image format
tga	Targa

The formats supporteb by the "DevIL" image I/O plugin are all 8-bit per channel, so we do not recommend rendering to any of these formats (although it will work). But the plugin is very useful in allowing iv and maketx to display or create textures from files in these formats.

Users familiar with DevIL may wonder about all the other formats it purportedly supports. Alas, these are the only formats for which we could verify that DevIL worked broadly across a selection of test images.

7.5 Antialiasing and Filtering

Antialiasing refers to the renderer's efforts to correctly capture details smaller than a pixel (including geometric edges) and to give a smooth appearance to the blur that results from motion or depth of field. There are several options that control the basic time versus quality tradeoffs when performing antialiasing, described below.

Edge antialiasing quality

The most basic antialiasing control is the *spatial quality*, which describes the number of subpixel regions (in *x* and *y*) comprising each pixel, for example:

```
Attribute ("int[2] spatialquality", (4, 4))
```

Dividing pixels into smaller regions, each of which is solved separately, is important to antialiasing because smaller regions are geometrically simpler (contain fewer objects and edges) and therefore easier to approximate with certain simplifying assumptions. More subpixel regions will yield higher quality, but will take slightly longer to render. Gelato allows you to greatly increase "spatialquality" with minimal changes to render time (unlike other renderers which can slow down significantly as antialiasing quality is improved). For this reason, the default spatial quality is (4,4), and there is virtually no discernable speedup by reducing quality to (1,1). Higher values can be used for scenes that are unusually difficult to antialias (e.g., if there is lots of small geometry like hair).

Motion blur quality

The quality of the motion blur is controlled by a camera attribute called "temporalquality" (see Section 4.1). For example:

```
Attribute ("int temporalquality", 16)
```

Specifying the spatial and temporal antialiasing controls separately helps the renderer to understand how to allocate its resources more optimally than if a single number were used. In particular, Gelato uses the extra temporal quality only for screen regions that actually contain moving geometry. In other words, if only some objects in the scene are moving, less work is done in pixels where all objects are standing still, therefore rendering is much faster than if the work was done uniformly in all pixels.

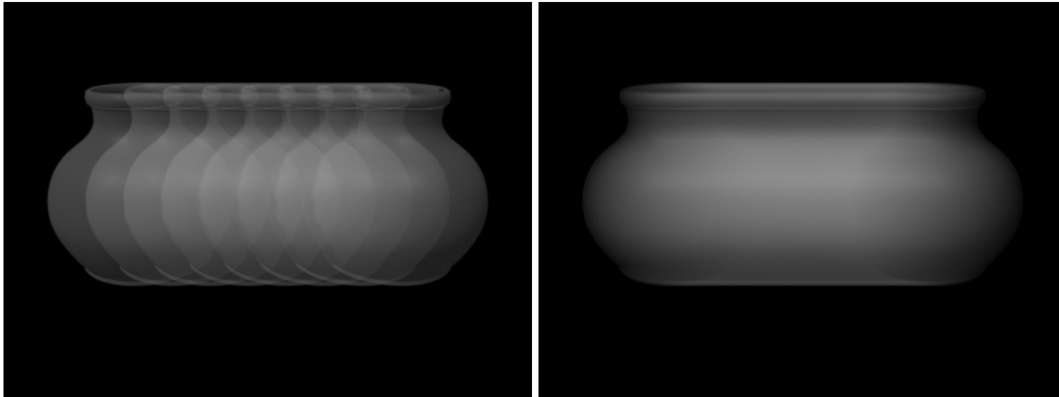

Figure 7.7: Left: Strobing artifacts from insufficient temporal quality (8). Right: using temporal quality 64.

When the number of time samples is sufficiently high, the motion blur should look completely smooth. Fast-moving objects will appear to *strobe*, or break up into several "stamped out" images, if the number of time samples is not high enough for the amount of motion blur (see Figure 7.7). When this happens, you should increase the `"temporalquality"`.

Depth of field quality

The quality of the depth of field is controlled by a camera attribute called `"dofquality"` (see Section 4.1). For example:

```
Attribute ("int dofquality", 32)
```

Actually, the renderer will use the greater of dofquality (if depth of field is used) and motion blur quality. These two quality settings do not "multiply" against each other.

When the number of depth of field samples is sufficiently high, the DOF blur should look completely smooth. Objects that are extremely out of focus, with insufficient `"dofquality"`, will appear to break up into several "stamped out" images (similar to what happens with motion blur when insufficient `"motionquality"` is used). If this artifact is seen, you should increase the `"dofquality"`.

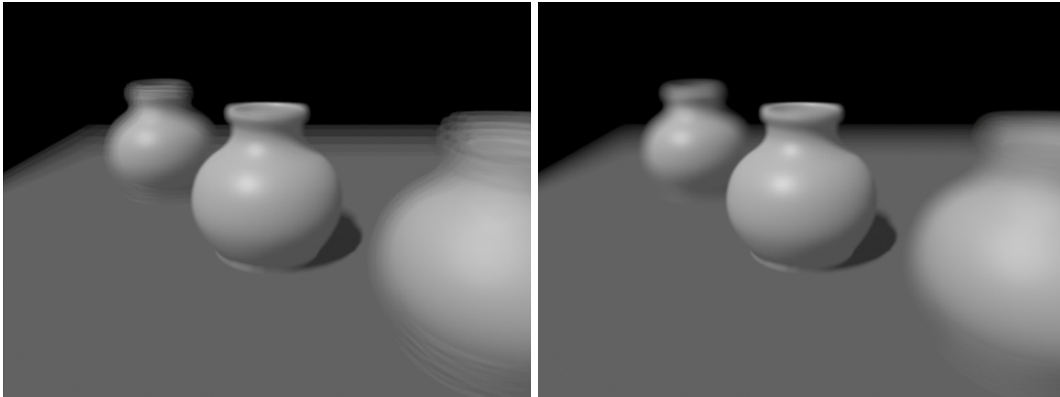

Figure 7.8: Left: Stamped-out artifacts from insufficient dofquality (4). Right: using "dofquality" 64.

Filtering

The several spatial subpixel regions must be combined to form the final discrete pixels. To do so with high quality, each pixel gets a weighted average of nearby regions (including regions outside the boundaries of the pixel. This process is known as *filtering*. Filtering has two aspects: the shape of the filter (specified by the name of the filtering function), and the width of the region to which the filter is applied. The filtering can be set by the "filter" and "filterwidth" image attributes. A 2×2 Gaussian filter (the default) may be specified by:

```
Parameter ("string filter", "gaussian")
Parameter ("float[2] filterwidth", (2, 2))
Output ("myfile.tif", "tiff", "rgb", "main")
```

Some people find the 2×2 Gaussian filter to be overly blurry. If that is the case, you could try a Gaussian filter with thinner width (but we wouldn't recommend using a width of less than 1.5), or you could use a different filter shape. The Catmull-Rom filter has nice edge sharpening properties, and can be specified as:

```
Parameter ("string filter", "catmull-rom")
Parameter ("float[2] filterwidth", (3, 3))
Output (...)
```

On the other hand, if it is important that pixels equally weight all regions and not consider any spatial regions outside the pixel boundary, then you would want to specify the (infamously low quality) box filter:

```
Parameter ("string filter", "box")
Parameter ("float[2] filterwidth", (1, 1))
Output (...)
```

Feel free to experiment with different filter functions and window sizes, to achieve a "look" that is right for your project. The available filters are listed in the formal description of the "filter" attribute in Section 4.2.

7.6 Stereo Rendering

Gelato allows a single Camera to easily render stereo views, one image for each of the left and right eyes. This is done in a single rendering pass, using a single render license, and tends to be much faster (stereo rendering usually takes about 30% longer than rendering a single image, as opposed to taking fully twice as long to render both views separately). However, this speed-up is only possible because as a shortcut, *the shading is only performed once for stereo rendering*, rather than separately for each eye. It is possible that some scenes exhibit unacceptable artifacts due to the "single-shade" simplification. In those cases, you will need to render each eye view separately, using the correct view-dependent shading for each one.

Stereo rendering is accomplished simply by specifying several optional parameters to Camera, detailed in Section 4.1. For example, the following Pyg:

```
Parameter ("string projection", "perspective")
Parameter ("float fov", 30)
Parameter ("float stereo:separation", 2.5)
Parameter ("float stereo:convergence", 300)
Parameter ("string stereo:projection", "off-axis")
Camera ("main")
```

The *separation* gives the distance along the x-axis between the two stereo cameras, as measured in "camera" space units. If *separation* is 0, only a single view will be rendered (not stereo).

If the projection is "off-axis" (the default), the cameras will face parallel directions but with their frusta constrained to share the same viewing area at $z = convergence$ (in camera space units); if "parallel", the camera views will be parallel with regular on-axis frusta (convergence is ignored); if "toe-in", the two cameras will be rotated about their y axes to converge at $z = convergence$. Figure 7.9 illustrates these three modes.

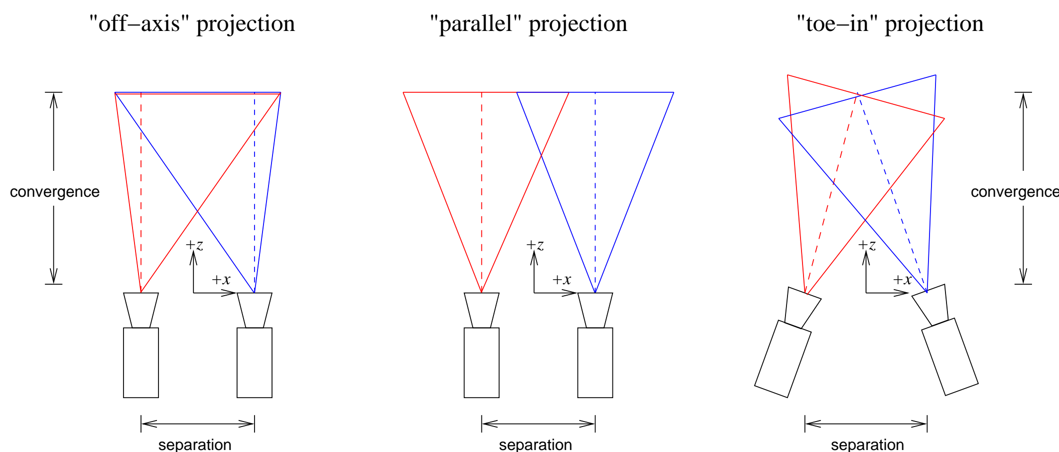

Figure 7.9: Stereo views: Separation and convergence.

Only one Output statement is required for stereo rendering, even though two images are generated. By default, the two images will be named "*name*-left.*ext*" and "*name*-right.*ext*",

where "*name.ext*" was the single filename given to `Output`. But it is easy to override these filenames to explicitly specify the two eye image filenames, using the optional `Output` parameters "`string stereo:left`" and "`string stereo:right`", as in the following Pyg example:

```
Parameter ("string stereo:left", "myleftfile.tif")
Parameter ("string stereo:right", "myrightfile.tif")
Output ("myfile.tif", "tiff", "rgba", "main")
```

Stereo rendering may be combined with any other rendering features; for example, antialiasing, motion blur, depth of field, etc. all work as expected.

For the occasional scene in which the proper shading parallax is important — that is, scenes for which shading from only one vantage point results in objectionable artifacts — stereo rendering is probably not recommended and the right solution is to render each eye as a completely separate render.

7.7 Multiple Cameras

Gelato is capable of rendering multiple camera views simultaneously, subject to the limitation that all tessellation and shading is performed from a single view corresponding to the *first* camera declared in the scene. Remember that many shading operations are *view dependent* — specular highlights, reflections, and refractions among the most important — and these may appear incorrect when viewed from camera views other than the "shading" camera.

As a special case of multi-camera rendering, *stereo rendering* is paricularly efficient and tends not to have visible artifacts from single shading. It is particularly easy to specify a stereo camera, without needing to specify multiple cameras or outputs, as detailed in Section 7.6.

General multiple camera rendering can be easily performed using the following rules:

1. Multiple `Camera` statements in the scene will instantiate multiple cameras.
2. The *first* `Camera` statement in the scene will be the one from which shading is computed. Tessellation decisions are also made from the shading camera, so views from other cameras may be somewhat under- or over-tessellated.
3. Each camera for which you want rendered images must have one or more corresponding `Output` statements. A camera with no corresponding `Output` statement will not be rendered and will produce no images.

As an example of using multiple cameras, consider the following Pyg fragment which sets up stereo rendering (in effect, the stereo method of the previous section is simply shorthand for the following):

```
# Set up first camera at the center as the shading reference camera
PushTransform ()
Parameter ("string projection", "perspective")
Parameter ("float fov", 30)
Camera ("shading_cam")
PopTransform ()
```

```
# Translate -sep and set up the left eye camera
PushTransform ()
Translate (-sep, 0, 0)
Parameter ("string projection", "perspective")
Parameter ("float fov", 30)
Camera ("left_cam")
PopTransform ()

# Translate +sep and set up the left eye camera
PushTransform ()
Translate (sep, 0, 0)
Parameter ("string projection", "perspective")
Parameter ("float fov", 30)
Camera ("right_cam")
PopTransform ()

# Create an output for each of the left and right eye cameras
Output ("left.tif", "tiff", "rgba", "left_cam")
Output ("right.tif", "tiff", "rgba", "right_cam")
```

8 Using Shaders

8.1 Shader Basics

Shaders are small user-supplied programs that describe materials and lights.

8.1.1 Types and usages of shaders

Every shader has a *shader type* given in the declaration in the shader's source code. The shader type may be one of: `surface`, `displacement`, `volume`, or `light`. Each type of shader may access a certain set of variables (see Section 5.7), and some shader types have restricted access to function calls or syntactic structures (for example, only light shaders may have `emit` statements).

At render time, various shaders may be bound to different pieces of geometry for a variety of *shader usages*. Shader usage refers to what functionality the shader is expected to provide, and exactly when in the rendering process it is executed. Shader usages currently include:

- `"displacement"` shaders may move the surface positions or alter their normals to make "dents" or other fine shape changes to an object. Displacements are calculated once, are the first shaders to run on any surface, and are independent of the viewing direction.

- `"surface"` shaders determine the color and opacity of the object, as viewed from a particular direction.

- `"volume"` shaders are run after the surface shader, and are allowed to modify the color and/or opacity in order to account for atmospheric effects along the viewing ray.

- `"light"` shaders are run when surface or volume shaders have `lights()` statements (or calls to functions that implicitly run the lights, such as `diffuse()` and `specular()`), and determine how much energy arrives at a point due to a particular light source.

Lights are instantiated with the `Light` API call, whereas the other shader usages are bound to specific pieces of geometry with the `Shader` call. The shader usage must match the shader type declared in the shader's source code.

Future releases of Gelato, or other renderers that support the Gelato API's or shading language, may support additional shader types and usages. A shader of a type or usage not supported by a particular renderer implementation will be ignored.

All objects are required to have a surface shader, and if none is specified in the scene, a `defaultsurface` shader will be used. Objects are not required to have displacement or volume

shader, and by default they do not. Objects are also not required to have any lights, though of course if there are no lights, objects will appear black unless their surface shaders assign colors regardless of the amount of light shining on them.

8.2 Compiling Shaders with gslc

Like many other programming language systems, shaders must be *compiled*. That is, they must be translated from human-readable form ("source code") into an encoded version that is ready for the renderer to process ("object code"). This extra step also serves another purpose — it allows the shader compiler to check your shader for errors before it is in the middle of rendering a frame.

Gelato shaders are compiled using a utility called `gslc`:

`gslc [options] sourcefile`

The source file may have any name you wish, but by established convention, Gelato shader source code is stored in a file whose name is the same as the name of the shader[1], with the extension `.gsl`. For example, if you had a "plastic" shader, you would store its source code in the file `plastic.gsl`. `gslc` can compile only one source file per invocation.

Assuming that there is no error found in the shader at compile time, `gslc` will write the resulting object code to a file called *shadername*`.gso`, where *shadername* is the name of the shader.

8.2.1 Command line arguments

The `gslc` program takes the following command line arguments:

`-I`*path*

Just like a C compiler, the `-I` option, followed immediately by a directory name (without a space between `-I` and the path), will add that path to the list of directories which will be searched for any files that are requested by any `#include` directives inside your shader source. Multiple directories may be specified by using multiple `-I` options.

`-D`*symbol*
`-D`*symbol*`=`*value*

Defines a preprocessor macro symbol. If no *value* is supplied, the macro is defined to have value `1`.

This allows, among other things, conditional compilation based on defined symbols using the `#if`, `#ifdef`, and `#ifndef` statements in your shader source code files.

`-U`*symbol*

Undefines the named symbol. That is, a symbol that ordinarily would have been set is eliminated.

[1]The name of the shader is the name that follows the `surface`, `displacement`, `volume`, or `light` statement in the shader

`-o` *name*

Overrides the filename of the resulting compiled shader.

`-verbosity` *level*

Controls the level of chit-chat. Level 0 remains fairly quiet except for error reporting. Level 1 (the default) also prints warnings. Level 2 may have additional information about the compilation process.

`-debugdso`

Prints extra information about all DSO/DLL loads, which may be helpful when debugging why your DSO shadeops are not being found.

`-x`

Encrypts the resulting `.gso` file. This is useful if you are distributing compiled shaders and do not want them easily disassembled.

Even with `-x`, the file header and information about shader parameters are not encrypted, so that `libgsoargs`, `gsoinfo`, or user programs may inquire about shader parameters. Also note that although the encryption should be enough to deter casual inspection of your shaders, we make no guarantees that it is perfectly secure (so don't use it for national security).

8.2.2 Using the preprocessor

`gslc` uses the "C preprocessor" (`/usr/bin/cpp` on Linux systems). This allows you to use the usual C/C++ preprocessor directives such as `#include`, `#define`, and so on.

If your shader uses the `#include` preprocessor directive to "include" another file, `gslc` will need to know where to find the file. By default, it will only look in the current directory. You can specify extra directories to search for included files by using the `-I` command-line argument. For example,

```
gslc -I/usr/local/shaders/include myshader.gsl
```

will look in the directory `/usr/local/shaders/include` for any `#include`'d files. You can specify multiple directories with multiple `-I` arguments.

Since shaders are passed through the preprocessor, you can also define and use macros (with `-D` or with `#define` in the source code) or use "conditional compilation" (`#if`, `#ifdef`, `#ifndef`, `#else` , `#endif`).

To aid in writing shaders that can be compiled for a variety of renderers, `gslc` predefines the preprocessor variable: `GELATO`.

8.3 Listing shader parameters with `gsoinfo`

The `gsoinfo` program reports the type of a shader and its parameter names and default values. For example,

```
gsoinfo plastic
```

reports:

```
surface plastic
float Ka 1
float Kd 0.5
float Ks 0.5
float roughness 0.1
color specularcolor [1 1 1]
```

When the `-v` flag is used, the following more verbose output is generated:

```
surface "plastic"
    "Ka" "float"
                Default value: 1
    "Kd" "float"
                Default value: 0.5
    "Ks" "float"
                Default value: 0.5
    "roughness" "float"
                Default value: 0.1
    "specularcolor" "color"
                Default value: "rgb" [1 1 1]
```

Note that `gsoinfo` can only report the default values for parameters that are given defaults by simple assignment. In other words, if a constant (or a named space point) is used as the default value, `gsoinfo` will report it correctly, but if the default value of a parameter is the result of a function, complex computation, or involves a graphics state variable, `gsoinfo` will simply report `"Unknown default value"`.
Options include:

`-v`

Verbose mode. See above example as an illustration of the verbose mode versus the rather terse default mode. Verbose mode also prints shader metadata, if present.

`-p` *path*

Sets the search path to use for shaders.

Developers who are interested in writing applications that, like `gsoinfo`, can find informaiton about shader parameters and their default values should refer to Chapter 15 (which even contains the full source code to the `gsoinfo` program).

8.4 Shader Metadata Conventions

Shader metadata (see Section 5.4) allows arbitrary annotations about the shader or its individual parameters to be embedded in the compiled shader. That data may be retrieved later by applications using the `GsoArgs` class (Chapter 15) or by the `gsoinfo` utility (Section 8.3).

Currently, no metadata are semantically meaningful. Furthermore, Gelato does not enforce any particular use of any metadata. However, we think it is wise to recommend certain metadata conventions for common uses. This is similar to the use of arbitrary shader parameters — Gelato does not force shaders to have a `Kd` parameter, nor require it, if present, to be a float that denotes diffuse reflectivity. But it's a useful convention to do so, and the world is a kinder place if a shader writer can count on `Kd` doing the same thing in a wide range of shaders.

In that spirit, the remainder of this section lists our recommendations for shader metadata conventions. These conventions are not intended to be comprehensive, nor to fit all your needs — merely to establish a common nomenclature for the most universal metadata that people will probably want. Feel free to use them, ignore them, or establish your own extensions. Also feel free to inform us of clever uses for metadata; we will add the best ones to this list in future releases.

In order to protect against future conflicts (especially if there are future reserved or semantically-meaningful metadata) and to ensure metadata interoperability with other shader writers, we recommend that any studio-specific metadata be prefixed with your studio name, or use some other similar practice to avoid name collisions with other studios.

8.4.1 Recommended shader metadata

These metadata are for the shader as a whole.

`string description`

Describes the purpose and use of the shader.

`string URL`

Provides a URL for full documentation of the shader.

`float layer`

If present and having a nonzero value, denotes a shader that is intended to be used strictly as a layer inside a shader group (that is, it does not set `C` or otherwise do enough to be used as a full material). If present and having a zero value, the shader is complete enough to be used alone (or if a layer, should generally be the last layer). If absent, not enough is known to conclude whether it is safe to use as a layer or as a full material.

8.4.2 Recommended parameter metadata

These metadata are for an individual shader parameter. Many are hints for any user interface for adjusting parameter values. By convention we always prefix such metadata with "UI."

`string description`

Describes the purpose and use of the parameter. Not necessary if the meaning is obvious from the parameter name itself.

`string URL`

Provides a URL for full documentation of the parameter.

`string units`

Gives the assumed units, if any, for the parameter (e.g., `"cm"`).

`string UIlabel`

A short label to be displayed in the UI for this parameter. If not present, the parameter name itself should be used as the widget label.

`float UImin`
`float UImax`

Minimum and maximum values for a `float` parameter that should never be exceeded. If either the minimum or maximum is not present, valid values for the parameter are assumed to be unbounded in one or both directions.

`float UIsoftmin`
`float UIsoftmax`

"Soft" minimum and maximum value for a `float` parameter, that is, a default range for presentation in the UI. But the UI may allow values outside this range if the user chooses to override the limits.

`float UIstep`

The suggested step size for incrementing or decrementing the value (within the appropriate min/max range).

`string UIenabler`

Gives the name of another parameter that, only if nonzero (or the non-empty string) enables adjustment of this parameter. For example, a parameter called `"use_shadows"` may be en enabler for parameters `"shadowname"` and `"shadowbias"`; the shadow-related controls are unused and need not be presented in the GUI if the `"use_shadows"` flag is zero.

`float UIseparator`

If nonzero, hints that the application's GUI should draw a visual separator immediately before (or above) this parameter.

`string UIgroupbegin`
`string UIgroupend`

Denote the first and last parameters of a group. In both cases, the string value should be the name or brief description of the group. Groups may be used as a hint for the application to label and visually separate the group of shader parameters.

`string UItype`

Each shader parameter will probably have a UI widget appropiate to its type: a slider for `float` parameters, a color picker for `color` parameters, a place to type in text for a string parameter, etc. The `UItype` metadatum gives a more detailed description of the intended type or use of the parameter, to allow more intelligent widget selection. Recommended values include:

`"float"` A slider and/or numeric input (default for float parameters).

`"bool"` Indicates that a float parameter should only take on a 0 or 1 value, and therefore a checkbox may be a more appropriate UI widget.

`"int"` Indicates that a float parameter should only take on integer values.

`"color"` Indicates that the appropriate UI widget is a color picker (should be the default for color parameters).

`"string"` Indicates that the appropriate UI widget is a text input box (default for string parameters). A smart UI may also allow a file selection dialog, since most strings (but not all) are used for texture names.

`"texture"` Indicates a string variable whose value will be used as the name of a texture map. An appropriate GUI widget might be a file selection dialog.

`"environment"` Indicates a string variable whose value may either be an environment map or the name of a geometry set. An appropriate GUI widget might be a combination file selection and pull-down menu of known geometry set names.

`"shadow"` Indicates a string variable whose value may either be a shadow map or the name of a geometry set.

`"geometryset"` Indicates a string variable whose value must be a geometry set. An appropriate GUI widget may be a pull-down menu of known geometry sets.

`"coordsys"` Indicates a string variable whose value will be used as a coordinate system name. An appropriate GUI widget may be a pull-down menu of known coordinate systems (including the built-in ones like `"world"`).

`"enum:label0,label1,..."` Indicates that the appropriate UI widget is a pulldown menu that selects among the strings `"label0"`, `"label1"`, and so on.

If the shader parameter is a string, the selcted label should be passed as the shader parameter value.

If the shader parameter is a float, the selected string should indicate that the value passed should be 0, 1, 2, ..., the index of the selected label.

Labels may be associated with float values other than their indices using the following syntax:

```
float frequency = 0.5
    [[ string UItype = "enum:low=0.2,medium=0.5,high=0.9" ]]
```

Remember that the presence of these user interface hints does not require an application to honor the request, nor is the shader-writer required to use any of the hints. But we hope that application developers try to support as many of the recommended metadata as practical, and that shader developers try to include thorough descriptions and UI hints in all their shaders.

8.5 Using Units

Neither Gelato nor most modeling systems dictate the *units* used for your scene. A particular scene may have world-space units be inches; another scene may have world-space units be centimeters. Furthermore, the times passed to `Motion` (and the `Camera`'s `"shutter"` parameter) are arbitrary units.

However, it is very useful to write shaders that understand physical units. Advantages include:

- Shaders can make features that correspond to real-world materials.
- Shaders can be made *portable* so that they produce consistent results no matter what the scene modeling units are. This allows you to re-use shaders across scenes and productions when your standard units change, without needing to readjust parameters or hard-coded constants within the shader.
- Shaders that are aware of radiometric units can be more physically realistic by modeling the behavior of light in a consistent and correct manner.

8.5.1 Declaring Units

The first step to using units effectively is to ensure that your scene declares its units. This is done using several scene-wide `Attribute` calls, as described formally in Section 4.3.2.

The `"string units:length"` and `"float units:lengthscale"` set the name and scaling of the units of `"common"` space. Valid unit names for `"units:length"` include `"mm"`, `"cm"`, `"m"`, `"km"`, `"in"`, `"ft"`, `"mi"`. For example, to explain to the renderer that a length of 1 in `"common"` space corresponds to 0.5 cm:

```
Attribute ("string units:length", "cm")
Attribute ("float units:lengthscale", 0.5)
```

The `"string units:time"` and `"float units:timescale"` similarly set the time units used for `Motion` and the `Camera`'s `"shutter"` parameter, and the `"float units:fps"` specifies the number of frames per second for animation. Valid unit names for `"units:time"` include `"s"` and `"frames"`. For example, to explain to the renderer that we are rendering 29.997 frames per second (NTSC) and that common (i.e., `Motion`) units are frames:

```
Attribute ("string units:time", "frames")
Attribute ("float units:timescale", 1.0)
Attribute ("float units:fps", 29.997)
```

8.5.2 Unit conversion in GSL

Remember that all spatial quantities (`point`, `vector`, `normal`, `matrix`) are converted into a canonical `"common"` coordinate system before a shader begins execution, and GSL functions such as `transform()`, `transformv()`, etc., may convert spatial quantities between coordinate systems (including from and to `"common"` space).

When using units, `"common"` space measurements are understood to be in canonical units, and the `transformu()` function can convert measurements among unit systems (including into and out of the canonical `"common"` space units). Please consult Section 5.8.3 for the formal definition of the `transformu()` function. The quick description is that `transformu(from,to,x)` returns the *to* units equivalent of the quantity *x* (in *from* units). The unit names may be either physical units (`"mm"`, `"in"`, `"s"`, etc.), or may be the name of any known coordinate system (such as `"common"`, `"world"`, or any coordinate system named with `SaveAttributes`). For example:

- `transformu ("common", "m", 1.0)` returns the length, in meters, corresponding to a length of 1 in `"common"` space.
- `transformu ("cm", "common", amplitude)` returns the length of amplitude (expressed in cm) as a `"common"` space measurement.
- `transformu ("in", "cm", 5)` returns the centimeter equivalent of 5 inches.
- `transformu ("mm", "shader", d)` tells you what distance in `"shader"` space corresponds to a length of *d* millimeters.
- `transformu ("s", time)` returns the value of the `time` variable, expressed in seconds (its raw value may be in arbitrary units).

8.5.3 Length-aware shaders

Individual productions may choose different "modeling units," for example, cm, mm, or km, depending on whether the film is about toys, insects, or space stations. Units may be different from one production to another within a single studio, making it difficult to reuse shader assets among productions. But with units, it is easy to create a shader that makes features corresponding to real-world sizes that look correct regardless of the modeling units, as in the following example that makes 1 mm-deep bumps with a period of 0.5 cm:

```
displacement bumpy (float amp = 1 /* mm */,
                    float period = 5 /* mm */)
{
    point Pshad = transform ("shader", P);
    float periodshad = transformu ("mm", "shader", period);
    float n = noise (Pshad / periodshad);
    n *= transform ("mm", "common", amp);
    displace (n);
}
```

8.5.4 Time-aware shaders

Conventions differ as to whether time (such as passed to `Motion`) is measured in seconds or frames, and in either case the number of frames per second may be 24, 25, 29.997, or 60 depending on the output medium (film, PAL, NTSC, or Showscan, respectively). Thus, it was previously impossible to write a shader with time-varying behavior that was truly portable among the various time and frame conventions. But with units it is easy to make a portable shader, as in the following example that makes a pulsing light with a 3-second period:

```
light pulse (float intensity = 1,
             color lightcolor = 1)
{
    float period = transformu ("s", "common", 3);
    emit (point ("shader", 0, 0, 0)) {
        float amp = 0.5 + 0.5 * sin (time/period);
        Cl = amp * intensity * lightcolor;
    }
}
```

9 Textures

9.1 Converting images to texture with maketx

The maketx program converts 2D images into multiresolution, tiled texture files and will combine six views into a cube face environment map.

9.1.1 General Options

The following maketx command-line options can be used for any kind of map:

-format *name*

Specifies the format to write texture files – that is, the ImageIO plugin to use when writing the texture file. The default is to use TIFF. Texture files must support tiles and multi-resolution files, so only formats that allow this may be used. The only formats that ship with Gelato suitable for textures in this way are currently "tiff" and "OpenEXR".

-noresize

Textures are stored in a multi-resolution format (a.k.a. MIP-map). By default, the top level will have its resolution rounded up to the next highest power of 2 in each dimension (unless it already is a power of 2). When the -noresize option is used, the top level of the MIP-map will be the same resolution as the original input image, rather than rounded up to power-of-two sizes.

-u

Update mode. If the output file already exists and has the same time stamp as the input file(s), the texture will not be remade (thus saving time and disk access). If the output file does not already exist, or does exist but has a different time than the input files, then the texture will be recreated and the new texture will get the time stamp of the input file. (Note than when not using update mode, texture files are created unconditionally and will be stamped with the current time.)

-p *searchpath*

Specifies a searchpath for image files. The searchpath is a colon-separated list of directories to search for input files. If no searchpath is specified, input files are assumed to either be in the current directory or are absolute paths.

`-separate`

Write the resulting texture file as an image using the “separate” planar configuration (instead of the default “contiguous” planar configuration). The native contiguous method is more efficient, but the separate method allows the files to be shared with other renderers that only support the reading of “separate” planar configuration files.

`-tilesize` *t*

Overrides the tile size of the resulting texture, allowing you to specify the tile size explicitly. This allows the files to be shared with other renderers that only support the reading of images with certain tile sizes.

`-ingamma` *g*

NEW!

Declares that the input files are already gamma corrected, having had their pixel values raised to the $1/g$ power. Therefore, the inverse of this correction will be applied automatically to all the input files so that the MIPmap math will be correct. Only the first three channels in the image will be so corrected, so that alpha values are unchanged.

`-outgamma` *g*

NEW!

Causes all pixels in the output image to be gamma corrected, i.e., raised to the $1/g$ power. *We recommend keeping textures in linear form, i.e., NOT gamma corrected, so that the math in your shaders will be correct.*

`-Mcamera` *m0 m1 ... m15*
`-Mscreen` *m0 m1 ... m15*

Sets the camera and screen matrices (sometimes called Nl and NP, respectively) in the texture file. This overrides any such matrices present in the input file (which ordinarily would get passed through to the output texture unmodified).

`-debugdso`

Prints extra information about all DSO/DLL loads, which may be helpful when debugging why your imageio plugins are not being found.

9.1.2 Texture Maps

The `maketx` program can be used to convert 2D image files—in any format for which you have an `imageio` plugin (see Chapter 13)—to texture files. The texture files are specially constructed to be very memory-efficient for texture access. Specifically, they are both MIP-mapped and *tiled*, so small bits of texture can easily be read on demand rather than requiring the renderer to read entire MIP-map levels or images.

`maketx [`*options*`]` *imagefile* `-o` *texturefile*

This converts ordinary 2D image files into a special tiled multiresolution format (actually, it's a multi-page tile-oriented TIFF file). Using textures in the tiled multiresolution format offers significant performance improvements over using the original image files directly.

Options include:

`-smode` *wrapmode*
`-tmode` *wrapmode*
`-mode` *wrapmode*

Sets the default *wrap mode* of the texture to one of: `periodic`, `black`, `clamp`, or `mirror`. The `-smode` and `-tmode` flags specify wrapping behavior separately for the *s* and *t* directions, while `-mode` specifies both at the same time. If none of these options are set, the default wrap mode will be `black`.

The wrap mode specifies the behavior of the texture when outside the [0,1] lookup range.

Note that this merely sets the *default* wrap mode for a texture. A shader may completely override the wrap mode by using the optional `"wrapmode"` parameter to the `texture()` function.

9.1.3 Environment Maps

`maketx -envcube [options]` *px nx py ny pz nz* `-o` *envfile*

This command takes six image files (all square and of the same resolution) and combines them into a cubeface environment map. Options include:

`-fov` *angle*

Specifies the field of view of the faces.

`maketx -envlatl [options]` *imagefile* `-o` *envfile*

This command takes a single ordinary image file representing a latitude-longitude reflection map, and converts it to a "latlong" environment map.

`maketx -lightprobe` *lpfile* `-o` *envfile*

This command takes an image in spherical "lightprobe" format and converts it to a cube face environment map (see `http://www.debevec.org/Probes` for the angular formulas).

`maketx -vertcross` *`vcfile`* `-o` *`envfile`*

NEW!

This command takes an image in the "vertical cross" configuration and converts it to a Gelato cube face environment map. The vertical cross configuration is an image that is laid out as follows:

(b)	py	(b)
nx	pz	px
(b)	ny	(b)
(b)	nz	(b)

(b) indicates blank/unused

9.1.4 Shadow Maps

`maketx -shadow [options]` *`depthfile`* `-o` *`shadowfile`*

If you've written out a depth map as a single-channel floating-point image, this command converts it to a shadow map. This step is unnecessary if you write depth maps directly as shadow maps using the `"shadow"` driver.

`maketx -shadcube [options]` *`px nx py ny pz nz`* `-o` *`shadowfile`*

This command takes six shadow maps or z images (each assumed to have a 90deg field of view) and combines them into a cubeface shadow map from which shadows may be sampled from any direction using a single `shadow()` call in a shader.

It is allowed to use `"-"` (that is, just a dash) as one or more of the file names. Any shadow cube face whose name is supplied as simply a dash will be assumed to be an empty z-buffer. That is, it is not necessary to create an empty shadow file for faces that you know will not cast any shadows. However, since the *pz* face supplies the matrix that is used to position the shadow cube, it is necessary to use the `-Mcamera` and `-Mscreen` if you don't supply a real shadow map for the *pz* face.

`maketx -volshad [options]` *`voldepthfile`* `-o` *`volshadowfile`*

Converts a single-level volume depth map (specified when the renderer `Output` is of data `"volz"`, as described in Section 10.1.3) into a fully MIP-mapped volume shadow map.

When making a volume shadow map, an additional option is recognized:

`-opaquewidth` *`f`*

The optional `-opaquewidth` argument, which defaults to 0.05 if not specified on the command line, specifies the z range over which close values of z_{opaque} within adjacent texels are considered "the same" when combined into a single texel at a lower-res level of the MIP-map. This is exactly analogous to the `Output` option `"opaquewidth"` that applies to combining multiple samples within a pixel when rendering out a single-level `"volz"` file.

9.2 Texture Formats

This section documents the formats created by `maketx`. These are the preferred texture formats read by Gelato, but other formats may be used where noted.

9.2.1 TIFF Texture Common Features

`maketx` uses TIFF files as its preferred texture format. This subsection details the specifications that, unless noted later, are common to all texture file types (plain textures, shadows, environment maps, etc.). This section is only describing the *output* of `maketx` (i.e., the texture format itself read directly by Gelato when rendering), not the *input* of `maketx` (the much wider variety of image formats that may be converted to Gelato textures).

MIP-mapped levels are all stored in one physical file, using one image subfiles per MIP-map level, starting with the highest-resolution MIP-map level and progressing to successively lower-resolution levels. Each successive level is exactly half the resolution of the immediately-previous level (obviously rounding, in the case of odd resolutions), until the it reaches a 1×1 texture.

All texture files are “full color images.” The `maketx` program never outputs palette or bilevel images. Any number of channels (R, G, B, alpha, etc.) may be stored in a texture, but all channels must be the same data format (byte, short, or float).

All texture files are *tiled* and `maketx` typically uses a tile size of 64×64 texels. For MIP-map levels that are smaller than 64×64, the tile size may be reduced for those levels, but will still be a power of 2. Any MIP-map level whose resolution does not evenly fill an integral number of tiles will be padded with black pixels. The default tile size may be overridden on the `maketx` command line, but it is strongly recommended to use a tile width and length that are each a power of 2.

9.2.2 Plain Textures

Plain textures have the following additional TIFF tags (using the nomenclature of `libtiff`):

`TIFFTAG_PIXAR_TEXTUREFORMAT` (TIFF tag: 33302, type: string)

> Contains the value `"Plain Texture"`, to identify an ordinary 2D texture. Gelato Image I/O plugins may be read or write this tag using the parameter `"string textureformat"`.

`TIFFTAG_PIXAR_WRAPMODES` (TIFF tag: 33303, type: string)

> Contains the the horizontal and vertical wrap modes, separated by a comma (or just one mode name, if horizontal and vertical use the same wrap mode). Valid wrap modes include: `"black"`, `"clamp"`, `"periodic"`, `"mirrow"`. Gelato Image I/O plugins may be read or write this tag using the parameter `"string wrapmodes"`.

9.2.3 Shadow Maps

Ordinary and Woo shadow maps are not MIP-mapped, and therefore are expected to have only a single image subfile.

Shadow maps have the following additional TIFF tags (using the nomenclature of `libtiff`):

`TIFFTAG_PIXAR_TEXTUREFORMAT` (TIFF tag: 33302, type: string)

Contains the value `"Shadow"`, to identify a shadow map. Gelato Image I/O plugins may be read or write this tag using the parameter `"string textureformat"`.

`TIFFTAG_PIXAR_MATRIX_WORLDTOSCREEN` (TIFF tag: 33305, type: 16 floats)

The full light projection matrix that transforms points from world space into a 2D screen space ranging from -1 to 1 in x and y. Gelato Image I/O plugins may be read or write this tag using the parameter `"matrix worldtoscreen"`.

`TIFFTAG_PIXAR_MATRIX_WORLDTOCAMERA` (TIFF tag: 33306, type: 16 floats)

The light placement matrix that transforms points from world space into the light's local 3D coordinte system. Gelato Image I/O plugins may be read or write this tag using the parameter `"matrix worldtocamera"`.

9.2.4 Volume Shadow Maps

The meaning of the channels of volume shadow maps are as follows:

Channel	Meaning
0	z_{opaque}, the depth at which opacity is 1, or fully opaque (or 1e30 if the pixel never reaches full opacity).
1	α_{trans}, the total alpha of the transparent portion (not counting the opaque surface, if present), i.e., the accumulated opacity at z_1.
2	z_0, the farthest depth at which the volume's accumulated opacity is still 0, i.e., the start of the volumetric medium.
$\vdots$	
$2+i$	z_{a_i}, the depth at which the accumulated opacity is $a_i \cdot \alpha_{\text{trans}}$, where n is the total number of transparent z channels stored (given by the `"zchannels"` parameter), i takes on values from $1 \ldots n-2$, and $a_i = i/(n-1)$.
$\vdots$	
nchannels–1	z_1, the depth at which the accumulated opacity is α_{trans}, i.e., the end of the volumetric medium prior to z_{opaque}.

The number of opacity thresholds, n, is simply *nchannels*–2, and currently they are required to be evenly spaced between 0 and 1.

Volume shadow maps have the following additional TIFF tags:

`TIFFTAG_PIXAR_TEXTUREFORMAT` (TIFF tag: 33302, type: string)

Contains the value `"Volume Shadow"`, to identify a volume shadow map. Gelato Image I/O plugins may be read or write this tag using the parameter `"string textureformat"`.

`TIFFTAG_PIXAR_MATRIX_WORLDTOSCREEN` (TIFF tag: 33305, type: 16 floats)
`TIFFTAG_PIXAR_MATRIX_WORLDTOCAMERA` (TIFF tag: 33306, type: 16 floats)

The light projection and light placement matrices, just as with ordinary shadow maps.

9.2.5 Cube-Face Shadow Maps

Cube-face shadow maps combine six orthogonal 90° shadow map views into a single shadow map file. This allows an omnidirectional shadow lookup to be performed with a single `shadow()` call in GSL. Section 9.3 for a table of the cube face directions and their meanings. The six faces are assembled into a "cubeface mosaic image" with the following layout:

px	py	pz
nx	ny	nz

No single image tile will contain texels from more than one cube face. That is, if the resolution of a single shadow map face is not an integral multiple of the image tile size, then the face data will be padded so that the upper-left corner of each face always aligns with a tile boundary.

Cube-face shadow maps have the following additional TIFF tags:

`TIFFTAG_PIXAR_TEXTUREFORMAT` (TIFF tag: 33302, type: string)

Contains the value `"CubeFace Shadow"`, to identify a cube-face shadow map. Gelato Image I/O plugins may be read or write this tag using the parameter `"string textureformat"`.

`TIFFTAG_PIXAR_MATRIX_WORLDTOSCREEN` (TIFF tag: 33305, type: 16 floats)
`TIFFTAG_PIXAR_MATRIX_WORLDTOCAMERA` (TIFF tag: 33306, type: 16 floats)

The light projection and light placement matrices indicate the transformation specifically for the *pz* (positive *z*) face. The transformations for other faces may be straightforwardly derived from these matrices.

`TIFFTAG_IMAGEWIDTH` (TIFF tag: 256, type: int)
`TIFFTAG_IMAGELENGTH` (TIFF tag: 257, type: int)

Contains the width and length of the "cubeface mosaic image" containing the assembly of all six faces, not the resolution of an individual face.

`TIFFTAG_PIXAR_IMAGEFULLWIDTH` (TIFF tag: 33300, type: int)
`TIFFTAG_PIXAR_IMAGEFULLLENGTH` (TIFF tag: 33301, type: int)

Contains the width and height of an individual cube face, *not* the resolution of the "cube-face mosaic image," which is the assembly of all six faces. Gelato Image I/O plugins may be read or write this tag using the parameter `"int fullwidth"` and `"int fullheight"`.

9.2.6 Latitude-Longitude Environment Maps

Latitude-longitude environment maps are simple 2D maps with a spherical parameterization and left handed coordinates. The "north pole" is $+z$ and corresponds to $t = 0$ in the map, the "south pole" is $-z$ corresponds to $t = 1$, and the "seam" ($s = 0$ or $s = 1$) intersects the xy plane at $(x = -1, y = 0)$. The center of the map corresponds to the $+x$ axis.
Latitude-longitude environment maps have the following additional TIFF tags:

`TIFFTAG_PIXAR_TEXTUREFORMAT` (TIFF tag: 33302, type: string)

> Contains the value `"LatLong Environment"`, to identify a latitude-longitude environment map. Gelato Image I/O plugins may be read or write this tag using the parameter `"string textureformat"`.

9.2.7 Cube-Face Environment Maps

Cube-face environment maps combine six orthogonal 90° images into a single file from which the color may be looked up in any direction. Section 9.3 for a table of the cube face directions and their meanings. The six faces are assembled into a "cubeface mosaic image" with the following layout:

px	py	pz
nx	ny	nz

No single image tile will contain texels from more than one cube face. That is, if the resolution of a single cube map face is not an integral multiple of the image tile size, then the face data will be padded so that the upper-left corner of each face always aligns with a tile boundary. This is particularly important to remember for MIP-map levels where the cube face itself is smaller than even a single tile. For example, the MIP-map level where each face is 4×4 pixels may have 16×16 texel tiles, so that level's full mosaic image will be 48×32, with appropriate padding of black pixels so that faces are aligned with image tile boundaries.
Cube-face environment maps have the following additional TIFF tags:

`TIFFTAG_PIXAR_TEXTUREFORMAT` (TIFF tag: 33302, type: string)

> Contains the value `"CubeFace Environment"`, to identify a cube-face environment map. Gelato Image I/O plugins may be read or write this tag using the parameter `"string textureformat"`.

`TIFFTAG_IMAGEWIDTH` (TIFF tag: 256, type: int)
`TIFFTAG_IMAGELENGTH` (TIFF tag: 257, type: int)

> As usual, contains the width and length of each level of the MIP-map, but please note that this is the resolution of the "cubeface mosaic image" containing the assembly of all six faces, not the resolution of an individual face.

`TIFFTAG_PIXAR_IMAGEFULLWIDTH` (TIFF tag: 33300, type: int)
`TIFFTAG_PIXAR_IMAGEFULLLENGTH` (TIFF tag: 33301, type: int)

Contains the width and height of an individual cube face at this level of the MIP-map, *not* the resolution of the "cubeface mosaic image," which is the assembly of all six faces. Gelato Image I/O plugins may be read or write this tag using the parameter `"int fullwidth"` and `"int fullheight"`.

9.2.8 Other Texture Formats

The previous sections describe the tiled, multiresolution TIFF files that Gelato typically uses for texture, environment, and shadow maps. This is the preferred format, and the format that `maketx` writes by default. But it is by no means required.

Gelato will directly read alternate texture formats, assuming that the appropriate Image I/O plugin for reading he format is available. The formats *must* be tiled, and multi-resolution (i.e., MIP-mapped) is strongly recommended for texture and environment maps.

Using the optional `-format` command-line argument, you can have `maketx` write texture files in other formats. This ability is limited to formats that are capable of writing tiled, multiresolution images, and for which the appripriate Image I/O plugin is available.

The main reason for wanting to use another format is if you require an ability that TIFF cannot support. For example, you may wish certain texture files to be stored as OpenEXR, since it supports 16-bit float values, and TIFF currently does not. At the present time, OpenEXR is the only format we are aware of that is comparable in flexibility to TIFF and supports the various tiled and multiresolution modes that are critical to good texture lookup performance.

The following specific notes may be helpful when using the OpenEXR formats for textures:

- By convention, OpenEXR latitude-longitude environment maps use a right-handed coordinate system and have the "north pole" as the $+y$ axis (as opposed to our TIFF convention of left-handed and $+z$ being "up"). Gelato automatically adjusts lat-long environment map lookups to the alternate coordinate system when it detects that a particular environment map came from an OpenEXR file.

- By convention, OpenEXR cube-face environment maps use a right-handed coordinate system and arrange the faces in a 1×6 mosaic in the order, from top to bottom: px, nx, py, ny, pz, nz (as opposed to our TIFF convention of left-handed and arranging the faces in a 3×2 mosaic pattern).

- Gelato has no trouble directly reading OpenEXR whose data is stored in 16-bit floating point numbers.

9.3 Cube-Face Directions

The following table gives the directions and matrices for each face of cube-face environment and shadow maps.

Face	Face view	Axis toward top	Axis toward right	Camera matrix
px (positive x)	$+x$	$+y$	$-z$	$\begin{bmatrix} 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
nx (negative x)	$-x$	$+y$	$+z$	$\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
py	$+y$	$-z$	$+x$	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
ny	$-y$	$+z$	$+x$	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
pz	$+z$	$+y$	$+x$	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
nz	$-z$	$+y$	$-x$	$\begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

10 Illumination: Shadows, Reflections, Global Effects

10.1 Shadows

Shadows are crucial to lighting a scene in a believable way. Gelato allows two main techniques to generate shadows: shadow maps (of which there are multiple types) and ray tracing.

10.1.1 Shadow Maps

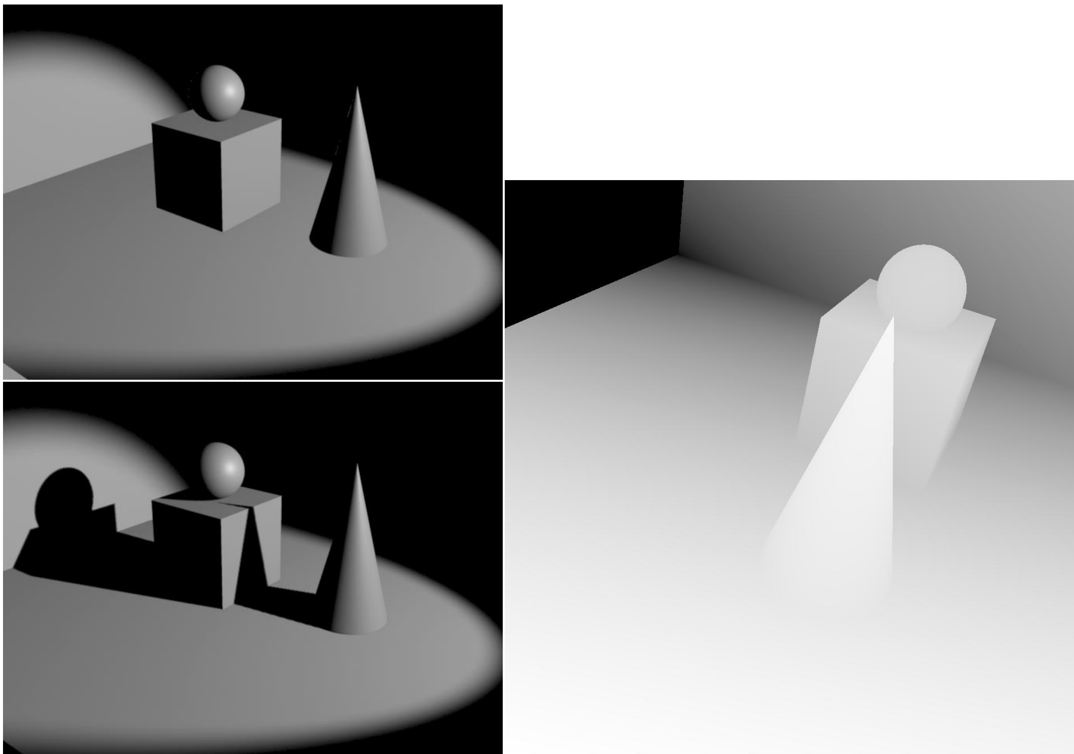

Figure 10.1: Shadow depth maps. A simple scene with and without shadows (left). The shadow map is just a depth image rendered from the point of view of the light source (right). To visualize the map, we assign white to near depths, black to far depths.

Shadow maps (also sometimes called *shadow depth maps*) are a simple, relatively cheap, and very flexible means to cause a light to cast shadows. The shadow map algorithm works in the following manner. Before rendering the main image, separate images are rendered *from the vantage points of the lights*. Rather than render RGB color images, these light source views record depth only (hence the name, *depth map*). Figure 10.1 shows a simple scene with and without shadows, as well as the depth map that was used to produce the shadows. Most modeling systems geared toward generating input for Gelato will automatically position and render the depth maps for each shadowing light source.

Once these depth maps have been created, the beauty pass is rendered from the point of view of the camera. In the beauty pass, the light shader can determine if a particular surface point is in shadow by comparing its distance to the light against that stored in the shadow map. If it matches the depth in the shadow map, it is the closest surface to the light in that direction, so the object receives light. If the point in question is *farther* than indicated by the shadow map, it indicates that some other object was closer to the light when the shadow map was created. In such a case, the point in question is known to be in shadow.

Gelato supports three types of shadow maps: ordinary depth maps, Woo shadow maps, and volume shadow maps. For all methods, a single shadow map may be used, or six shadow maps may be combined by form a cube-face shadow map.

The basic shadow map technique was pioneered by Lance Williams ("Casting Curved Shadows on Curved Surfaces," *Computer Graphics* 12(3), SIGGRAPH '78 Proceedings, pp. 270–274) and further refined by Reeves, et al. ("Rendering Antialiased Shadows with Depth Maps," *Computer Graphics* 21, SIGGRAPH '87 Proceedings, pp. 283–291). Woo shadows were documented by Andrew Woo in *Graphics Gems III*.

Creating an ordinary shadow map

Shadow maps are generated in a separate rendering pass. When creating a shadow map, only depth (z) is needed, color is not used. Also, we recommend creating depth maps with the `"min"` filter and using a small number of samples per pixel (usually 1×1 is adequate, or 2×2 if you want to be especially careful about plugging cracks or holes that sometimes occur). Gelato has a special image format for shadow maps, using the `shadow` image I/O plugin. Putting this all together, here are the commands that actually cause a render to create a shadow map:

```
Parameter ("int[2] spatialquality", (1,1))
Camera (...)
...
Output ("shadow.sm", "shadow", "z", "camera",
        "float[4] quantize", (0,0,0,0), "float dither", 0,
        "string filter", "min", "float[2] filterwidth", (1,1))
```

When rendering a shadow map pass, the scene must be rendered from the point of view of the light.

For "finite" lights such as point or spot lights, that come from specific positions, the shadow map should be made using a perspective projection, and the position of the light camera should be the point where the light orginates.[1] The direction and field of view of the light camera is less

[1]Strictly speaking, the shadow need not originate from the same position that the light shader uses as the point of origin of the light. That is, the illumination may appear to come from a different location than the shadows are cast

important and does not need to match the direction and fov of the spotlight that uses the shadow map, but obviously the view must be chosen to include any objects which may cast shadows for that light.

For "infinite" or distant lights with parallel rays, an orthographic projection should be used, the light may be positioned anywhere such that it views shadow-casting objects, but the light direction should match the direction of the directional light in order to look correct.

The field of view (for shadow cameras with perspective projections) or the screen window (for orthographic projections) should be adjusted so that all of your shadow map's resolution (and expense) is concentrated on objects that actually cast shadows; large blank regions in the shadow map are simply wasted.

Since shadow maps store only depth, and not color, it is wasteful to run compicated surface shaders when creating a depth map. For any image in which only z is output, Gelato will automatically skip the running of surface shaders for all objects that are known to be opaque (those whose surface shader simply copies `opacity` and the attribute state's opacity is 1), and will not run light shaders even on objects whose surface shaders must still run. This speeds up the rendering of shadow maps significantly. If you have complex shaders that are impervious to simple analysis of this kind (for example, they modify `opacity` in a more complex way than simply inheriting the attribute state's opacity value), you may be able to speed up your shadow map passes by setting the Attribute `"ray:opaqueshadows"` to nonzero, which also will skip surface shader execution on a shadow pass, or you could also substitute a simpler shader (such as `"constant"`) for opaque objects on the shadow pass.

It also may speed up shadow map generation to decrease the `"shadingquality"` for any objects that do not need fine displacement or opacity variation recorded in the shadow map. This is because slight simplification of object shapes in the shadow map will generally not result in noticeable artifacts in the shadows. But this is not true of displaced objects — to cast accurate shadows, you must still run displacement shaders, and do so at a reasonably accurate shading quality.

Higher resolution shadow maps will take longer to compute and will result in larger shadow map files, but will result in higher-quality shadows. Experiment to find an appropriate resolution that avoids artifacts. General guidelines might be to use 1024×1024 shadow maps for ordinary renders and tests, 2k $\times$ 2k or even higher-resolution shadow maps for high-quality or film work.

When rendering ordinary shadow maps (but not Woo shadows), only include objects that will actually cast shadows on themselves or other objects. Objects that only receive but do not cast shadows, such as walls or floors, can be eliminated from the shadow map pass entirely. This saves rendering time when creating the shadow map and also eliminates the possibility that poorly chosen bias will cause these objects to incorrectly self-shadow (since they aren't in the maps anyway). Important caveat: Woo shadows, which greatly reduce the need to adjust bias, require shadow receivers as well as shadow casters to be in the shadow map.

Querying shadow map results from a shader

Gelato's shading language gives us a handy built-in function to access shadow maps:

```
color shadow ( string shadowname, point Ptest, ...  )
```

from. This is called "cheating the light" and is frequently used, but can lead to strange effects if the light position is cheated too much.

The `shadow()` function tests the point `Ptest` (in `"common"` space) against the shadow map file specified by `shadowname`. The return value is 0 if `Ptest` is unoccluded, and 1 if `Ptest` is occluded (in shadow according to the map). The return value may also be between 0 and 1, indicating that the point is in partial shadow (this is very handy for soft shadows). The `shadow()` call has several optional arguments that can be specified as token/value pairs, of which the most imporant and commonly modified are the following:

- `"blur"` takes a `float` and controls the amount of blurring at the shadow edges, as if to simulate the penumbra resulting from an area light source (see Figure 10.9). A blur value of 0 makes perfectly sharp shadows; larger values blur the edges. It is strongly advised to add some blur, as perfectly sharp shadows look unnatural and can also reveal the limited resolution of the shadow map. The `"blur"` value is measured against the full width of the shadow map. That is, a blur of 1 will spread the penumbra over the entire width of the map.

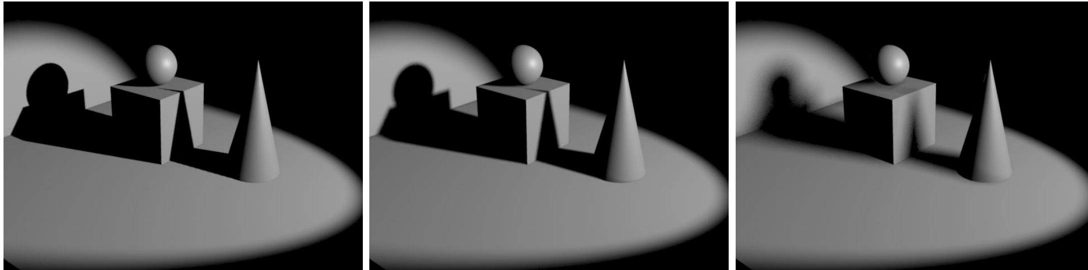

Figure 10.2: Adding blur to shadow map lookups can give a penumbra effect. Increasing blurs from left to right: 0, 0.02, 0.1.

- `"samples"` is a `float` specifying the number of samples used to test the shadow map. Shadow maps are antialiased by supersampling, so although having larger numbers of samples is more expensive, they can reduce the graininess in the blurry regions. We recommend a minimum of 16 samples, and for blurry shadows it may be quite reasonable to use 64 samples or more. Figure 10.3 shows how this can affect the quality of the shadow penumbra. Using more samples certainly makes shadow map lookups more expensive, but not nearly as expensive as a similar number of ray-traced shadow samples.

- `"bias"` is a `float` that *shifts the apparent depth of the objects from the light*. The shadow map is just an approximation, and often not a very good one. Because of numerical imprecisions in the rendering process and the limited resolution of the shadow map, it is possible for the shadow map lookups to incorrectly indicate that a surface is in partial shadow, even if the object is indeed the closest to the light. The solution we use is to add a “fudge factor” to the lookup to make sure that objects are pushed out of their own shadows.

  Selecting an appropriate bias value can be tricky. Too small a bias will lead to incorrect self-shadowing, which can manifest itself as an explicably “dirty” look to objects that should not be shadowed, as well as dark artifacts near corners. Too large a bias can result in illumination of parts of objects that should be in shadow, often leading to “floating

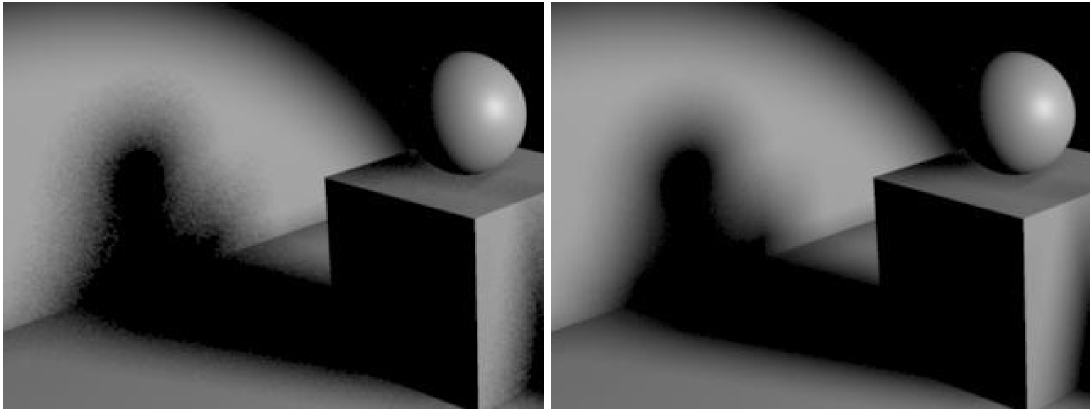

Figure 10.3: The "samples" parameter controls the amount of "noise" in the shadow penumbra. Left: 16 samples; right: 256 samples.

objects" or "detatched shadows." Figure 10.4 shows what can go wrong if you select a value that is either too small or too large. Selecting a bias which is just right for your scene sometimes takes a bit of fiddling. However, the appearance of many, if not most, scenes are much less sensitive to bias values when Woo shadows are used, as explained in the next subsection.

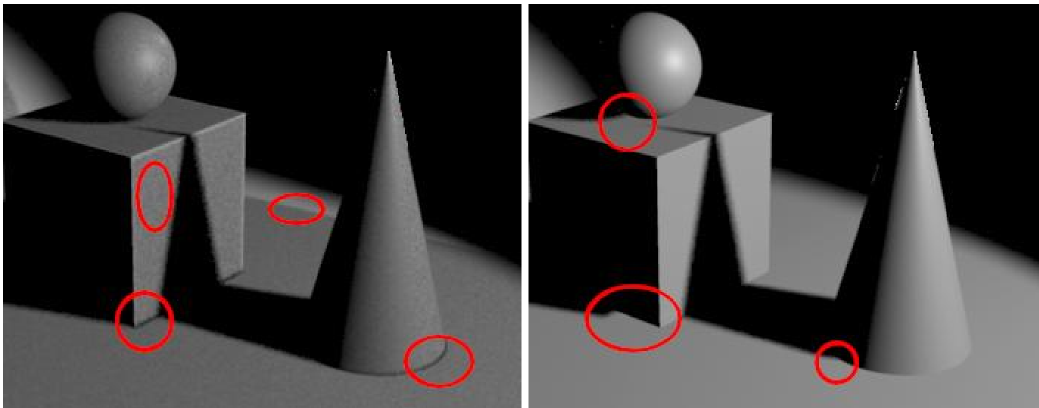

Figure 10.4: Left: Too small a bias value will result in incorrect self-shadowing (see highlighted artifacts). Right: Too much bias (right) can also introduce artifacts, such as the appearance of "floating objects" or "detached shadows," highlighted.

The Ptest parameter determines the point at which to determine how much light is shadowed, but how does the renderer know the point of origin of the light? When the renderer creates a shadow map, it also stores in the shadow file the world-to-camera transformation at the time that the shadow map was made. The shadow() function knows to look for this information in the shadow map file, and thus both the origin of the light as well as the transformation needed to map positions in space to texels in the shadow map. Notice that since the shadow origin comes from the shadow map file rather than the light shader, it's permissible (and often useful) for the

shadows to be cast from an entirely different position than the point from which the light shader illuminates. Gelato's `spotlight` shader, Listing 10.1, illustrates use of the `shadow()` function in a light shader.

Listing 10.1 `spotlight`: A simple light shader that uses a shadow call.

```
light
spotlight (
            float  intensity = 1,
            color  lightcolor = 1,
            point  from = point ("shader", 0, 0, 0),
            point  to = point ("shader", 0, 0, 1),
            float  coneangle = radians(30),
            float  conedeltaangle = radians(5),
            float  beamdistribution = 2,
            float  falloff = 2,
            string shadowname = "",
            float  shadowsamples = 1,
            float  shadowblur = 0,
            float  shadowbias = -1,
            float __nonspecular = 0
          )
{
    vector axis = normalize(to-from);

    emit (from, axis, coneangle) {
        float Llen = length(L);
        float cosangle = dot (L, -axis) / Llen;
        float atten = pow (cosangle, beamdistribution) / pow (Llen, falloff);
        atten *= smoothstep (cos(coneangle), cos(coneangle-conedeltaangle),
                             cosangle);
        Cl = atten * intensity * lightcolor;

        // If shadowing, scale by the "unshadowed" amount
        if (shadowname)
            Cl *= 1 - (color) shadow (shadowname, Ps, "samples", shadowsamples,
                                      "blur", shadowblur, "bias", shadowbias);
    }
}
```

10.1.2 Woo Shadows

The texels in ordinary shadow maps store the z depth of the closest object imaged in that direction when rendering the shadow pass. As we have seen, this leads to occasional incorrect self-shadowing of these closest surfaces, which is counteracted by adjusting shadow bias.

Gelato also supports an alternative technique, *Woo*[2] shadows, also sometimes called "mid-point shadows" or "average shadows." Rather that recording the depth of the closest surface, Woo shadows record the average of the depths of the closest and second-closest surfaces. This average depth is greater than that of the closest surface, so there is less tendency to incorrectly self-shadow. But it is still less than that of the second and subsequent surfaces, thus correctly rendering them in shadow. If only one surface is found in a pixel, the shadow map records infinite depth – that is, it behaves as if no surface were in that pixel, which also eliminates any incorrect self-shadowing on that object.

[2]Woo shadows are named after Andrew Woo, who wrote the *Graphics Gems III* article describing the technique.

Woo shadows can be generated by asking Gelato to output the "avgz" (rather than "z"):

```
Output ("shadow.sm", "shadow", "z", "camera",
        "string filter", "min", "float[2] filterwidth", (1,1))
```

The only other difference between generating Woo versus ordinary shadows is that Woo shadows require shadow *receivers* to be in the shadow map, or the averaging math will not work out right. Thus, for Woo shadows, you may not eliminate objects such as the floor on the basis that it will not cast shadows on other objects (since it almost certainly will have a shadow cast upon it).

Woo shadow maps take about twice as long to generate than ordinary shadow maps. This is because both the closest and second-closest surfaces must be processed, and also because receivers must be included.

Once a Woo shadow map is generated, its use inside a light shader is unchanged — the light shader does not need to be modified, or even aware that it is using Woo shadows. Shadow lookup speed should be identical for Woo and ordinary shadows.

The tradeoffs involved in Woo versus ordinary shadow maps are summarized in the following table:

	Ordinary shadow maps	**Woo shadow maps**
Generate	`Output ("shadow.sm", "shadow",` **"z"**`, ...)`	`Output ("shadow.sm", "shadow",` **"avgz"**`, ...)`
Stores...	Depth of the first surface	Average depth of first and second surfaces
Included Objects	Shadow casters	Shadow casters *and receivers*
Bias	Very sensitive to bias	Fairly insensitive to bias
Generation speed	Pretty fast	About half the speed of ordinary shadow maps
Shader use	Using the `shadow()` function	Identical to ordinary shadow maps

10.1.3 Volume Shadow Maps

Ordinary shadow maps store a single z value per pixel that stores the depth of the closest surface for which accumulated opacity is above the opacity threshold. Woo shadows are identical, except that they store the average of the first and second surface above the threshold, thus reducing some undesired self-shadowing. These techniques are adequate for representing shadows of opaque objects with well-defined surfaces, but cannot represent shadows cast by objects that are partially transparent, many layers of transparent surfaces, volumetric phenomena, motion-blurred objects, or scenes with lots of small or thin objects (such as hair or fur).

Volume shadow maps are a solution to these problems. When shadowing hair, smoke, or other volumetric, wispy, or layered objects, you can (1) create a shadow map with a "volz" rather than a simple "z" output; (2) turn it into a MIP-mapped volume using `maketx -volshad`; and (3) use GSL `shadow()` as usual in your shaders to perform volumetric shadow lookups.

Volume shadow maps address the shortcomings of ordinary shadow maps by storing not a single depth value per pixel, but rather a series of depth values representing the depths at

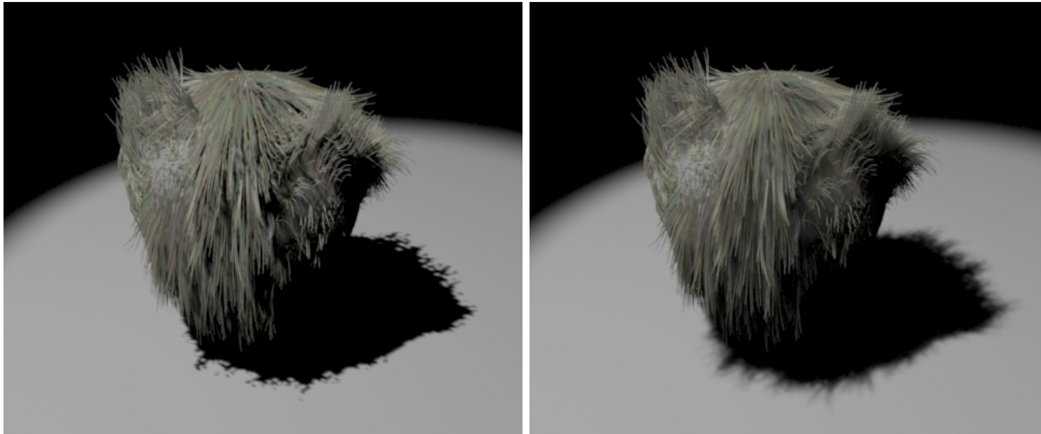

Figure 10.5: Hair shadowed with an ordinary shadow depth map (left) and with a volume shadow map (right).

which the accumulated opacity crosses various thresholds. Specifically, a volume shadow map contains the following data for each pixel:

- z_{opaque}, the depth at which opacity is 1, or fully opaque (or 1e30 if the pixel never reaches full opacity).
- α_{trans}, the total alpha of the transparent portion (not counting the opaque surface, if present), i.e., the accumulated opacity at z_1.
- z_0 the farthest depth at which the volume's accumulated opacity is still 0, i.e., the start of the volumetric medium.
- z_{a_i} the depth at which the accumulated opacity is $a_i \cdot \alpha_{\text{trans}}$, where n is the total number of transparent z channels stored (given by the `"zchannels"` parameter), i takes on values from $1 \ldots n-2$, and $a_i = i/(n-1)$.
- z_1 the depth at which the accumulated opacity is α_{trans}, i.e., the end of the volumetric medium prior to z_{opaque}.

Volume shadow maps take longer to generate than ordinary or Woo shadow maps of the same resolution and use much more disk storage for the maps themselves. But they can capture shadows of transparent as well as opaque objects, including many layers of transparency. They are ideal for hair and fur.

Creating a volume z file

A volume z file may be created by rendering the scene as you would to create an ordinary shadow map, but specifying the `Output` data name as `"volz"`:

```
Output ("shadow.vz", "tiff", "volz", "camera",
        "int zchannels", 3, "float opaquewidth", 0.05)
```

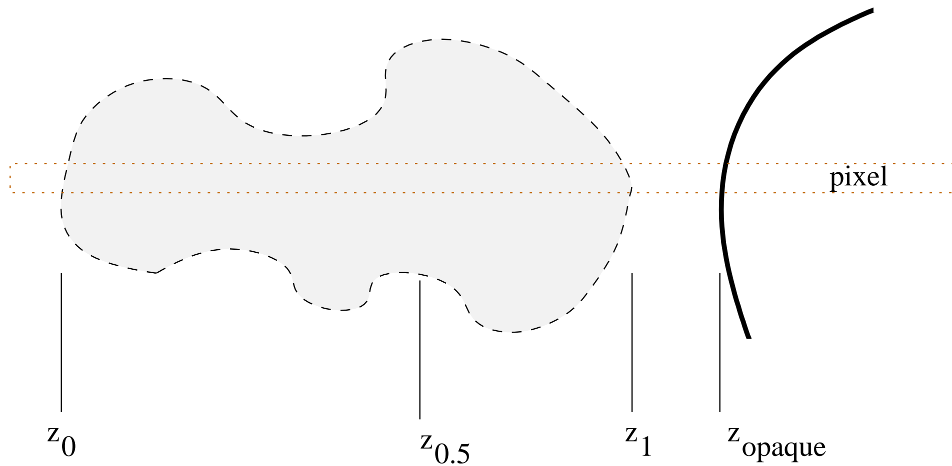

Figure 10.6: Volume shadow abstraction, for a case where $n = 3$, so $a_i = \{0, 0.5, 1\}$.

The optional `"zchannels"` parameter tells how many z_{a_i} values to store in the file. The default is 3, meaning that z_0, $z_{0.5}$, and z_1 will be created. That is probably sufficient most of the time, but complex volumetric density functions can be more accurately represented by increasing the number of z channels (with a linear increase in generation and storage expense). For very simple volumes, only 2 z channels may be sufficient, and are faster than three.

Multiple samples per pixel (i.e., `"spatialquality"`) may be used, and in fact are strongly encouraged, when creating `"volz"` outputs. However, when outputting `"volz"`, the optional `Output` parameters `"filter"` and `"filterwidth"` are ignored — the multiple samples within a pixel are combined in a particular way that is different than the way that ordinary pixel colors are filtered.

The optional `"opaquewidth"` parameter gives a hint about the depth range over which which the opaque z's within a pixel are considered "the same object" when multiple samples per pixel are combined into a single output pixel for the volume depth map. The default value is 0.05. This is somewhat similar to "bias" for ordinary shadow maps — if the `"opaquewidth"` is too small, slanted opaque objects may appear to incorrectly self-shadow; but if it is too large, opaque objects that are too close (in depth) may fail to correctly shadow each other.

Creating a mip-mapped volume shadow

After creating the initial volume z file, it should be turned into a full-fledged *MIP-mapped volume shadow map* using `maketx -volshad`:

```
maketx -volshad shadow.vz -opaquewidth 0.05 -o shadow.vsm
```

The optional `-opaquewidth` argument, which defaults to 0.05 if not specified on the command line, specifies the z range over which close values of z_{opaque} within adjacent texels are considered "the same" when combined into a single texel at a lower-res level of the MIP-map. This is exactly analogous to the `Output` option `"opaquewidth"` that applies to combining multiple samples within a pixel when rendering out a single-level `"volz"` file.

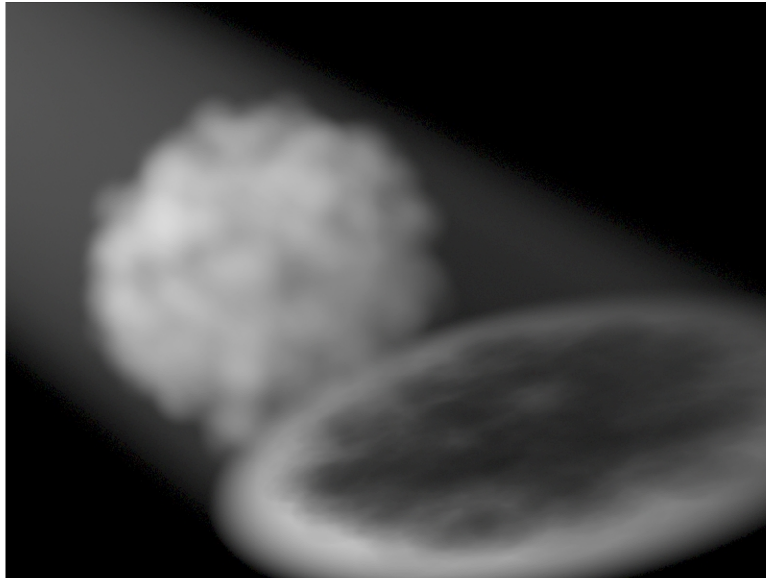

Figure 10.7: Example of an image that can easily be created with volume shadow maps, but not with ordinary shadow maps.

Using a volume shadow

You don't have to do anything special to use a volume shadow. Just use it in place of an ordinary shadow map — the GSL `shadow()` function notices that a map is a volume shadow and does the right thing automatically.

10.1.4 Cube-face shadow maps

Both ordinary and Woo shadow maps are created by rendering the scene from the point of view of the light, but this obviously only incorporates a *single* view. This is sufficient for a spotlight, but for a point- or omni-light that illuminates all directions, all of the shadows cannot be completely captured by any single shadow view.

The solution is a *cube-face shadow map*: six shadow map views are combined into a single file that may be accessed by a single `shadow()` lookup in a shader (see Figure 10.8). Cube-face shadow maps are easy to set up:

- Cube-face shadows may be either ordinary or Woo shadows.
- Each of the six faces of the shadow map is generated in the usual way.
- Each of the six faces should be a perspective view with exactly a 90° field of view.
- The six faces are combined using the `maketx` utility:
  ```
  maketx -shadcube [options] px nx py ny pz nz -o shadowfile
  ```
  where *px, nx, py, ny, pz, nz* are the names of the individual shadow files for the six cube faces. See Section 9.1.4 for more information on `maketx` and its options.

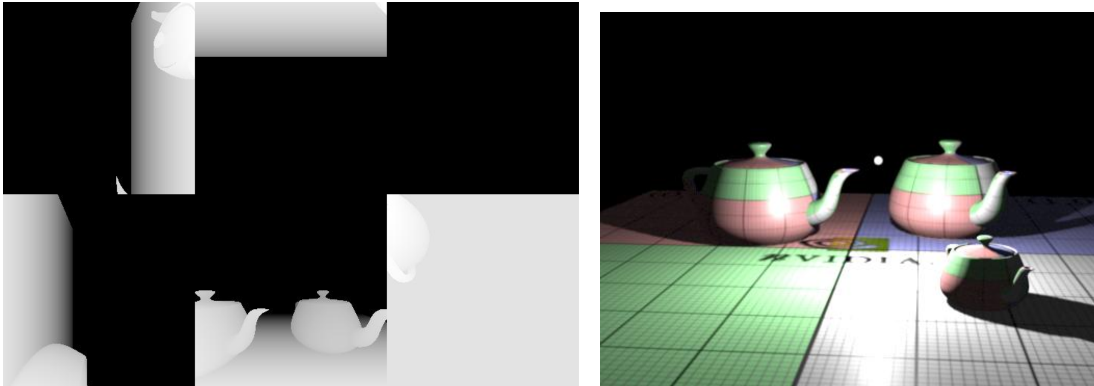

Figure 10.8: Cube-face shadow map and its application.

The table in Section 9.3 lists the directions for each face of a cube-face shadow map.

When it comes to accessing shadows in a light shader, there is no difference between a cube-face or single-view shadow map. For a cube-face shadow lookup, a single call to `shadow()` will automatically choose the right face, including properly "wrapping" around the edges and corners of the faces.

10.1.5 Dynamic shadow maps

Beginning with Gelato 2.0, it became possible to generate and use shadow maps *dynamically* — that is, the renderer can generate pieces of a shadow map as they are needed by the beauty pass, rather than require that a shadow map be previously generated by an entirely separate pass. The shadow map is treated in the usual way, except that instead of reading shadow tiles from a disk file, the tiles are generated by the renderer itself as they are needed.

Dynamic shadow maps have several advantages: time may be saved because only the parts of the shadow map that are actually needed are rendered, no separate passes are required to generate the shadows (including the cost of reading the scene file separately for each shadow map pass), and no time is spent writing or reading the shadow map to or from from disk (or over a network). Also, whereas a separate shadow map pass must finish before a beauty pass may begin, dynamic shadow maps may greatly decrease time between the beginning of a render and when you start to see the first completed pixels, which can be especially critical for re-rendering (see Chapter 11).

On the other hand, dynamic shadow maps are not without their disadvantages. Dynamic shadow maps are recomputed on every frame, whereas actual disk-based shadow map files may be reused for rendering many frames (assuming the shadow-casting objects therein do not move from frame to frame). Moreover, even during a single frame render, a particular piece of the dynamic shadow map may be recomputed multiple times, depending on its pattern of use and how much texture cache space is available. Finally, like other uses of multiple cameras in Gelato, the dynamic shadows will share a single tessellation with the beauty pass. Nonetheless, we think that dynamic shadow maps are extremely convenient and efficient for shadow maps containing objects that move on every frame (and therefore could not have been shared among frames anyway).

To use dynamic shadow maps:

1. Make sure that the beauty pass is the first `Camera` declared (so that it is the one that controls shading and tessellation).

2. Set up a `Camera` corresponding to the shadow map's camera.

3. The shadow map's `Output` statement must have
   `Parameter ("int dynamic", 1)`
   to indicate that it is a dynamic shadow, and should not render a separate file to disk.

4. Any objects to appear in the shadow map are in the geometry set corresponding to the shadow camera (i.e., the geom set has the same name as the camera).

5. Shaders reference the shadow map in the usual manner — by its file name as specified in the shadow map's `Output` statement — even though the file will not actually be written to disk.

Currently, dynamic shadow maps may be used for ordinary and Woo shadows. As of release 2.0, cube-map shadows and volume shadows may not be computed dynamically.

10.1.6 Ray Traced Shadows

Gelato also supports ray-traced shadows: instead of using a shadow map, the path joining the light source to the point being shaded is tested for occlusion against other objects in the scene.

Gelato extends the `shadow()` call to support ray tracing. If the `shadowname` parameter is the name of a geometry set, ray tracing will be used instead of looking up from a shadow depth map file. Thus, there is no difference in the light shader, whether shadow maps or ray-traced shadows are used. In fact, the exact same light shader may be used for maps on one light, and ray tracing on antoher light in the same scene. However, the optional arguments to `shadow()` have slight differences depending on whether maps or ray tracing is used:

- The `"blur"` parameter causes a penumbra effect (in fact, a much more realistic appearance than blurred shadow maps), but is measured differently. For shadow maps, the `"blur"` amount is interpreted as a fraction of the resolution of the shadow map camera, whereas when ray tracing, `"blur"` is interpreted as the apparent angular size of the light source (1.0 being a 90 degree light source). In both cases, 0 means perfectly sharp and larger values quickly become blurrier. But you should not expect the same blur value to have an identical appearance when using ray tracing versus when using shadow maps.

- The `"samples"` parameter controls the number of ray samples that are used to antialias the shadow edge and to give the appearance of penumbra (for nonzero `"blur"`). Note that to an even greater degree than with shadow maps, ray traced shadows become much more expensive as `"samples"` increases.

- The `"bias"` parameter is also honored (and necessary) for ray-traced shadows, although the amount of bias required to eliminate self-shadowing artifacts may be different for ray-traced shadows than for shadow maps.

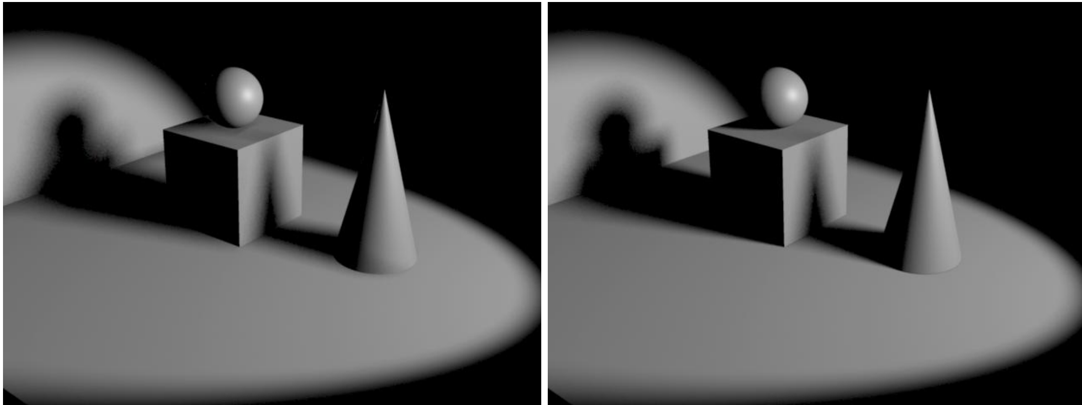

Figure 10.9: Blurry shadows with depth maps (left) have a uniformly blurry penumbra, wheras ray traced shadows with blur (right) have a more realistic penumbra, sharp where near the occluder and blurry where far from the occluder.

Another thing to keep in mind when using ray-traced shadows is that objects will only cast ray-traced shadows if they are visible to the named geometry set. Remember that the attribute to add subsequent primitives to a geometry set named "shadow" is:

```
Attribute ("string geometryset", "+shadow")
```

10.2 Reflections

Many surfaces are polished to a sufficient degree that one can see coherent reflections of the surrounding environment. The presence and appearance of reflections is controlled by the surface shader. Gelato provides for three different methods for making reflections in surfaces: environment mapping, flat reflection mapping, and ray tracing.

Environment mapping works best if the reflective object is curved, and is really the only applicable technique if the environment is painted or is captured from a real scene.

Flat reflection mapping is generally superior to environment mapping if the reflective object is flat (like a floor or a large flat mirror), but is tricky to do properly if there are objects behind the mirror, or if the reflective object is not almost perfectly flat.

Ray tracing is much slower and more memory-intensive than environment or reflection mapping, but it works for both curved and flat objects, is geometrically accurate, and may be the only method that looks right for tricky situations such as mutually-reflective objects.

10.2.1 Environment Maps

Environment mapping takes a pre-rendered, captured, or painted image of the reflective environment and looks up texture indexed by a direction vector, thus simulating reflection. The environment map is either rectangular (called a latitude-longitude environment map) or composed of six axis-aligned directions from a particular point (called a cube-face environment map).

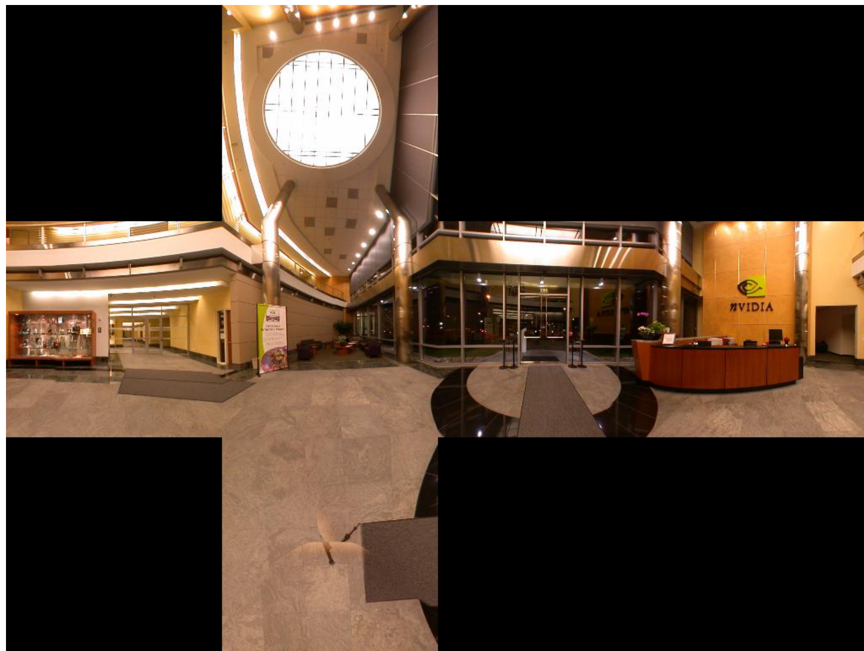

Figure 10.10: An example environment map displayed in the “cross” configuration.

Environment maps may be completely synthetic, from six rendered images, or a painting.

Environment maps may also be captured from a real environment, as six 90° photos, two fisheye lenses, a single spherical "light probe" map, or any other means of "stitching" multiple views together to achieve full directional coverage.

See Section 9.1 for details on converting various formats into Gelato environment maps. An example of an "unwrapped" environment map, captured from six images of a real scene, is shown in Figure 10.10. Its use on a reflective object is shown in Figure 10.11. Most modeling systems geared toward making input for Gelato will have an automatic facility for generating cube face environment maps of your synthetic scene for any object that is reflective.

Accessing an environment map from inside your shader is straightforward with the built-in `environment()` function:

```
type environment (string envname, vector R, ...)
```

For environment maps, the coordinates coordinates consist of a direction vector. In order to properly filter the environment map lookup, based not merely on the direction but also on the solid angle of reflection being sampled on each surface element, derivatives are automatically taken of the reflection vector. Optionally, explicit derivative vectors may passed to `environment()`, in which case no derivatives will be taken.

Environment maps usually sample in the direction of mirror-like reflections. The reflection direction may be easily computed in the shader based on the incident ray direction and object normal:

```
normal Nf = normalize (faceforward (N, I));
vector R = reflect (normalize(I), Nf);
```

The return type can be explicitly cast to either `float` or `color`. If you do not explicitly cast the results, the compiler will try to infer the return type, which could lead to ambiguous situations.

The `environment()` function takes many optional arguments, passed as pairs of string tokens and values. The most commonly-used optional arguments are:

- `"blur"`, `float`: controls how sharp or blurry the reflections are. A value of 0 (the default) is perfectly sharp — a polished mirror, for example. Increasing values of `"blur"` indicate progressively blurrier surfaces — better for dull metal or gloss. A value of 1 indicates a 90° blur angle. Figure 10.11 shows a succession of blur values. Real objects are rarely perfectly sharp, so we advise always using some blur, even if a very small amount.

- `"space"`, `string`: Environment maps must be indexed by a vector in the same coordinate system that they were created in (frequently `"world"` space). If you index the environment map in the wrong space (particularly `"common"` space), you could get very strange results with your reflections. You could `transformv` the vector before passing it into `environment()`, but it is much easier to use the optional `"space"` parameter, which specifies the name of the coordinate system giving the position and orientation of the environment map. Obviously, the name of the coordinate system must be one that the renderer knows about: either a built-in coordinate system (such as `"world"`), or else a coordinate system that has been named with `SaveAttributes`.

By using a named coordinate system that can be manipulated independently of the object itself, it's easy to manipulate the reflections in various ways (such as re-orienting them relative to the object). An example of this is shown in Figure 10.12.

- `"radius"`, `float`: By default, the environment map is indexed by a direction, as if the environment map is a textured sphere with an infinite radius. This can cause two problems: (1) as objects move, there is no *parallax*, in particular the reflections will become less geometrically accurate the farther the reflective object is from the position where the environment map views were rendered or captured; and (2) flat objects look wrong because all the reflection rays point the same direction, and therefore look up the same part of the environment map. This can be largely alleviated by passing a `"radius"` parameter, which will compute the parallax properly assuming that the environment is a finitely-sized sphere with the given radius, centered at the origin of the coordinate system specified by the optional `"space"` parameter. In this case, the surface position `P`, as well as the reflection direction passed to `environment()`, is transformed into the named coordinate system.

  Thus, for a closed environment like a room, the argument for `"radius"` should be the approximate size of the room.

Figure 10.11: Increasing the `"blur"` parameter to `environment()`: 0, 0.075, 0.25.

Figure 10.12: Changing the coordinate system with the `"space"` parameter can change the orientation of the reflections relative to the object.

Please refer to Section 5.8.10 for a full specification of the `environment()` function and all its options. Listing 10.2 shows a typical use of `environment()` to make a shiny metal object.

Table **??** lists the directions for each face of a cube-face environment map, which are identical to that of a cube-face shadow map.

Listing 10.2 A simple surface shader that makes a mirror-like surface with `environment()`.

```
surface shiny ( float Ka = 1,
                float Kd = 0.1,
                float Ks = 0.8,
                float roughness = 0.2,
                float Kr = 0.5,
                float blur = 0,
                string envname = "",
                string envspace = "world",
                float envrad = 0,
                float samples = 1 )
{
    normal Nf = faceforward (normalize(N), I);
    vector IN = normalize(I);

    color Cbase = C;
    C = Ka*ambient() + Kd*diffuse(Nf) + Ks*specular(Nf,-IN,roughness);

    if (envname) {
        vector R = reflect (IN, Nf);
        C += Kr * (color) environment (envname, R, "blur", blur,
                                       "samples", samples,
                                       "space", envspace, "radius", envrad);
    }

    C *= Cbase;  // Metals filter all reflected light by base color
    C *= opacity;
}
```

10.2.2 Ray traced reflections and refractions

There are times when environment or reflection mapping is not adequate. Reflection maps are really only applicable if the reflector is perfectly flat. Environment mapping doesn't work well for flat or nearly flat objects, and is incorrect for objects that touch (or nearly touch) the reflective object. There can be parallax problems. Neither technique works well for scenes in which there are mutually-reflective objects, or in which an object must reflect another part of itself. Also, it is more difficult to use environment maps to convincingly model refraction.

Gelato also supports ray-traced reflections and refractions by extending the `environment()` routine. If the environment name supplied to the `environment()` is the name of a geometry set rather than the name of a file, then the `environment()` function will ray trace against the primitives in the named geometry set instead of looking up the result from an environment map file. In other words, the `shiny.sl` shader in Listing 10.2 can be used for ray tracing *without modification*. We only need to pass the name of a geometry set rather than the name of an environment map file. In other words, specifying this shader in the scene file as:

```
Shader ("surface", "shiny", "string envname", "room.env",
        "string envspace", "world")
```

will get reflections from the environment map "`room.env`", whereas

```
Shader ("surface", "shiny", "string envname", "reflection")
```

would use ray-traced reflections, assuming that primitives had been put in the "`reflection`" geometry set using the following Attribute:

```
Attribute ("string geometryset", "+reflection")
```

Most of the optional environment map parameters are implemented analogously when performing ray tracing. The most commonly-used optional parameters to `environment()` that are used for ray tracing are:

- The `"blur"` parameter works for ray tracing analogously to its operation for environment maps. The only difference is that when ray tracing, large blurs will look “noisy” if the number of samples is not increased (it is not necessary to specify samples for map lookups).
- The `"samples"` parameter is used as with ray-traced shadow map lookups: it specifies the number of rays to sample and average to determine the reflection color. More samples makes ray-traced reflections more expensive (roughly proportional to the number of samples), but reflections may alias or look noisy with inadequate samples, most noticeably when `"blur"` is also used.
- The `"bias"` parameter is often necessary to prevent incorrect self-intersection of rays against the surface being shaded, much like it is necessary for shadow maps and ray-traced shadows. If not specified in the `environment()` call itself, or if the value passed is negative, the bias value will be taken from the global `"shadow:bias"` attribute.

There are many other, more advanced, ray-tracing parameters for the `environment()` function that are less commonly used. Please refer to Section 5.8.10 for a full specification of the `environment()` function and all its options.

10.3 Indirect Illumination

Direct illumination refers to light leaving a light source, traveling in a straight line, and arriving (possibly shadowed) at an object seen by the camera. In the real world, much of the light arriving at objects did not come via a straight line from the light. Rather, light will bounce between objects in the scene. This *indirect illumination* can be computed by Gelato, albeit at an additional expense. Figure 10.13 shows the contribution of indirect illumination to an example scene.

Figure 10.13: Left: direct light only. Middle: direct and indirect. Right: indirect illumination only. Model courtesy of Headus, Inc.

Gelato supports indirect illumination using a Monte Carlo technique. Rather than enmeshing the scene and solving the light transport up front (as finite element-based radiosity techniques would do), the Monte Carlo approach is "pay as you go." As it's rendering, when it needs information about indirect illumination, it will do a bunch of extra ray tracing to figure out the irradiance. It will save those irradiance values, and try to reuse them for nearby points.

To use the Monte Carlo irradiance calculations for global illumination, you need to follow the following steps:

1. **Turn on indirect illumination for the scene**

   Add a light source to the scene using the `"indirectlight"` light shader. The shader has a string parameter `"indirectname"` which names a geometry set containing all the objects that are "visible" to indirect illumination sampling. In our examples below, we will name this geometry set `"indirect"`, but you can choose any name you want, including the name of the geometry set you use for other ray-traced reflections or shadows (but you cannot use the name of a camera).

   ```
   Light ("lightname", "indirectlight",
          "string indirectname", "indirect", ...)
   ```

2. **Reflected objects**

   Indirect illumination is, in many ways, just a very blurry ray-traced reflection. As such, only objects that will be sampled by indirect rays are those in the named geometry set passed as the `"indirectname"` parameter of the `indirectlight`. In our example, all objects that can "be seen" by the indirect illumination rays must be tagged as being visible in the `"indirect"` geometry set:

```
Attribute ("string geometryset", "+indirect")
```

To denote an object that does not interact with indirect rays, just remove it from the geometry set:

```
Attribute ("string geometryset", "-indirect")
```

3. **Receivers**

   All objects that are illuminated by the "indirectlight" light will receive indirect illumination. If there are any objects that you specifically do not want indirect illumination to fall on, you can just use LightSwitch to turn the light off for those objects.

4. **Adjust Parameters**

   It's expensive to recompute the indirect illumination at every pixel, so it's only done sparsely, with the results interpolated or extrapolated. There are several time/quality controls adjusting how often this recomputation is done. The parameters roughly break down into those controlling how frequently to do a full recomputation of indirect illumination, and those controlling how much work to do each time a full recomputation is necessary.

   The indirectlight shader takes the following additional controls (explained more fully and formally in Section 5.8.11):

   - "float samples": controls the maximum number of rays used to sample indirect illumination, any time that a full sampling is needed. The default is 64. Try low values such as 16 for fast, rough previews, and higher values such as 256 or higher for full-quality final frames.
   - "float adaptive": when nonzero, attempts to use fewer than the maximum samples when the variance of the samples appears to be low. The default is zero. If your indirect renders are taking too long, try setting it to 1 and see if that speeds things up without diminishing the quality (it'll work well on some scenes, poorly on others).
   - "float maxerror": A maximum error metric. Smaller numbers cause recomputation to happen more often. Larger numbers render faster, but you will see artifacts in the form of obvious "splotches" in the neighborhood of each sample. Values between 0.1–0.5 work reasonably well, but you should experiment. But in any case, this is a fairly straightforward time/quality knob. The default is -1, which indicates that the shader should use the Attribute "indirect:maxerror", which defaults to 0.25.
   - "float maxpixeldist": gives a maximum distance, in pixels, for which a computed value will be used for interpolation. Higher values produce rougher, faster renders. The default is -1, which indicates that the shader should use the Attribute "indirect:maxpixeldist", which defaults to 20.

   We recommend adjusting these parameters to the indirectlight and using them uniformly across the scene. However, advanced users may wish to adjust them on an object-by-object basis. If that is desired, please note that it is possible to set them via certain Attributes (see Section 4.4.10) and tell the shader to use the attributes (see the comments in "indirectlight.gsl" for details).

It is also possible to bypass the caching and interpolation entirely, by setting both `"maxerror"` and `"maxpixeldist"` to 0, which forces a full resampling to be done for every pixel. In this mode, you will probably also be able to use a much lower `"samples"` value.

10.4 Ambient Occlusion

Ambient occlusion is a technique used to simulate global effects without doing a full global illumination simulation. This is especially helpful when no global simulation can be performed — for example, on a CG character that will be composited into a live-action background.

The basic technique of ambient occlusion involves calculating, for every point in the scene, how much of the hemisphere above each point is unoccluded by local geometry. Typically, this forms an image that is white for points that are unoccluded, black for points whose surrounding hemisphere is completely occluded by other geometry (see Figure 10.14).

Figure 10.14: Example of an ambient occlusion image.

This information may be used directly, but more commonly it is computed in a separate rendering pass (viewed from the same camera as the beauty pass), and saved in an image, which is turned into a texture map. During the beauty pass, `P` is projected into `"NDC"` space to yield coordinates used to look up from the ambient occlusion texture. This value — a gray-scale representing how relatively unoccluded each point is — may be used to scale an ambient, environmental, or fill light.

Why is this helpful? If you light a scene with just key lights (as in Figure 10.13, left), the lighting can be too harsh, with large black areas anyplace in shadow or oriented away from the key light. The traditional solution is to add an *ambient* light of constant intensity to the scene to prevent these areas from being completely dark, but this often looks unrealistic. A superior solution is to scale the contribution of the ambient light by the ambient occlusion image, which will yield a much more realistic result (as in Figure 10.15).

Ambient occlusion is easy to incorporate into your rendering pipeline, and is summarized below.

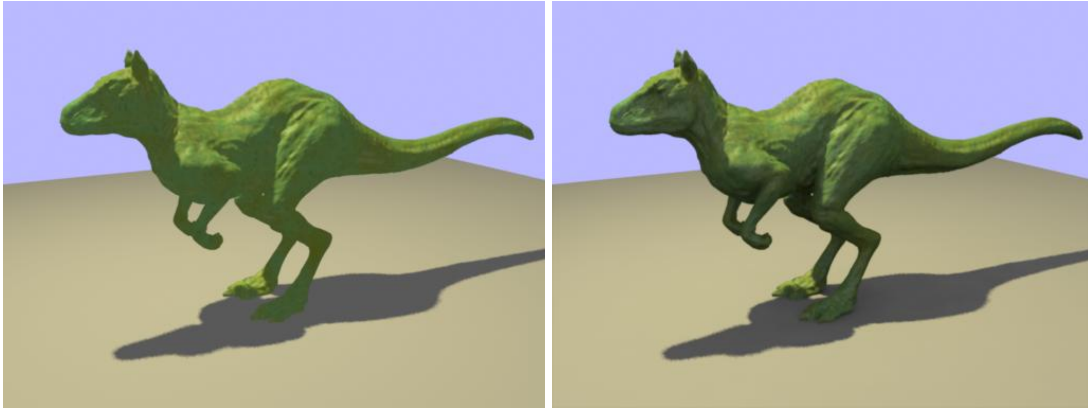

Figure 10.15: Traditional ambient light looks flat (left); ambient light scaled by the ambient occlusion image looks much more realistic (right).

Pass 1: Computing ambient occlusion

The ambient occlusion pass is typically rendered from the same point of view as the beauty pass. The resulting image should be saved separately, for example:

```
Output ("ambocc.tif", "tiff", "rgb", "main")
```

Although displacement shaders should remain, surface and volume shaders should be removed from all objects, and instead Gelato's `ambocclude` shader should be used:

```
Shader ("surface", "ambocclude", ...)
```

The `ambocclude` shader takes a number of parameters that control time/quality tradeoffs, in particular how often ambient occlusion is fully sampled and how much work is done for each full sampling:

- `"float samples"`: controls the maximum number of rays used to sample occlusion, any time that a full sampling is needed. The default is 64. Try low values such as 16 for fast, rough previews, and higher values such as 256 or higher for full-quality final frames.
- `"float adaptive"`: when nonzero, attempts to use fewer than the maximum samples when the variance of the samples appears to be low. The default is 1. If your occlusion renders are too noisy, try setting it to 0 and see if that improves the quality. Adaptive sampling will work well on some scenes, poorly on others.
- `"float maxerror"`: A maximum error metric. Smaller numbers cause full occlusion sampling to happen more often. Larger numbers render faster, but you will see artifacts in the form of obvious "splotches" in the neighborhood of each sample. Values between 0.1–0.5 work reasonably well, but you should experiment. But in any case, this is a fairly straightforward time/quality knob. The default is -1, which indicates that the shader should use the Attribute `"occlusion:maxerror"`, which defaults to 0.25.

- `"float maxpixeldist"`: gives a maximum distance, in pixels, for which a computed value will be used for interpolation. Higher values produce rougher, faster renders. The default is -1, which indicates that the shader should use the Attribute `"occlusion:maxpixeldist"`, which defaults to 20.

We recommend adjusting these parameters to the `ambocclude` and using them uniformly across the scene. However, advanced users may wish to adjust them on an object-by-object basis. If that is desired, please note that it is possible to set them via certain Attributes (see Section 4.4.11) and tell the shader to use the attributes (see the comments in `ambocclude.gsl` for details).

There are also several parameters that control the appearance of the ambient occlusion image:

- `"float bias"` works as with other ray-tracing functionality, ignoring hits closer than this distance in an effort to reduce incorrect self-occlusion of the surface.
- `"float maxhitdist"` gives a maximum distance to search for ray hits. Occluders farther than this distance are ignored. This is also helpful in speeding up ambient occlusion computations, by reducing the geometric region that needs to be searched for ray-object intersections.
- `"float falloff"` and `"float falloffmode"` control how distance of occluding objects affects the amount of occlusion. If $falloff = 0$ (the default), all ray hits are "equally occluding." If $falloff > 0$ and $falloffmode = 0$, the amount of occlusion will be $e^{-falloff/dist}$. If $falloff > 0$ and $falloffmode = 1$, the amount of occlusion will be $(1 - dist/maxhitdist)^{falloff}$. The default for both parameters is 0, indicating that the effects of occlusion are not subject to distance falloff.

  The most important use of falloff is when you are using ambient occlusion in a closed environment.

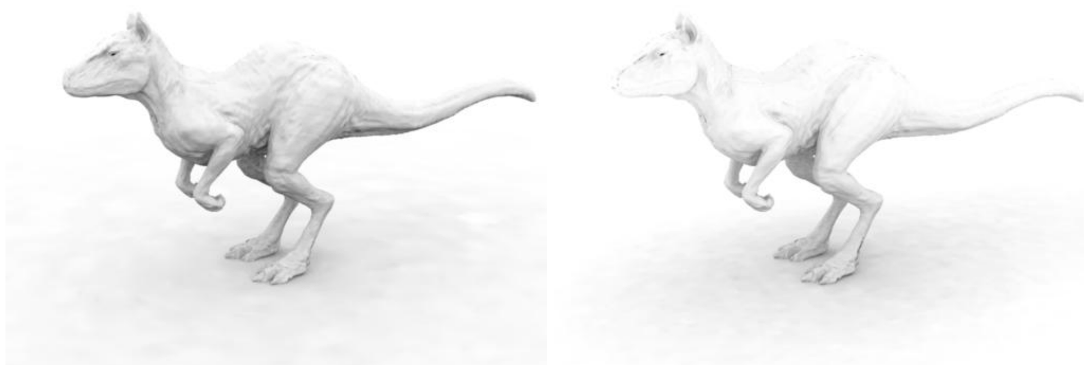

Figure 10.16: Ambient occlusion with no falloff (left), and with falloff (right).

- `"float twosided"`: when zero (the default), ambient occlusion will only be computed at points where the normal is facing toward the camera, in order to save time by skipping

the expensive computation for points that are facing away from the camera. This trick only works for objects whose normals consistently face outward. If you cannot guarantee well-behaved normals, you should set `"twosided"` to 1.

The source code of `ambocclude.gsl` is shown in Listing 10.3.

Listing 10.3 `ambocclude.gsl`: Computing ambient occlusion.

```
surface
ambocclude ( string occlusionname = "",
             string sdbname = "",
             string sdbmode = "",
             float samples = 64,
             float adaptive = 1,
             float maxerror = -1,
             float maxpixeldist = -1,
             float bias = -1,
             float maxhitdist = 1.0e6,
             float falloff = 0,
             float falloffmode = 0,
             float twosided = 0,
             output float unocc = 1,
             output normal Nunocc = 0 )
{
    if (occlusionname) {
        normal Nf = normalize(N);
        if (twosided != 0)
            Nf = faceforward (Nf, I);
        unocc = 1 - occlusion (occlusionname, P, Nf, M_PI_2, Nunocc,
                               "samples", samples, "maxerror", maxerror,
                               "maxpixeldist", maxpixeldist,
                               "maxhitdist", maxhitdist, "bias", bias,
                               "falloff", falloff, "falloffmode", falloffmode,
                               "adaptive", adaptive,
                               "filename", sdbname, "filemode", sdbmode);
        C = unocc;
    } else {
        C = 1;
    }
    opacity = 1;
}
```

Beauty pass: Using ambient occlusion

First, you will need to convert the image created by the first pass into a texture, using `maketx`:

```
maketx ambocc.tif ambocc.tx
```

The most straightforward use of the ambient occlusion information is to modulate the intensity of a “fill light,” in place of a uniform-intensity ambient light, which is visually unsophisticated. An example of such a light is Gelato’s sample `envlight.gsl`. The `envlight.gsl` shader is shown in Listing 10.4. Beyond the basic operation of scaling uniform ambient illumination by an ambient occlusion map, the `envlight` shader also allows on-the-fly computation of the ambient occlusion factor, as well as scaling the light color by an environment map

Listing 10.4 `envlight.gsl`: Environment lighting that uses ambient occlusion.

```
light envlight ( color lightcolor = 1,
                 float intensity = 1,
                 string occlusionname = "",
                 string sdbname = "",
                 string sdbmode = "",
                 string envname = "",
                 string envspace = "world",
                 float envrad = 0,
                 float blur = 0.5,
                 float maxhitdist = 1.0e6,
                 float bias = -1,
                 float samples = 16,
                 float adaptive = 1,
                 float maxerror = -1,
                 float maxpixeldist = -1,
                 float falloff = 0,
                 float falloffmode = 0,
                 string occlusionmap = "",
                 float Kocclusion = 1,
                 string normalmap = "",
                 output float __nonspecular = 0 )
{
    vector Nunocc;
    float a;

    if (occlusionmap != "") {
        // An occlusion map was supplied - look up using "NDC" coords
        point PNDC = transform ("NDC", P);
        float x = PNDC[0], y = PNDC[1];
        a = texture (occlusionmap, x, y);
        if (normalmap)
            Nunocc = (normal) texture (normalmap, x, y);
        else Nunocc = Ns;
    } else if (occlusionname != "") {
        // No occlusion map given, but a geometryset was passed --
        // compute occlusion
        a = 1 - occlusion (occlusionname, Ps, Ns, M_PI_2, Nunocc,
                           "samples", samples, "maxerror", maxerror,
                           "maxpixeldist", maxpixeldist,
                           "maxhitdist", maxhitdist, "bias", bias,
                           "falloff", falloff, "falloffmode", falloffmode,
                           "adaptive", adaptive,
                           "filename", sdbname, "filemode", sdbmode);
        Nunocc = Ns;
    } else {
        // Neither an occlusion map nor an occlusion geomset was supplied.
        a = 1;
        Nunocc = Ns;
    }
    a = mix (1, a, Kocclusion);

    emit (-Nunocc, 0) {
        Cl = lightcolor * intensity * a;
        if (envname)
            Cl *= (color) environment (envname, Nunocc, "space", envspace,
                                       "radius", envrad, "blur", blur);
    }
}
```

10.5 HDRI and Image-Based Lighting

It is often useful to light objects using an environment map as the light source. This kind of technique is particularly handy for inserting synthetic objects into real scenes or lighting environments. To have a remotely accurate representation of a scene's lighting, 8-bit integer values of 0-255 will not do. Instead, to achieve a reasonable quality level, an *HDR* (high dynamic range) representation is needed, which really means that you need some kind of floating point environment map.

Once you have an HDR environment map representing the lighting in your scene, there are several techniques that Gelato supports to light scenes using that map. The easiest two techniques are described below.

10.5.1 Technique 1: Environment light source

The easiest technique to make the environment lighting look like an actual light source in the scene is to use the `"envlight"` light source shader that comes with Gelato (see the example in Figure 10.17). This shader looks up the lighting from an environment environment map, optionally modulating the light using the ambient occlusion (either pre-calculated or computed on the fly). Furthermore, when using ambient occlusion, the direction of the environment map lookup is perturbed toward the average unoccluded direction. The steps for using this technique are straightforward:

- Add a light using the `"envlight"` shader, specifying the name of the environment map as the `"envname"` parameter. The orientation of the lighting may be changed by adjusting the coordinate system named by the `"envspace"` parameter.
- To calculate ambient occlusion on the fly, put all objects visible to occlusion rays in a named geometry set, and pass the name of the geometry set as the `"occlusionname"` parameter to `envlight`. Several other parameters adjust the occlusion calculation, please see the comments in the shader's source code (in `$GELATOHOME/shaders`).
- To use a previously-computed ambient occlusion image, pass the name of an ambient occlusion texture map as the `"occlusionmap"` parameter to `envlight`. If the occlusion pass also produced a "bent normal" pass, the normal image may be specified as the `"normalmap"` parameter and used to adjust the directional lookup into the environment map.

With this technique, it is not necessary to add an `"indirectlight"` (but is certainly allowed if you want to also account for light interreflected between objects).

10.5.2 Technique 2: Emissive sky sphere

If you are using indirect illumination anyway, another simple way to light a scene using an environment map is to simply place a large sphere around the whole scene, which has a surface shader that is emissive (glows, without needing external lights), and let the indirect illumination do its job. This involves the following steps:

Figure 10.17: Environment lighting using the map shown in Figure 10.10 as the only light in the scene.

1. Place a large sphere around the whole scene. Make sure that the camera's `"far"` clip plane is large enough not to clip the sphere.
2. The surface shader of the sphere should color itself using a blurred lookup from the environment map, without needing any other lights. The `"envsurf"` shader that comes with Gelato is ideal. For example:

```
Shader ("surface", "envsurf", "string envname", "hdroom.env",
        "float blur", 0.1)
```

3. Be sure to put the sphere in the geometry set used for indirect rays (passed as the `"indirectname"` parameter to the `indirectlight`). If you don't want the sphere (and the image on it) to be visible to the camera, make sure to make it invisible to the camera.
4. Make sure you are using indirect illumination by adding the `"indirectlight"` light source to the scene, making objects visible to reflections, and tuning the indirect parameters appropriately (see Section 10.3).

The main drawback to this technique is that you may require a large number of samples for your indirect illumination.

10.6 Caustics

Caustics are bright spots caused by the focusing of light that is reflected or refracted, particularly by curved objects (see Figure 10.18).

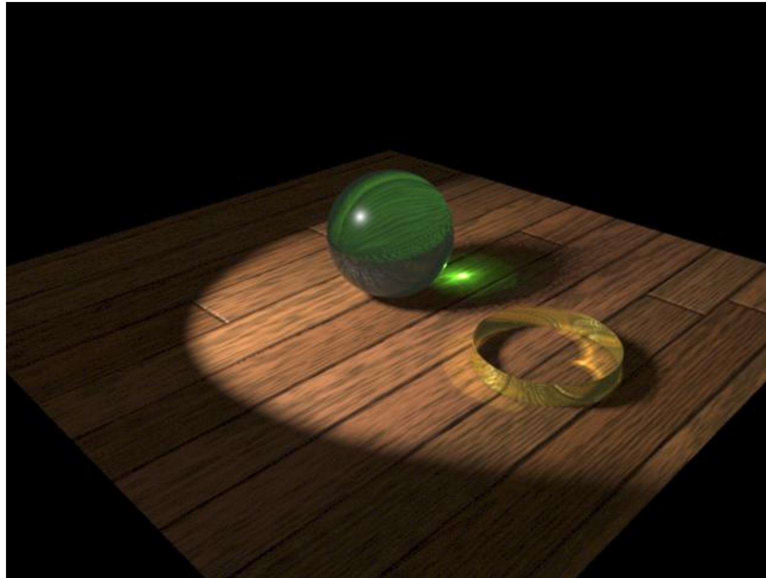

Figure 10.18: Example of refractive and reflective caustics.

Gelato can compute caustics by simulating the action of light with *photon mapping*. This is done with a multi-pass technique. The initial passes involve shooting photons from a light (one light source per pass). The final pass is an ordinary render from the main camera, but will use data from the initial passes to reconstruct the caustics.

10.6.1 Initial pass: creating the photon map

In the initial pass, we are concerned only with shooting the photons from a light, propagating the photons, and saving the resulting photon map. This is accomplished with the following steps:

1. Image the scene using a camera at the light position, similarly to how a shadow map is created. Use a perspective projection for spot- or point-like light sources, an orthographic projection for distant-like light sources with parallel rays.

2. Completely cover the camera view with a patch that has the `"shootphotons"` as its surface shader. The camera-covering object should not be visible to rays (i.e., it should only be in the camera's geometry set). The easiest way to do this is simply to input the the file `$GELATOHOME/inputs/frontplane.pyg`, which contains the Pyg commands to place a patch at the near clip plane that exactly covers the camera view and is invisible to rays:

```
PushAttributes()
Shader ("surface", "shootphotons", ...)
Input ("frontplane.pyg")
PopAttributes()
```

(Go ahead and look at the `frontplane.pyg` file to see how it works.)

3. The `"envname"` parameter to the `"shootphotons"` shader should be the name of a geometry set that contains all objects that may interact with the photons — reflective and refractive objects, objects that photons may land upon, and objects that may block photons from propagating.

4. The `"distribution"` parameter to the `"shootphotons"` controls the distribution of photons, which is corresponds to the type of light source. A value of `"spot"` (the default) shoots from the camera position, covering the view, and is appropriate for spot lights or point lights when it is known the photons only need to go in particular angles. A value of `"omni"` shoots in all directions from the camera position, as appropriate for point lights when photons are needed in all directions. A value of `"parallel"` shoots parallel rays covering the view, as appropriate for "distant" lights.

5. All other objects in the scene should have the `"movephotons"` shader attached to them. Whether an object is a photon receiver, reflector, or refractor is determined by the parameters to this shader. Thus, you may attach `"movephotons"` with different parameters to different objects, or attach overriding parameters to individual pieces of geometry. The meaning of the material parameters is discussed below, in Section 10.6.3.

   The `movephotons` shader will write the photons to a spatial database file named by its `"photonfile"` parameter, which by default is `"caustics.sdb"`. The name is arbitrary—you can pass any name, or even have different pieces of geometry save their photons to different files.

The `"movephotons"` shader will reflect/refract and continue to trace photons for refractive or refractive objects. For objects that are not reflective or refractive, they will write the photon to the photon map (but not the first objects hit by the — i.e., it doesn't record direct light).

The total number of photons is determined by the resolution of the camera-covering patch, since each point shaded on that patch will result in one photon being emitted. Thus, for more accurate photon tracing, just increase either the resolution of the first-pass image or increase the `"shadingquality"` of the camera-covering patch.

The actual image produced by the photon-shooting pass is never used again. But for debugging purposes, the `"shootphotons"` and `"movephotons"` shaders have been constructed so that the first-pass image shows a color-coded image of where the photons were shot: red hit a reflector, green hit a refractor, white hit a receiver, and black hit no object.

10.6.2 Beauty pass: using the photon map

Creating the "beauty pass" utilizing the photon map is straightforward and nearly identical to rendering a scene without any caustics at all.

1. Image the scene from the normal camera, using all the usual shaders on all objects, and without the camera-covering "shootphotons" object in the scene.

2. Add a light to the scene to gather the photons, using the "causticlight" shader. For example:

```
Light ("caustics", "causticlight", "string photonfile", "caustics.sdb")
```

   The "photonfile" parameter must be the name of the spatial database containing the caustic photons. Other parameters to "causticlight" that might be helpful to adjust are:

   "float intensity" Overall scale on the intensity of the caustics.

   "float maxerror" Error tolerance and for estimating the photon density.

   "float maxpixeldist" Search radius for estimating the photon density.

   "float minphotons" The minimum number of photons to gather to approximate the photon density. Higher numbers will make the photon reconstruction smoother, but also possibly blurrier.

10.6.3 Photon Propagation Parameters

The default "movephotons" parameters make an object a receiver of photons, but not a reflective or refractive object. To make it a reflector or refractor, adjust the various shader parameters as follows.

Receiver

A receiver is an object that receives caustics, but is neither reflective nor refractive (that is, photons deposit on the surface but do not continue to propagate). The default parameters of the "movephotons" shader designate an object as a receiver, Specifically, receivers are objects whose "Kr" and "Kt" parameters are both zero.

Metal / Reflector

Objects whose "movephotons" shaders have a nonzero "Kr" parameter reflect photons like a mirror. For these objects, the following parameters are significant:

"float Kr" Reflectivity of the surface.

"color specularcolor" Color filter for the reflected light.

Glass / Refractor

"float Kt" Transmission of the surface.

"color transmitcolor" Color filter for the refracted light.

"float eta" Index of refraction of the light (e.g., 1.5 for ordinary glass).

10.7 Subsurface Scattering

Subsurface scattering in the effect in which light scatters with a solid material and emerges at points far from where the light first entered the surface (see Figure 10.19). Most materials do not exhibit this effect to a significant degree, but for those that do — such as skin, marble, wax, and certain plastics — subsurface scattering is indispensible to making the materials appear photorealistic.

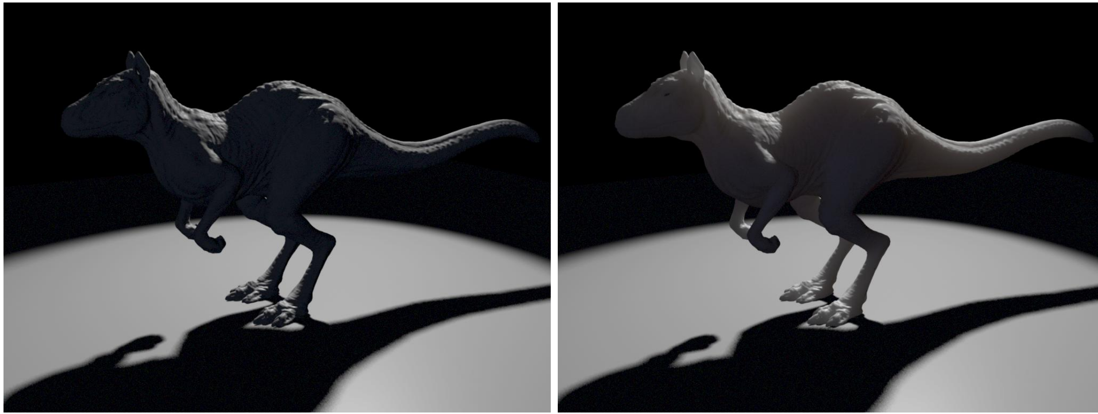

Figure 10.19: Example of subsurface scattering.

Gelato can compute subsurface scattering using a two-pass approach. In the first pass, a special shader is used to sample the diffuse illumination arriving at points on the object surface, and store those samples in a spatial database. In the second pass, that spatial database is used to reconstruct the subsurface scattering through the material.

The remainder of this subsection will provide the recipe for using subsurface scattering, using the following simple scene:

```
Attribute ("string geometryset", "+shadow")
Attribute ("string projection", "perspective")
Attribute ("float fov", 19)
AppendTransform (0.995037, -0.0170699, -0.0980286, 0, -0.0995037, -0.170699,
                 -0.980286, 0, 1.86265e-09, 0.985175, -0.17155, 0, 0, 0, 0, 1)
Translate (-2, -20, -5)
World ()
Light ("light1", "spotlight", "float intensity", 1.5, "point from", (2, -14.5, 5),
       "point to", (0, -1.5, 0), "float coneangle", 0.5, "float falloff", 0,
       "string shadowname", "shadow", "float shadowblur", 0.03,
       "float shadowsamples", 16)

PushAttributes ()
Shader ("surface", "plastic")
Attribute ("color C", (0.75, 0.75, 0.75))
Translate (0, -2, 0)
Rotate (20, 0, 0, 1)
Input ("teapot.pyg")
PopAttributes ()

Shader ("surface", "plastic")
Patch ("linear", 2, 2, "vertex point P", (-20, -20, 0, 20, -20, 0, -20, 20, 0, 20, 20, 0))

Render ("camera")
```

10.7.1 Initial pass: creating the diffuse illumination database

The first step for using subsurface scattering is to do an initial pass in which we bake the diffuse illumination into a spatial database. Compared to an ordinary rendering, the following modifications should be made to the scene:

- The object experiencing subsurface scattering should have a shader that bakes out the incident diffuse illumination, weighted by sample area. To avoid double-saving points at grid boundaries, the `spatialdbsave` should use the optional `"interpolate"` parameter, set to 1. Gelato's `bakediffuse.gsl` shader performs this task if its `"weightarea"` parameter is nonzero:

```
surface bakediffuse ( float Ka = 0,            // Ambient scaling
                      float Kd = 1,            // Diffuse scaling
                      string filename = "diffuse.sdb",
                      string sdbmode = "wo",
                      float reversenormals = 0,
                      float weightarea = 0,
                      string space = "world",
                      float interpolate = 1 )
{
    normal NN = reversenormals ? -normalize(N) : normalize(N);
    C = C * (Ka*ambient() + Kd*diffuse(NN));
    point Pworld = transform(space,P);
    normal Nworld = normalize(transformn(space,NN));
    spatialdbsave (filename, Pworld, Nworld,
                   weightarea ? C*samplearea(Pworld) : C,
                   "interpolate", interpolate, "filemode", sdbmode);
}
```

- If you have multiple disconnected objects that experience subsurface scattering, you should probably use a different spatial database for each object.

- Passing `"w"` or `"wo"` for the optional `"sdbmode"` parameter to the `spatialdbsave` function will cause the database file to be written to disk. If for some reason you don't specify this in the `spatialdbsave` call, your scene will need the appropriate `"spatialdb:write"` `Attribute`.

- It is important that the illumination is sampled fairly evenly, everywhere on the object surface. We must disable occlusion culling for the object, so that parts not visible to the camera still are shaded. Furthermore, to get an even distribution of samples over the object, it is important to turn off raster-oriented dicing (which will tend to have less sampling on parts of the object nearly perpendicular to the camera). These two changes can be made with the following attributes:

```
Attribute ("int cull:occlusion", 0)
Attribute ("int dice:rasterorient", 0)
```

- These attributes only need to be applied to the objects that experience subsurface scattering. Non-subsurface objects are needed in the initial pass only to the extent that they may affect the illumination reaching the subsurface objects (for example, if they glow, cast shadows, or contribute to global illumination). Remember that the only important output of the initial pass is the spatial database; the resulting image is never used. So you may speed up the initial pass by simplifying non-subsurface objects, such as by substituting less expensive shaders, reducing the `"shadingquality"`, etc.

Here is our example Pyg file, modified to bake out the diffuse illumination falling on the teapot:

```
Attribute ("string geometryset", "+shadow")
Attribute ("string projection", "perspective")
Attribute ("float fov", 19)
AppendTransform (0.995037, -0.0170699, -0.0980286, 0, -0.0995037, -0.170699,
                 -0.980286, 0, 1.86265e-09, 0.985175, -0.17155, 0, 0, 0, 0, 1)
Translate (-2, -20, -5)
World ()
Light ("light1", "spotlight", "float intensity", 1.5, "point from", (2, -14.5, 5),
       "point to", (0, -1.5, 0), "float coneangle", 0.5, "float falloff", 0,
       "string shadowname", "shadow", "float shadowblur", 0.03,
       "float shadowsamples", 16)
PushAttributes ()
Attribute ("int cull:occlusion", 0)
Attribute ("int dice:rasterorient", 0)
Shader ("surface", "bakediffuse", "string filename", "diffuse.sdb",
                "float weightarea", 1, "interpolate", 1)
Attribute ("color C", (0.75, 0.75, 0.75))
Translate (0, -2, 0)
Rotate (20, 0, 0, 1)
Input ("teapot.pyg")
PopAttributes ()
Shader ("surface", "plastic")
Patch ("linear", 2, 2, "vertex point P", (-20, -20, 0, 20, -20, 0, -20, 20, 0, 20, 20, 0))
Render ("camera")
```

10.7.2 Beauty pass: using subsurface scattering

Once we have baked out the incident illumination, we can easily incorporate subsurface scattering into any shader using the built-in `subsurface()` function (see Section 5.8.11). Below is a simple plastic-like shader that also incorporates subsurface scattering:

```
surface
plasticss (float Ka = 1,           // Ambient scaling
           float Kd = 0.5,          // Diffuse scaling
           float Ks = 0.5,          // Specular scaling
           float roughness = 0.1,   // Roughness
           color specularcolor = 1, // Color filter for highlights
           // Subsurface parameters:
           float Kss = 1,
           string diffusefile = "",
           color scattering = color (2.19, 2.62, 3.0),
           color absorption = color (.0021, .0041, .0071),
           float eta = 1.5,
           float maxsolidangle = 1,
           color subsurfcolor = 1,
           string ssunits = "mm",
           float ssscale = 1,
           string space = "world",
           float twosided = 0 )
{
    color Cpre = C;   // Save base surface color

    // Construct a frontfacing, unit-length normal
    normal Nf = faceforward (normalize(N), I);

    // Blend ambient, diffuse, and specular lighting
    C = C * (Ka*ambient() + Kd*diffuse(Nf)) +
        specularcolor * Ks*specular(Nf, -normalize(I), roughness);

    float visible = twosided || (dot(N,I) < 0);
    visible *= Kss;
    if (diffusefile && gridany(visible)) {
        point Pworld = transform(space,P);
```

```
        normal Nworld = normalize(transformn(space,N));
        // Convert absorption and scattering, input in 'per ssunits' (using
        // parameter 'ssunits'), into 'per common unit'.
        float scale = transformu ("common", ssunits, ssscale);
        color Css = subsurface (diffusefile, Pworld, Nworld,
                                "scattering", scattering*scale,
                                "absorption", absorption*scale,
                                "eta", eta, "maxsolidangle", maxsolidangle);
        C += Cpre * Kss * subsurfcolor * Css;
    }

    // Scale color by opacity.
    C *= opacity;
}
```

Below is our modified Pyg file for the beauty pass:

```
Attribute ("string geometryset", "+shadow")
Attribute ("string projection", "perspective")
Attribute ("float fov", 19)
AppendTransform (0.995037, -0.0170699, -0.0980286, 0, -0.0995037, -0.170699,
                 -0.980286, 0, 1.86265e-09, 0.985175, -0.17155, 0, 0, 0, 0, 1)
Translate (-2, -20, -5)
World ()
Light ("light1", "spotlight", "float intensity", 1.5, "point from", (2, -14.5, 5),
       "point to", (0, -1.5, 0), "float coneangle", 0.5, "float falloff", 0,
       "string shadowname", "shadow", "float shadowblur", 0.03,
       "float shadowsamples", 16)
PushAttributes ()
Shader ("surface", "plasticss", "string diffusefile", "diffuse.sdb")
Attribute ("color C", (0.75, 0.75, 0.75))
Translate (0, -2, 0)
Rotate (20, 0, 0, 1)
Input ("teapot.pyg")
PopAttributes ()
Shader ("surface", "plastic")
Patch ("linear", 2, 2, "vertex point P", (-20, -20, 0, 20, -20, 0, -20, 20, 0, 20, 20, 0))
Render ("camera")
```

10.7.3 Subsurface Material Properties

The `subsurface()` function in GSL (and therefore by convention, any shaders that call it, such as `plasticss.gsl`) takes three parameters that control the material properties of the subsurface scattering: scattering, absorption, and eta.

scattering controls the amount of scattering of light within the material (this is the σ_s' term you see in the papers). Higher numbers indicate more scattering (changes in the direction of light through the material), lower numbers indicate less scattering within the material. All components should be ≥ 0.

absorption controls how quickly light is absorbed as it travels through the material (this is σ_a in the papers). Higher numbers indicate that light is absorbed quickly, lower numbers indicate that light travels quite far through the material. All components should be ≥ 0.

eta is the index of refraction of the material. This should be ≥ 1 and tends to be in the range of 1.3–1.5 for most materials. This is the same index of refraction that is used to compute the Fresnel term for reflectivity of non-metallic objects.

Table 10.1 lists the measured subsurface parameters for some selected materials. These measured data are reproduced from a paper by Jensen, et al, "A Practical Model for Subsurface Light Transport," ACM SIGGRAPH 2001 (see the paper for a more complete table). You should feel free to try your own material parameters, use the ones from the table, or use a combination both.

Material	**scattering** (σ_s', in mm^{-1})	**absorption** (σ_a, in mm^{-1})	**eta** (η)
Marble	(2.19, 2.62, 3.00)	(0.0021, 0.0041, 0.0071)	1.5
Cream	(7.38, 5.47, 3.15)	(0.0002, 0.0028, 0.0163)	1.3
Skim milk	(0.7, 1.22, 1.9)	(0.0014, 0.0025, 0.0142)	1.3
Skin1	(0.74, 0.88, 1.01)	(0.032, 0.17, 0.48)	1.3
Skin2	(1.09, 1.59, 1.79)	(0.013, 0.070, 0.145)	1.3

Table 10.1: Selected measured subsurface material parameters, from Jensen, et al, "A Practical Model for Subsurface Light Transport," ACM SIGGRAPH 2001.

Because σ_s' and σ_a are measured in some physical units (in mm^{-1} in Table 10.1), the subsurface scattering will not look correct if your scene is measured in different units. By convention, shaders like `"plasticss"` have `"ssunits"` and `"ssscale"` parameters that can help you convert between physical units and scene units. For example, if your σ_s' and σ_a are measured in mm^{-1}, you should pass `"ssunits"` as `"mm"` and correctly designate the modeling units for your scene as described in Section 4.3.2. The `"ssscale"` parameter is also used to scale the unit conversions, with larger scale numbers making it seem that the objects are larger, and thus that light doesn't travel as far through their material.

10.7.4 Additional Subsurface Tips

Below are a variety of tips and things to consider when using subsurface scattering, presented in no particular order:

- The diffuse database you save in the intial pass is view-independent (if you save the points in a non-camera space, such as `"world"`). Therefore, you may reuse that database multiple times without recomputing the first pass if you are rerendering the same frame multiple times, or changing only the camera position or parameters.

- The diffuse database will remain valid as long as the lighting on the object does not change. You may reuse the database while rerendering the beauty pass with changes to material properties, including changes to the subsurface parameters. In other words, *the subsurface material properties themselves are not baked into the spatial database*, and there is no need to recompute the diffuse database just to change the subsurface parameters.

- You may be able to save time on the first pass (and disk space for the diffuse database) by reducing the `"shadingquality"` of the subsurface object, and therfore sampling the diffuse illumination less frequently. Reducing the image resolution of the first pass will also accomplish the same thing. However, if the samples are too infrequent (specifically,

if they are spaced farther apart than $1/(\sigma_s' + \sigma_a)$, which is the mean free path through the material), you will see artifacts in the beauty pass.

- In Gelato's `shaders` directory, you will see a shader called `subsurflayer.gsl`. This is intended to be used to add subsurface scattering to a shader that doesn't already incorporate the effect (perhaps one for which you don't even have the source code) by being layered at the end of an existing surface shader list.

11 Sorbetto Re-rendering

Gelato, through its Sorbetto ^{TM} re-rendering technology, allows frames to be rapidly re-rendered with changes to lights and certain other parameters. This chapter describes the API conventions allowing this.

NOTE: Sorbetto re-rendering is a Gelato Pro-only feature and is not available in basic Gelato except as a demonstration mode that will execute a limited number of re-renders.

Gelato Pro

11.1 Re-rendering Rules

Gelato allows certain re-rendering functionality and imposes restrictions on some re-rendering functionality. Some or all of these restrictions may be lifted in future versions of Gelato.

- Only the following global attributes may be changed during a re-render: memory limit attributes (such as `"limits:texturememory"`), search path attributes, statistics attributes.
- Re-renders must use the same camera each time they render the frame.
- If a re-render re-emits a `Camera` call, the replacement camera is expected to have its transformation unchanged (that is, the camera may not move when re-rendering). Additionally, most camera parameters may not change for a re-render. Currently, only the following camera parameters may be changed for re-renders: `"crop"`, `"spatialquality"`, `"temporalquality"`, `"dofquality"`, `"bucketorder"`.
- If a re-render re-emits an `Output` call, the replacement output may only change the following output attributes: `"gamma"`, `"gain"`, `"quantize"`, `"dither"`.
- `Light` may be called to add a light, re-emit an existing light source with different shader parameters or a different transformation, or delete a light (by re-emitting the light with a null shader).
- During a re-render in which `Modify` is used to change per-object attributes, the `LightSwitch` call may turn the light on or off for the set of modified states. Currently, no other per-object attributes may be altered during a re-render (including surface shaders and their parameters).
- Geometric primitives may not be added or changed during a re-render.
- `TrimCurve` may not be called during a re-render. That is, the trim curves may not be changed on existing geometry.

In future releases, we anticipate that re-renders will be able to change per-object attributes including surface shader assignments and parameters, color, and user attributes.

Although Gelato may someday support it, we do not anticipate near-term support for changing of attributes that affect what geometry is rendered, and where it appears on screen. This includes addition, deletion, movement, or deformation of geometric primitives themselves, changes to displacement shaders, attributes that affect object shape (such as trim curves, dicing attributes, shadingquality), twosided, holdout, or changing objects from opaque to transparent and vice versa.

11.2 Idioms for iterative relighting

In Gelato 2.1, the capabilities and limitations of re-rendering, as detailed in Section 11.1, primarily allow "relighting" — fast rerendering of a single frame with only the lights changing. This section describes common tasks to be performed when relighting, and the preferred idioms for accomplishing them.

All the examples assume that the global `"rerender"` attribute has been set to 1, that the full initial scene has already been emitted, thus these commands are all post-`Render`.

11.2.1 Changing which objects are illuminated by a light

`Modify` and `LightSwitch` may be used to specify which objects should be illuminated by a particular light. For example, suppose that the light `"fill"` illuminates all objects in the scene, but we wish for it not to illuminate the truck object:

```
r->World ();
r->Modify ("truck");
r->LightSwitch ("fill", 0);
r->Render ();
```

`Modify` actually matches regular expressions. Thus, given a suitable naming scheme, there is a lot of flexibility in addressing objects. For example, assuming everything in the scene is named in a hierarchical fashion much like file directory entries, just the fenders and tail lights (and all their sub-parts) of trucks40 through 45 could be illuminated like this:

```
...
r->Modify ("^truck4[0-9]/(fender)|(taillight)/.*");
r->LightSwitch ("fill", 1);
...
```

11.2.2 Adding a light to the scene

Adding a new light to the scene is as simple as calling `World`, setting the transformation (and possibly other attributes) for the light, and calling `Light` with a unique *lightid* that has not yet been used by any other light the scene.

The tricky part is that all of the primitives already in the scene have attribute states whose active light lists do not include the new light source. The solution is to use `Modify` and `LightSwitch` to specify which objects should be illuminated.

```
r->World ();
r->PushTransform ();
Matrix4 M (1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, -5, 0);
r->SetTransform ((float *)&M);
r->Parameter ("float intensity", 10);
r->Parameter ("float falloff", 2)
r->Light ("key", "spotlight");
r->PopTransform ();
r->Modify ("");
r->LightSwitch ("key", 1);
r->Render ();
```

The above example creates a new point light with the *lightid* `"key"`, and causes it to illuminate everything in the scene. We could easily illuminate only part of the scene with a suitable pattern passed to `Modify`.

11.2.3 Changing the parameters of a light

You may change any of the shader parameters of an existing light by re-emitting the entire light source with the desired parameter set. Re-emitting a light follows the same procedure as adding a light, except that it uses the *lightid* of an existing light.

Because `Light` sets the light's `"shader"` space to be the CTM at the time of the `Light` call, it is important to fully specify the transformation of the light when re-emitting it. Otherwise, it will inherit the current transformation, which is almost certainly not what it was supposed to be. The easiest way to do this is to use `SetTransform` to "place" the re-emitted light in the same position and orientation as it had before.

For example, the following code re-emits the key light with a new `"intensity"` value:

```
r->World ();
r->PushTransform ();
Matrix4 M (1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, -5, 0);
r->SetTransform ((float *)&M);
r->Parameter ("float intensity", 40);
r->Parameter ("float falloff", 2)
r->Parameter ("string shadowname", "key.sm")
r->Light ("key", "spotlight");
r->PopTransform ();
r->Render ();
```

11.2.4 Moving or re-orienting a light

Moving or re-orienting a light follows the same procedure as above, except that (1) the transformation prior to `Light` should place it in the new desired position and orientation; and (2) the re-emitted `Light` call should specify all the same parameters as before, without modification:

```
r->World ();
r->PushTransform ();
Matrix4 M (1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 3, 0, 0, 0);
r->SetTransform ((float *)&M);
```

```
r->Parameter ("float intensity", 40);
r->Parameter ("float falloff", 2)
r->Parameter ("string shadowname", "key.sm")
r->Light ("key", "spotlight");
r->PopTransform ();
r->Render ();
```

However, moving a light in this manner does not automatically change the shadow map `"key.sm"`. It is up to the calling application to cause shadow maps to recompute (possibly by launching an entirely separate render).

When a shadow map changes on disk (or any disk file that's accessed by shaders, such as environment maps, texture maps, or SDB files), objects whose shaders access those files will automatically be re-shaded upon re-rendering. This will happen even if none of the object's shader or light parameters have themselves changed, because the renderer will notice that the map on disk has been updated since the last re-render. However, it is not entirely free to be checking access times of every texture, especially if there are thousands of textures which may be located on a network server. So there is an Attribute, `"rerender:filepattern"` that allows you to provide a regular expression that narrows down which files to check (all others are assumed to not change within a rerendering session). For example, to instruct the renderer to only check shadow and environment maps for changes, but not ordinary textures:

```
r->Attribute ("string rerender:filepattern", ".*\.(sm)|(env)");
```

11.2.5 Deleting a light

Deleting a light entirely is easily accomplished by replacing a light and specifying that its shader is `"null"`:

```
r->World ();
r->Light ("fill", "null");
r->Render ();
```

11.3 Idioms for improving performance

11.3.1 Selecting a crop window

A re-render will happen much more quickly if it is restricted to only part of the scene. This is easily accomplished by re-emitting the camera with a new crop window.

```
r->SetTransform (...);    // camera position
int res[2] = { 640, 480 };
float crop[4] = { .25, .5, .38, .72 };
int sq[2] = { 4, 4 };
r->Parameter ("int[2] resolution", &res);
r->Parameter ("float[4] crop", &crop);
r->Parameter ("int[4] spatialquality", &sq);
r->Camera ("maincam");
r->World ();
r->Render ();
```

11.3.2 Changing antialiasing settings

Although re-renders for lighting preview may happen with full antialiasing, faster relighting updates may be achieved by reducing the spatial quality to one sample per pixel. The spatial quality may be freely changed between subsequently re-rendered frames.

```
r->SetTransform (...);    // camera position
int res[2] = { 640, 480 };
int sq[2] = { 1, 1 };
r->Parameter ("int[2] resolution", &res);
r->Parameter ("float[4] crop", &crop);
r->Parameter ("int[4] spatialquality", &sq);
r->Camera ("maincam");
r->World ();
r->Render ();
```

11.3.3 Locking shading

When re-rendering, the renderer tries to only re-run shaders whose parameters or dependencies have changed. But it is fairly conservative in this analysis, so you can often speed up re-rendering by explicitly specifying that particular objects have their shading "locked" and thus do not need to be re-shaded. It may be very helpful for users of a lighting application to select objects that do or do not update as lights change in an interactive session. Shading may be locked for a set of objects by using the `"int rerender:locked"` attribute.

```
r->World ();
r->Modify ("");
r->Attribute ("int rerender:locked", 1)
r->Modify ("^truck/.*");
r->Attribute ("int rerender:locked", 0)
r->Render ();
```

The above fragment locks shading for the entire scene, then unlocks it for the truck and all its parts. Thus, subsequent re-renders should go very quickly because they will not even consider any other objects as candidates for reshading.

Part III

Developer Tools

12 Generator Plugins and Scene File Readers

`Input(`*command*`)` will dynamically load and execute a generator plugin from a DSO/DLL. The renderer will use a dynamic library named *name*`.generator.so` (or, under Windows, *name*`.generator.dll`), where *name* is the first word (up to a space) of *command*.

In the case that the *command* passed to `Input` is actually the name of a scene file in the "input" search path, then *command* is replaced with "*format filename*" where *filename* is the full path of the file, and *format* is the format of the file (given simply by *filename*'s extension). It is presumed that there is a *format* generator which will read commands from the named file (as its sole argument) and make the appropriate Gelato API calls.

For example,

```
Input ("teapot.obj")
```

is equivalent to:

```
Input ("obj teapot.obj")
```

Either of these commands will cause the renderer to load a shared library named `"obj.generator.so"` (or `"obj.generator.dll"` under Windows), which is presumed to be somewhere in the "generator" search path.

Generator DSO/DLL's are expected to:

1. Implement a class that is a subclass of an `Generator` class defined as:

```
class Generator {
public:
    enum { DSO_VERSION=0 };
    Generator();
    virtual ~Generator();
    virtual void run (GelatoAPI *renderer, const char *params);
};
```

The subclass, which publicly inherits from `Generator`, must define a `run` method. Optionally, it may implement a replacement constructor and destructor, as well as any additional data or function methods that its implementation requires.

2. Contain a C-linkage function called `name_generator_create` that returns a pointer to a newly allocated and constructed `Generator` object. Using the `GELATO_EXPORT` macro (defined in `export.h`) ensures that the renderer can correctly reference the symbol.

3. Contain an integer variable called `generator_version` that contains the value `GelatoAPI::API_VERSION`. This is necessary for checking version compatibility between the renderer and the plugin. Using the `GELATO_EXPORT` macro ensures that the renderer can correctly reference the symbol.

So, for example, below is a skeleton to implement a generator that acts as a scene reader for "`.obj`" files. The compiled C++ module should be stored in a file named "`obj.generator.so`" (or "`obj.generator.dll`").

```
class ObjReader : public GelatoAPI::Generator {
public:
    ObjReader();
    virtual ~ObjReader();
    virtual void run (GelatoAPI *renderer, const char *filename);
};


ObjReader::ObjReader (void)
{
    // Implementation of ObjReader constructor goes here
}


ObjReader::~ObjReader (void)
{
    // Implementation of ObjReader destructor goes here
}


void
ObjReader::run (GelatoAPI *renderer, const char *filename)
{
    // Here we read "Obj" from filename, and make GelatoAPI calls
    // to renderer in order to communicate the commands in the file
    // to Gelato.
}


extern "C" {

GELATO_EXPORT int generator_version = GelatoAPI::API_VERSION;

GELATO_EXPORT GelatoAPI::Generator *obj_generator_create (const char *command)
{
    return new ObjReader;
}
```

```
};
```

13 Image I/O Plugins

In the course of rendering, various parts of Gelato will input or output images. Examples include:

- Writing rendered images to disk.
- Reading tiles from MIPmap images used as texture, shadow, and environment maps.
- Reading images in arbitrary formats for `maketx` to convert to texture, shadow, and environment maps (and of course, `maketx` must *write* those map files in the proper tiled formats).
- Displaying images with `iv`.

Gelato does not natively understand how to read or write images of any format.[1] Instead, all requests to read and write images, parts of images (such as individual scanlines, tiles, or subimages), or query image information go through a simple, publicly-documented API that relies on user-supplied DSO's that contain code that correctly reads and writes from image files of various formats.

13.1 Formats and Plugins

When Gelato (or any application) uses ImageIO classes to write an image file, it does so by calling `MakeImageOutput`, which has the following declaration:

```
ImageOutput *MakeImageOutput (const char *filename,
                              const char *format,
                              const char *searchpath);
ImageOutput *MakeImageOutput (const char *filename,
                              const char *searchpath);
```

In the first form, the caller provides a format name. In the second form, it is assumed that the file extension (the part of the filename following the last period) is the name of the format. First, a search will be performed looking for *`format`*`.imageio.so` (Linux) or *`format`*`.imageio.dll` (Windows). If no such DSO/DLL is found, *every* ImageIO plugin in the search path will be

[1] Technically, that's a fib. A TIFF (.tif) Image I/O plugin is compiled into `gelato` so that images may be written and textures may be read, even if no DSO is found. But this is merely a case of packaging the DSO in the executable; its operation is identical to the general Image I/O plugin method described in this section.

opened until one is found with a symbol called `format_output_extensions` (that contains a list of supported formats or file extensions) that matches the format or file extension specified.

Similarly, when an application uses ImageIO classes to read from an image file, it does so by calling `MakeImageInput` which has the following declaration:

```
ImageInput *MakeImageInput (const char *filename,
                            const char *searchpath);
```

The `MakeImageInput` routine will first look for a plugin called *extension*`.imageio.so` (on Linux) or *extension*`.imageio.dll` (on Windows), where *extension* is the file extension of the input filename. If no such exact plugin name is found, then every ImageIO plugin in the searchpath will be opened until one is found with a symbol called `format_input_extensions` (that contains a list of supported formats or file extensions) that matches the extension of the input file. If no plugin is found that explicitly names the extension desired as supported, then all such plugins will be attempted simply by calling their `open()` methods to see if any will successfully open the file.

13.2 Writing Image I/O Plugins

These *Image I/O Plugins* implement subclasses of abstract classes called `ImageOutput` and `ImageInput`, both of which are documented in the header file `imageio.h`, which is included in the Gelato distribution. The contents of `imageio.h` are contained within the `Gelato::` namespace. The easiest thing to do is to tell the C++ compiler to automatically resolve symbols in this namespace:

```
using namespace Gelato;
```

Of course, if you also need to use libraries from some other source that have unqualified names in common with the Gelato libraries, you may wish to forgo the `using` directive and explicitly prepend `Gelato::` onto all references to symbols defined in the header. For the remainder of this section, we will assume that you are employing the `using namespace Gelato` directive.

The plugin library file should be named *format*`.imageio.so` (Linux) or *format*`.imageio.dll` (Windows).

Helper structures

When opening files for reading or writing, information about the format of the data (either present in the file or requested by the client) is communicated via an `ImageIOFormatSpec` structure:

```
struct ImageIOFormatSpec {
    int x, y, z;              // image origin (0,0,0)
    int width, height, depth; // width, height, depth (depth>1 for volume)
    int full_width;           // width of entire image (not just cropwindow)
    int full_height;
    int full_depth;
```

```
    int tile_width;           // tile size (0 if tiles are not supported)
    int tile_height;
    int tile_depth;
    int nchannels;            // e.g., 4 for RGBA
    ParamBaseType format;     // format of data in each channel
    std::vector<const char*> channelnames;  // e.g., {"R","G","B","A"}
    char unused[256];         // for future expansion
};
```

Additional parameters controlling options of format plugins are communicated via `ImageIOParameter` structures, defined as:

```
struct ImageIOParameter {
    const char *name;           // Token string
    ParamBaseType type;         // data type
    int nvalues;                // number of elements
    const void *value;          // array of values

    ImageIOParameter () {};
    ImageIOParameter (const char *name, ParamBaseType type,
                      int nvalues, const void *value)
        : name(name), type(type), nvalues(nvalues), value(value) {}
};
```

13.3 Image Writers

To expand Gelato's file-writing repertoire to include a new format, you must provide a DSO named *format*`.imageio.so`. The DSO must contain the following two items:

```
extern "C" {
    GELATO_EXPORT ImageOutput *format_output_imageio_create ()
    {
        ...
    }

    GELATO_EXPORT int imageio_version = IMAGEIO_VERSION;
};
```

The purpose of the create function is to return an `ImageOutput` object that implements writing the desired format. Since `ImageOutput` is an abstract base class, this involves deriving a subclass and overloading various virtual functions. The create function then constructs an instance of this subclass and returns it.

Additionally, the DSO may supply a list of extensions it supports for writing by having a symbol called *format*`_output_extensions`. This symbol should be an array of `char *`'s, each of which points to a null-terminated string giving a supported file extension, with the end denoted by a NULL pointer. For example:

```
extern "C" {
    GELATO_EXPORT const char *format_output_extensions[] = {
```

```
            "ext1", "ext2", NULL };
    };
```

13.3.1 Functionality Queries

An ImageOutput subclass may overload any or all of the following query methods to allow the renderer to understand its capabilities. In all cases, if these functions are not overloaded with an implementation that returns true, the default (inherited) implementation will return false, signifying that the plugin does not support the feature and that the renderer should therefore not request it. Therefore, you need not override these functions if your plugin does not support the named feature.

`bool supports_tiles (void) const`

Return true if the plugin supports tiled output, false if the plugin only supports scanline-oriented output.

`bool supports_random_access (void) const`

Return true if the plugin supports random access (i.e., scanlines or tiles may be written in arbitrary order). Return false to signal that the renderer may only send scanlines or tiles in order, buffering up data on the renderer side if the pixels are generated out of order.

`bool supports_multiimage (void) const`

Return true if the plugin supports multiple images in one file.

`bool supports_volumes (void) const`

Return true if the plugin supports writing volumetric (3D) image data, false if the format is strictly a 2D image format.

`bool supports_rewrite (void) const`

Return true if the plugin allows the client (renderer) to send data for particular scanlines or tiles more than once (for example, for progressive update or redisplay). It is strongly encouraged that Image I/O plugins that implement live displays (such as iv) support this feature; it is generally not necessary for plugins that write images to files.

`bool supports_empty (void) const`

Return true if the plugin will properly handle write_scanline and write_tile calls with a NULL data pointer indicating a scanline or tile that is entirely composed zero values. Return false to indicate that the client (renderer) should actually allocate, fill, and pass a buffer of zero values to indicate a blank scanline or tile. (Supporting empty data can greatly reduce the communication overhead between the renderer and plugin for images with empty regions.)

```
bool supports_rectangles (void) const
```

Return `true` if the plugin will properly handle `write_rectangle` calls to send arbitrary rectangles of pixels (rather than only entire scanlines or tiles). Return `false` to indicate that the client (renderer) should only send scanlines or tiles. It is strongly encouraged that Image I/O plugins that implement live displays (such as `iv`) support this feature; it is generally not necessary for plugins that write images to files.

13.3.2 Data Writing Methods

A `ImageOutput` subclass implements the following methods;

```
bool open (const char *name, const ImageIOFormatSpec &spec,
           int nparams, const ImageIOParameter *param, bool append=false)
```

Initial opening of a file. The *name* is the name of the file to write. The *spec* is a reference to an `ImageIOFormatStruct` that dictates what format the client (renderer) would like to write. Additional parameters controlling options of the format plugin are communicated in the `param[0..nparams-1]`.

If the value of *append* is `true`, the plugin should attempt to append a new subimage onto the file (if it already exists), rather than deleting the file (note that this will never be requested of the plugin if its `supports_multiimage()` method returns `false`). It is legal to open a file multiple times without an intervening call to `close()` provided the file format supports multiple images and the subsequent calls to `open()` set the *append* flag to `true`.

Different format plugins will support different parameters communicated in `param`. The author of an Image I/O plugin should provide documentation on exactly what parameters are supported. At a minimum, all plugins should support the following parameters to the best of their ability:

`"quantize"` An `int[4]` containing the quantization parameters (just as in Section 4.2).

`"dither"` A `float` giving the dither amplitude.

`"gain"` A `float` giving a gain multiplier for all color data (but not alpha).

`"gamma"` A `float` giving the gamma correction exponent for all color data (but not alpha).

`"worldtocamera"` A `float[16]` giving the world-to-camera matrix of the rendering.

`"worldtoscreen"` A `float[16]` giving the transformation that converts world space to perspective-corrected view space randing from -1 to 1 in both x and y, and 0 to 1 in z (clip coordinates).

The `open` function should return `true` if the operation succeeds, `false` if it did not.

```
bool close ()
```

Called when the client (renderer) is done writing an image. No more pixel data will be sent.

```
bool write_scanline (int y, int z, const float *data,
                     int xstride)
```

Write the scanline that includes pixels (*,y,z). Note that $z = 0$ for non-volume images. The stride value gives the data layout: one pixel to the "right" is *xstride* floats away. Return `true` for success, `false` for failure. It is a failure to call `write_scanline()` with an out-of-order scanline if this format driver does not support random access.

Note that the client (renderer) *always* passes the image data as `float`'s. It is the job of the plugin to convert the `float` data into whatever format was specified to `open`.

```
bool write_tile (int x, int y, int z,
                 const float *data, int xstride, int ystride, int zstride)
```

Write the tile with (x,y,z) as the upper left corner ($z = 0$ for non-volume images). The three stride values give the data layout: one pixel to the "right" is *xstride* floats away, one pixel "down" is *ystride* floats away, one pixel "in" (the next volumetric slice) is *zstride* floats away. Return `true` for success, `false` for failure. It is a failure to call `write_tile()` with an out-of-order tile if this format driver does not support random access.

The *data* points to an area holding $tilewidth \times tileheight \times tiledepth$ pixels. This is true even if this tile is part of the rightmost column or bottommost row and $x + tilewidth > width$ or $y + tileheight > height$. Pixels outside the image window will not actually written to the file, but a full tile of pixels is still passed in.

Note that the plugin does not need to provide a `write_tile` method if the `supports_tiles` function returns `false`. If none is provided, a default (do-nothing) implementation will be inherited from the `ImageOutput` base class.

```
bool write_rectangle (int xmin, int xmax, int ymin, int ymax,
                 int zmin, int zmax, const float *data,
                 int xstride, int ystride, int zstride)
```

Write a rectangle of data with $(xmin, ymin, zmin)$ as the upper left corner ($z = 0$ for non-volume images) and including $(xmax, ymax, zmax)$ as the lower right corner. The three stride values give the data layout: one pixel to the "right" is *xstride* floats away, one pixel "down" is *ystride* floats away, one pixel "in" (the next volumetric slice) is *zstride* floats away. Return `true` for success, `false` for failure. It is a failure to call `write_rectangle()` for an Image I/O plugin that does not return `true` for both `supports_rectangles()` and `supports_random_access()`.

The *data* points to an area holding $tilewidth \times tileheight \times tiledepth$ pixels. This is true even if this tile is part of the rightmost column or bottommost row and $x + tilewidth > width$ or $y + tileheight > height$. Pixels outside the image window will not actually written to the file, but a full tile of pixels is still passed in.

Note that the plugin does not need to provide a `write_rectangle` method if the `supports_-rectangles` function returns `false`. If none is provided, a default (do-nothing) implementation will be inherited from the `ImageOutput` base class.

13.3.3 Message Passing

A simple message passing interface between the client and plugin is provided for expansion:

```
int send_to_output (const char *format, ...)
```

This function may be called by the client (renderer). If the plugin chooses to implement the function, it may interpret certain messages. Currently, no messages are sent by Gelato; this is strictly for future expansion of the protocol.

```
int send_to_client (const char *format, ...)
```

This function is implemented in the `ImageOutput` base class — plugins do not implement it. Though no messages are currently supported by Gelato, this facility is envisioned as an opportunity for future expansion (say, an interactive image display program that accepts mouse commands to sweep out a crop window, and communicates the crop coordinates back to the renderer for further image refinement).

13.3.4 Error Handling

Any of your implemented methods, in addition to returning `false` upon failure, may set an error string as follows:

```
void error (const char *message, ...)
```

This function takes arguments following the conventions of `printf`. It is not to be implemented by your plugin — it is already provided in the `ImageOutput` base class. But your functions may call it to set the current error string to a detailed error message (which will presumably be printed or logged by the client, if appropriate).

13.3.5 Odds and Ends

Your `ImageOutput` subclass may, of course, also implement a constructor, destructor, private methods used internally to your class (but obviously not called by the client, since they are not part of the `ImageOutput` public API), and have whatever private data members are appropriate.

13.4 Image Readers

Although many renderers support a mechanism to allow user-defined output formats (though perhaps less cleanly than Gelato's `ImageOutput` protocol), Gelato is relatively unique in also allowing users to implement image reader plugins. As you might guess, these are analogous to image writers, but allow `gelato`, `maketx`, `iv`, and other Gelato components to read arbitrary image formats.

To expand Gelato's file-reading, you must provide a DSO named *format*`.imageio.so`. The DSO must contain the following two items:

```
extern "C" {
    GELATO_EXPORT ImageInput *format_input_imageio_create ()
    {
        ...
    }

    GELATO_EXPORT int imageio_version = IMAGEIO_VERSION;
};
```

The purpose of the create function is to return an `ImageInput` object that implements reading the desired format. Since `ImageInput` is an abstract base class, this involves deriving a subclass and overloading various virtual functions. The create function then constructs an instance of this subclass and returns it.

Additionally, the DSO may supply a list of extensions it supports for reading by having a symbol called *format*`_input_extensions`. This symbol should be an array of `char *`'s, each of which points to a null-terminated string giving a supported file extension, with the end denoted by a NULL pointer. For example:

```
extern "C" {
    GELATO_EXPORT const char *format_input_extensions[] = {
        "ext1", "ext2", NULL };
};
```

13.4.1 Opening and Closing

```
bool open (const char *name, ImageIOFormatSpec &newspec,
           int nparams, const ImageIOParameter *param)
```

Opens an image file with the given *name*. The `open()` method writes format information about the first image in the file into *newspec*, and is also expected to save a copy in `this->spec`. Returns `true` if the file successfully is opened, `false` if an error occurred that prevents reading of the named file.

Additional parameters may be passed in `param[0..nparams-1]`. Authors of `ImageInput` should publish the list of parameters supported by any particular plugin.

```
int current_subimage (void) const
```

Returns the current subimage being examined, for multi-image files. If the format (or reader plugin) does not support multi-image files, this function should not be implemented (it will inherit a version from the base class that always returns 0).

`bool seek_subimage (int index, ImageIOFormatSpec &newspec)`

Resets the current subimage to the *index*'th image in the file, writing the format information for that subimage into *newspec* (and also saving the new format information in `this->spec`). Future reads or queries are assumed to refer to the indexed subimage. Return `true` upon success, `false` upon failure (including that there are fewer subimages than the *index* requests). If the format (or reader plugins) does not support multi-image files, this function should not be implemented (it will inherit a version from the base class that always returns false).

`bool close ()`

Called by the client when no more scanlines or tiles are required.

13.4.2 Data Reading Methods

`ImageInput` subclass implements the following methods;

`bool read_native_scanline (int y, int z, void *data)`

Read the scanline that includes pixels $(*, y, z)$ into contiguous memory beginning at *data* (note that $z = 0$) for non-volume images). The data should be kept in the native (uncompressed) data format of the file. It is the caller's responsibility to ensure that *data* points to a large enough piece of memory, the size of which should be

```
width * channels * ParamBaseTypeSize(format)
```

Return `true` upon success, `false` upon failure. It may be considered a failure to read a scanline from a tiled file, though an individual imageio plugin may choose to "simulate" scanline access to a tiled image file.

If a reader supports scanline access at all, it is expected to give the appearance of random access – in other words, if it can't randomly seek to the given scanline, it should transparently close, reopen, and sequentially read through prior scanlines.

`bool read_native_tile (int x, int y, int z, void *data)`

Read the tile that includes pixel (x, y, z) into contiguous memory beginning at *data* (note that $z = 0$ for non-volume images). The data should be kept in the native data format of the file (though uncompressed). It is the caller's responsibility to ensure that *data* points to a large enough piece of memory, the size of which should be

```
tile_width * tile_height * tile_depth * channels * ParamBaseTypeSize(format)
```

Note that *data* points to an area big enough to hold $tilewidth \times tileheight$ pixels, even if this tile is part of the rightmost column or bottommost row and $x + tilewidth > width$ or $y + tileheight > height$. Pixels outside the image window will simply not have their values filled in.

Return `true` upon success, `false` upon failure. Plugins that do not support tiled image reading need not implement this function; an inherited default exists that simply returns `false`.

The reader is expected to give the appearance of random access – in other words, if it can't randomly seek to the given tile, it should transparently close, reopen, and sequentially read through prior tiles.

```
bool read_scanline (int y, int z, float *data)
bool read_scanline (int y, int z, float *data, int xstride)
bool read_tile (int x, int y, int z, float *data)
bool read_tile (int x, int y, int z, float *data,
                int xstride, int ystride, int zstride)
```

Similar to `read_raw_scanline` and `read_raw_tile`, except that these routines are expected to convert the data into `float` format, and optionally may read into noncontiguous memory.

The stride values, if supplied, give the layout of the caller's *data* buffer: one pixel to the "right" is *xstride* floats away, one pixel "down" is *ystride* floats away, one pixel "in" (the next volumetric slice) is *zstride* floats away. If no stride values are supplied, it is assumed that the caller's *data* buffer expects entire scanlines or tiles to end up in contiguous memory.

Default implementations of `read_scanline` and `read_tile` exist in the `ImageInput` parent class and simply call the corresponding `read_native` routine to read into a temporary buffer, and then convert the data into floats in the client's buffer while honoring the caller's stride values. These default routines will work properly for linear unsigned 8- and 16-bit integer data, float, and half (16-bit float) data. However, image formats that store data in some other format (say, a special 10-bit log-encoded format) may need to provide their own implementation of `read_scanline` and/or `read_tile` to properly convert to float.

```
bool get_parameter (const char *name, Gelato::ParamType t, void *val)
```

Try to find a named parameter from the currently opened image (or subimage) and store its value in `*val`. Return `true` if the plugin knows about that parameter, it's of the type requested, and it's in the file. Return `false` (and don't modify *val) if the parameter *name* is unrecognized, is of the wrong type, or doesn't have an entry in the file.

13.4.3 Message Passing

A simple message passing interface between the client and plugin is provided for expansion:

```
int send_to_input (const char *format, ...)
```

This function may be called by the client (renderer). If the plugin chooses to implement the function, it may interpret certain messages. Currently, no messages are sent by Gelato; this is strictly for future expansion of the protocol.

```
int send_to_client (const char *format, ...)
```

This function is implemented in the `ImageInput` base class — plugins do not implement it. Though no messages are currently supported by Gelato, this facility is envisioned as an opportunity for future expansion.

13.4.4 Error Handling

Any of your implemented methods, in addition to returning `false` upon failure, may set an error string as follows:

```
void error (const char *message, ...)
```

This function takes arguments following the conventions of `printf`. It is not to be implemented by your plugin — it is already provided in the `ImageOutput` base class. But your functions may call it to set the current error string to a detailed error message (which will presumably be printed or logged by the client, if appropriate).

13.4.5 Odds and Ends

Your `ImageInput` subclass may, of course, also implement a constructor, destructor, private methods used internally to your class (but obviously not called by the client, since they are not part of the `ImageOutput` public API), and have whatever private data members are appropriate.

13.5 Other Helper Functions

The following functions are also defined in `imageio.h`, in the `namespace ImageIO`. You may use them in the implementation of Image I/O plugins:

```
const void *IOParamFindValue (const char *name, ParamBaseType type,
             int count, int& index, int nparams, const ImageIOParameter *param)
```

Searches the parameters in `param[0..nparams-1]` for one matching the *name*, *type*, and item *count*. If an exact match is found, *index* is set to the position in the *param* array of the first match, and the return value is the data address of that parameter. If no match is found, `index` is set to -1, and NULL is returned.

```
const char *IOParamFindString (const char *name, int nparams,
             const ImageIOParameter *param)
```

Similar to `IOParamFindValue`, but for the specific case of searching for a string parameter – the `char *` address of the string data is returned.

```
int quantize (float value, int black, int white,
             int clamp_min, int clamp_max, float ditheramp)
```

Implements quantization — given black and white, min and max, and dither amounts, converts a float *value* to its quantized integer representation.

```
float exposure (float value, float gain, float invgamma)
```

Implements the exposure operation, i.e., applying gain and gamma.

13.6 Example: JPEG ImageIO

The following code implements a JPEG reader and writer for Gelato. It depends on the publicly-available `libjpeg6`, which comes with most Linux distributions. It should be compiled to create the file named `jpg.imageio.so`.

```
///////////////////////////////////////////////////////////////////////////
// Copyright 2004 NVIDIA Corporation.  All Rights Reserved.
//
// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions are
// met:
//
// * Redistributions of source code must retain the above copyright
//   notice, this list of conditions and the following disclaimer.
// * Redistributions in binary form must reproduce the above copyright
//   notice, this list of conditions and the following disclaimer in the
//   documentation and/or other materials provided with the distribution.
// * Neither the name of NVIDIA nor the names of its contributors
//   may be used to endorse or promote products derived from this software
//   without specific prior written permission.
//
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
// "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
// LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
// A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
// OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
// SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
// LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
// DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
// THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
// (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
// OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
//
// (This is the Modified BSD License)
///////////////////////////////////////////////////////////////////////////

#include <assert.h>
#include <malloc.h>
#include <string.h>
#include <ctype.h>
#include <cstdio>
#include <setjmp.h>

#include "imageio.h"
#include "jpeglib.h"

using namespace Gelato;


// See JPEG library documentation in /usr/share/doc/libjpeg-devel-6b


class JpgOutput : public ImageOutput {
 public:
    JpgOutput() : fd(NULL), yorigin(0), gain(1), invgamma(1), black(0),
        white(255), min(0), max(255), ditheramp(0.5) {}
    bool open (const char *name, const ImageIOFormatSpec &spec,
        int nparams, const ImageIOParameter *param, bool append=false);
    bool write_scanline (int y, int z, const float *data, int xstride);
    bool close ();
 private:
```

```
    FILE *fd;
    int yorigin;
    JSAMPROW row[1];
    struct jpeg_compress_struct cinfo;
    struct jpeg_error_mgr jerr;
    float gain, invgamma;
    int black, white, min, max;
    float ditheramp;
};



extern "C" {
    GELATO_EXPORT JpgOutput *jpg_output_imageio_create () {
        return new JpgOutput;
    }

    GELATO_EXPORT const char *jpg_output_extensions[] = { "jpg", "jpe", "jpeg", NULL };
};




bool
JpgOutput::open (const char *name, const ImageIOFormatSpec &spec, int nparams,
    const ImageIOParameter *param, bool append)
{
    if (append)
        return false;

    fd = fopen (name, "wb");
    if (fd == NULL) {
        return false;
    }

    int quality = 98;

    yorigin = spec.y;
    cinfo.err = jpeg_std_error (&jerr);                 // set error handler
    jpeg_create_compress (&cinfo);                      // create compressor
    jpeg_stdio_dest (&cinfo, fd);                       // set output stream

    // set compression parameters
    cinfo.image_width = spec.width;
    cinfo.image_height = spec.height;

    if (spec.nchannels == 4) {
        fprintf (stderr, "Warning: JPEG only supports RGB images. The alpha "
            "channel will be discarded.\n");
        cinfo.input_components = 3;
        cinfo.in_color_space = JCS_RGB;
    } else if (spec.nchannels == 3) {
        cinfo.input_components = 3;                     // no JPEG RGBA!
        cinfo.in_color_space = JCS_RGB;
    } else if (spec.nchannels == 1) {
        cinfo.input_components = 1;
        cinfo.in_color_space = JCS_GRAYSCALE;
    }

    // allocate a buffer to store the row data for one scanline
    row[0] = new unsigned char[cinfo.image_width * cinfo.input_components];

    jpeg_set_defaults (&cinfo);                         // default compression
```

```
    jpeg_set_quality (&cinfo, quality, TRUE);          // baseline values
    jpeg_start_compress (&cinfo, TRUE);                // start working

    // Gather options from parameter list
    int index;
    const float *float_value;
    const int *int_value;
    float_value = (float *)IOParamFindValue ("gamma", PT_FLOAT, 1,
        index, nparams, param);
    if (float_value != NULL && *float_value != 0)
        invgamma = 1.0f / *float_value;
    float_value = (float *)IOParamFindValue ("gain", PT_FLOAT, 1,
        index, nparams, param);
    if (float_value != NULL) gain = *float_value;
    int_value = (int *)IOParamFindValue ("quantize", PT_INT, 4,
        index, nparams, param);
    if (int_value != NULL) {
        black = int_value[0];
        white = int_value[1];
        min = int_value[2];
        max = int_value[3];
        // override bit depth requested by client
        if ((min == 0 && max == 0 && white == 0) || max > 255) {
            fprintf (stderr, "Warning: JPEG only supports 8bit output.\n");
        }
    }
    // make sure to set ditheramp after we check quantize for float ouput
    float_value = (float *)IOParamFindValue ("dither", PT_FLOAT, 1,
        index, nparams, param);
    if (float_value != NULL) ditheramp = *float_value;
    else if ((black == 0) && (white == 0) && (min == 0) && (max == 0))
        ditheramp = 0;

    return true;
}



bool
JpgOutput::write_scanline (int y, int z, const float *data, int xstride)
{
    y -= yorigin;
    assert (y == (int)cinfo.next_scanline);
    assert (y < (int)cinfo.image_height);

    // convert the floating point data into 8bit one scanline at a time
    unsigned char *cur = row[0];
    for (unsigned int i = 0; i < cinfo.image_width; i++, data += xstride) {
        for (int j = 0; j < cinfo.input_components; j++) {
            float p = exposure (data[j], gain, invgamma);
            *cur++ = quantize (p, 0, 255, 0, 255,
                cinfo.input_components == 1 ? 0 : ditheramp);
        }
    }

    jpeg_write_scanlines (&cinfo, row, 1);

    return true;
}



bool
```

```
JpgOutput::close ()
{
    jpeg_finish_compress (&cinfo);
    fclose (fd);
    fd = NULL;
    jpeg_destroy_compress (&cinfo);
    delete row[0];

    return true;
}


///////////////////////////////////////////////////////////////////


class JpgInput : public ImageInput {
 public:
    JpgInput () : fd(NULL) {}
    ~JpgInput () {}
    bool open (const char *name, ImageIOFormatSpec &spec,
        int nparams, const ImageIOParameter *param);
    bool read_native_scanline (int y, int z, void *data);
    bool close ();
 private:
    FILE *fd;
    struct jpeg_decompress_struct cinfo;
    struct jpeg_error_mgr jerr;
};


// Export version number and create function symbols
extern "C" {
    GELATO_EXPORT int imageio_version = IMAGEIO_VERSION;
    GELATO_EXPORT JpgInput *jpg_input_imageio_create () {
        return new JpgInput;
    }
    GELATO_EXPORT const char *jpg_input_extensions[] = { "jpg", "jpe", "jpeg", NULL };
};



bool
JpgInput::open (const char *name, ImageIOFormatSpec &newspec,
    int nparams, const ImageIOParameter *param)
{
    // Check that file exists and can be opened
    fd = fopen (name, "rb");
    if (fd == NULL)
        return false;

    // Check magic number to assure this is a JPEG file
    int magic = 0;
    fread (&magic, 4, 1, fd);
    rewind (fd);
    const int JPEG_MAGIC = 0xffd8ffe0, JPEG_MAGIC_OTHER_ENDIAN =  0xe0ffd8ff;
    const int JPEG_MAGIC2 = 0xffd8ffe1, JPEG_MAGIC2_OTHER_ENDIAN =  0xe1ffd8ff;
    if (magic != JPEG_MAGIC && magic != JPEG_MAGIC_OTHER_ENDIAN &&
        magic != JPEG_MAGIC2 && magic != JPEG_MAGIC2_OTHER_ENDIAN) {
        fclose (fd);
        return false;
    }

    cinfo.err = jpeg_std_error(&jerr);
```

```
    jpeg_create_decompress (&cinfo);            // initialize decompressor
    jpeg_stdio_src (&cinfo, fd);                // specify the data source
    jpeg_read_header (&cinfo, FALSE);           // read the file parameters
    jpeg_start_decompress (&cinfo);             // start working

    spec.x = 0;
    spec.y = 0;
    spec.width = cinfo.output_width;
    spec.height = cinfo.output_height;
    spec.nchannels = cinfo.output_components;
    spec.depth = 0;
    spec.full_width = spec.width;
    spec.full_height = spec.height;
    spec.full_depth = 0;
    spec.format = PT_UINT8;
    spec.tile_width = 0;
    spec.tile_height = 0;
    spec.tile_depth = 0;

    spec.channelnames.clear();
    switch (spec.nchannels) {
    case 1:
        spec.channelnames.push_back("a");
        break;
    case 3:
        spec.channelnames.push_back("r");
        spec.channelnames.push_back("g");
        spec.channelnames.push_back("b");
        break;
    case 4:
        spec.channelnames.push_back("r");
        spec.channelnames.push_back("g");
        spec.channelnames.push_back("b");
        spec.channelnames.push_back("a");
        break;
    default:
        fclose (fd);
        return false;
    }
    newspec = spec;
    return true;
}



bool
JpgInput::read_native_scanline (int y, int z, void *data)
{
    assert (y == (int)cinfo.output_scanline);
    assert (y < (int)cinfo.output_height);
    jpeg_read_scanlines (&cinfo, (JSAMPLE **)&data, 1); // read one scanline

    return true;
}


bool
JpgInput::close ()
{
    jpeg_finish_decompress (&cinfo);
    jpeg_destroy_decompress(&cinfo);
    fclose (fd);
```

```
    return true;
}
```

13.7 Using ImageIO plugins to handle pixels directly

NEW!

In an application that has an embedded Gelato library directly linked in, you may want the application to capture pixels "directly" rather than loading an ImageIO DSO/DLL that writes the pixels to a file or sends them to an `iv` window. For example, you may want those pixels drawn in a window of the application, or used as data for further computation. The steps to perform this are straightforward:

1. Design an `ImageOutput` subclass with minimal functionality, that will be passed completed pixels from the renderer. You can do whatever you want with those pixel values — copy them around in memory, display them in a window, pass them on to compositing functionality, etc. Essentially, this `ImageOutput` serves as a "callback" mechanism for your application to be passed the completed pixels.
2. Create your `ImageOutput` in the application.
3. When calling `GelatoAPI::Output()`, pass the *address* of your `ImageOutput` (expressed as an ASCII hexidecimal number) as the *format* parameter (see Section 2.3). For example:

```
ImageOutput *myimageoutput = new MyImageOutput (...);
char addr[20];
sprintf (addr, "%#lx", myimageoutput);
r->Output (dummy_filename, addr, "rgba", "maincamera");
```

As a full example, consider the following code that renders a simple image (which can be found in the Gelato distribution's `examples/imageio-callback` directory):

```
// C++ API example showing the use of a custom ImageOutput to directly
// capture renderer output.
//
// Copyright 2006 NVIDIA Corporation.


#include <stdio.h>
#include <stdlib.h>
#include "gelatoapi.h"
#include "imageio.h"

static Gelato::ErrorManager *err = NULL;

class MyImageOutput : public Gelato::ImageOutput {
 public:
    MyImageOutput() {}
    bool supports_tiles (void) const { return true; }
    bool supports_random_access (void) const { return true; }
    bool supports_volumes (void) const { return true; }
    bool supports_rewrite (void) const { return true; }
    bool supports_empty (void) const { return false; }
```

```
    bool supports_rectangles (void) const { return false; }

    bool open (const char *name, const Gelato::ImageIOFormatSpec &spec,
               int nparams, const Gelato::ImageIOParameter *param,
               bool append=false) {
        this->spec = spec;
        return true;
    }
    bool close () { return true; }

    // write_scanline and write_tile both just call write_rectangle in the
    // obvious way.  All the smarts are in write_rectangle, but of course
    // you could implement something special for write_scanline or write_tile
    // if you prefer.
    bool write_scanline (int y, int z, const float *data, int xstride) {
        return write_rectangle (0, spec.width, y, y, z, z, data,
                                xstride, 0, 0);
    }
    bool write_tile (int x, int y, int z, const float *data, int xstride,
                     int ystride,int zstride) {
        return write_rectangle (x, x + spec.tile_width - 1,
                                y, y + spec.tile_height - 1,
                                z, z + spec.tile_depth - 1,
                                data, xstride, ystride, zstride);
    }

    // Here's the ultimate call.  For this example, we don't do anything
    // but use the Gelato error handler to print the coordinates.  But
    // this is where you'd do fancy things with the pixel values.
    bool write_rectangle (int xmin, int xmax, int ymin, int ymax,
                          int zmin, int zmax, const float *data,
                          int xstride, int ystride, int zstride) {
        err->Message ("received pixels %3i-%3i, %3i-%3i, %3i-%3i %#lx",
                      xmin, xmax, ymin, ymax, zmin, zmax, data);
        return true;
    }
private:
    Gelato::ImageIOFormatSpec spec;
};



int
main (int argc, char **argv)
{
    // Create a renderer object
    err = Gelato::ErrorManager::Create ();
    GelatoAPI *r = GelatoAPI::CreateRenderer (NULL, err);

    // Check if we created the renderer okay.
    if (r == NULL) {
        fprintf (stderr, "Couldn't create renderer! Exiting...\n");
        exit (1);
    }

    // Create a MyImageOutput
    MyImageOutput *myimageio = new MyImageOutput;

    // The way we pass our own ImageOutput is to sprintf its address
    // to a string, in hexidecimal, using the "%#p" format.
    char ptrstring[20]; // make sure it is big enough for a 64-bit pointer
    sprintf (ptrstring, "%#p", (void *)myimageio);
    err->Message ("Output (%s, ...)", ptrstring);
```

```
    r->Output ("blah", ptrstring, "rgb", "camera");

    r->World ();    // Signal the end of the camera section

    // Just make a trivial scene -- a sphere
    r->Translate (0, 0, 4);
    r->Sphere (1.0f, -1.0f, 1.0f, 360.0f);

    r->Render ();

    delete myimageio;
    delete r;
}
```

14 Calling C++ functions from Shaders

There are times when your shaders require functionality that is not available from the built-in library functions (Section 5.8), and is awkward or inefficient to write as a user-written function (Section 5.5.6). For this reason, there is a mechanism by which a function may be written in C++, compiled into DSO's/DLL's, and called from shaders. These routines are sometimes called "DSO Shadeops."

DSO Shadeop support is a Gelato Pro-only feature and is not available in basic Gelato.

Gelato Pro

DSO shadeops are best used to add functionality that could not be performed at all in ordinary shaders. Examples include: file I/O (other than reading texture files), building complex data structures such as large tables or spatial search trees, or accessing OS functionality such as pipes or resource management.

The developer compiles these routines into a DSO. When compiling the shader, the function may be called in the same manner as any built-in or user-defined shader function. When the shader compiler, `gslc`, encounters a call for a function that is not defined, it will search all directories specified by the `-I` switch, looking for DSO's. Any DSO's that are found will be searched for the appropriately-named declaration table (see below), and if found, `gslc` will understand that the call is to a DSO Shadeop.

For rendering, the DSO must be compiled and placed in one of the directories that contain compiled shaders (see the search path options described in Section 4.3.6). The DSO will be loaded only as needed at render time.

14.1 Calling Conventions

This section will explain all the pertinent details to allow you to write C++ functions that may be called from shaders. For illustrative purposes, we will use an example of a `cube` function that returns the cube of its argument (for "triples," it returns a component-by-component cube of its argument).

14.1.1 Header file and Namespace

DSO shadeops must include the `shadeop.h` header file:

```
#include "shadeop.h"
```

The contents of this header file are contained within the `Gelato::` namespace. The easiest thing to do is to tell the C++ compiler to automatically resolve symbols in this namespace:

```
using namespace Gelato;
```

Of course, if you also need to use libraries from some other source that have unqualified names in common with the Gelato libraries, you may wish to forgo the `using` directive and explicitly prepend `Gelato::` onto all references to symbols defined in the header. For the remainder of this section, we will assume that you are employing the `using namespace Gelato` directive.

14.1.2 C linkage

The declaration table and shadeop implementation must be declared with C linkage. That is, those two declarations must reside within the following structure:

```
extern "C" {
   ...
};
```

14.1.3 Declaration Table

The DSO shadeop must include a *declaration table*, which is an array of `char *`'s giving declarations for each of the polymorphic versions of the function, terminated by a `NULL`. The table must be named *op*`_shadeop`, where *op* is the name of the function we are implementing.

Each declaration is a C-like function prototype giving the return type, implementation function name, and comma-separated list of types for each parameter of the function. The keyword `output` *must* be used to designate parameters that the DSO shadeop may modify. As examples,

```
"float foo (float)"
"vector bar (float, point, string)"
"void procedure (float, output float)"
```

The implementation function is the name of your C++ function that implements the shadeop, and does not need to be the same as the name of the shadeop. Furthermore, you can have different polymorphic versions call different implementations.

For example, to implement a function called `cube`, the declaration table might look like this:

```
GELATO_EXPORT const char *cube_shadeop[] = {
      "float cube (float)",
      "color cube3 (color)",
      "point cube3 (point)",
      "vector cube3 (vector)",
      "normal cube3 (normal)",
      NULL  /* no more variants */
};
```

The above example indicates that there are five ways that the cube function may be called, each with one argument (of different types), and that each one returns a result whose type is the same as the type of the argument. The version that takes (and returns) a float is implemented by a C++ function called cube, whereas the versions that involve "triples" are implemented by a C++ function called cube3.

Note that the GELATO_EXPORT symbol is defined in shadeop.h and correctly handles OS-specific declarations of which routines must be visible to the renderer. On Windows, in particular, the GELATO_EXPORT macro (which actually expands to __declspec(dllexport)) is required, or the renderer will not be able to correctly reference the DSO routines.

14.1.4 Implementation

DSO shadeop implementations must have the following prototype:

```
GELATO_EXPORT void impl (ShadingExecution *exec, int nargs, const ShaderArg *args)
{...}
```

where

- exec is a pointer to the execution context of the shader.
- nargs is the number of arguments to the shadeop, including the result (which, if it exists, is argument 0).
- args[0..nargs-1] are handles for variables holding the arguments to the shadeop.

14.1.5 Accessing arguments

Handles for the arguments to the shadeop may be found in args[0..nargs-1]. There are several methods of the ShadingExecution that can return useful information about individual arguments:

`ParamType type (ShaderArg arg)`

Returns the type of the shading variable, as a ParamType.

`bool isvarying (ShaderArg arg)`

Returns true if shading variable is currently varying (having separate values for each point being simultaneously shaded) or false if the variable is uniform (having one shared value for all points being shaded).

`void *data (ShaderArg arg, int gridindex=0)`

Returns a pointer to the beginning of storage for the variable, for the particular grid point. You should cast it as required. If the variable is uniform, its single value is pointed to by the return value. If the variable is varying, then the pointer is to the beginning of the storage of that variable for the particular grid index given.

`int stride (ShaderArg arg)`

Returns the stride (in number of `float`s or number of `char*`'s) between successive grid points' data for the indexed variable. For uniform variables, `stride()` returns 0.

`bool forcevarying (ShaderArg arg)`

Forces the shading variable to be varying (able to take on a different value at every point), if at all possible. Returns `true` if the shading variable was successfully made varying (or already was varying), `false` if it was not possible. The shading system will already have made all of your shadeop's output variables (and the result) varying if any of the input arguments were varying. But occasionally you may write a shadeop that must always return a varying value, even if all of its inputs happen to be uniform, and in those cases, `forcevarying()` is the way to achieve this.

14.1.6 Iterating over the grid the easy way

For efficiency, Gelato will attempt to shade many points at once (this collection of points is sometimes called a *grid*). Your shadeop should perform its operations on all these points. However, because of loops or conditionals in the shader, not all points in the grid may be "active."

The `shadeop.h` file defines a helper class, `ShadeGridIter`, that helps you to iterate over all the points in a grid that are turned on, and correctly iterates only once for the uniform case. It has the following useful methods:

`ShadeGridIter iter (ShadingExecution *exec, bool vary)`

Initializes an iterator for the given shading execution. The value of `vary` should be `true` if you intend to compute a different result for each point in the grid, which is generally true if you are storing the result in a varying variable.

`int iter()`

The `()` operator (that is, calling the iterator as if it were a function) returns the index of the current grid point that you are executing.

`iter++`

The `++` operator advances the iterator to the next grid point that is active.

`bool iter.done()`

The `done()` method returns `true` if there are no more active grid points left to run.

Within any particular iteration on the grid, the data for any shader argument (say, `ShaderArg arg`) may be accessed using

```
exec->data (arg, iter())
```

or, as convenient shorthand,

```
arg.data(iter())
```

Thus, putting it all together, here is how you would compute the cube of all active grid points (for the `float` version of the `cube` function):

```
ShaderArg result = args[0];           // Result is in argument 0
ShaderArg val = args[1];              // Passed-in value is argument 1
int rvary = exec->isvarying (result); // Check whether result is varying

// Assign float to float
for (ShadeGridIter i (exec, rvary); !i.done(); ++i) {
    float *r = (float *) result.data(i);
    float *v = (float *) val.data(i);
    r[0] = v[0]*v[0]*v[0];
}
```

14.1.7 Iterating the hard way

For those people (you know who you are) who really have to get their hands dirty with all the gory details, you can bypass the `ShadeGridIter` and run through all the grid points yourself. This is accomplished by using additional `ShadingExecution` methods `begin()` and `end()`, which return the indices of the first active grid point and one past the last active grid point, respectively, and `active()`, which returns a pointer to the array of runflags (which, for each grid point, is nonzero if the grid point is active).

If you feel the need, you may also keep track of the pointers to the variables yourself. Here's the result (which is essentially how `ShadeGridIter` is implemented):

```
ShaderArg result = args[0];           // Result is in argument 0
ShaderArg val = args[1];              // Passed-in value is argument 1
int rvary = exec->isvarying (result); // Check whether result is varying
float *r = exec->data (result);
float *v = exec->data (val);
const ShadeRunFlags *active = exec->active();

if (rvary) {
    int rstride = exec->stride (result);
    int vstride = exec->stride (val);
    int first = exec->begin();
    int end = exec->end();
    r += first * rstride;
    v += first * vstride;
    for (int i = first;  i != end;  ++i) {
        if (active[i])
            r[0] = v[0]*v[0]*v[0];
        r += rstride;
        v += vstride;
    }
} else {
    // Uniform case -- just do one operation
    r[0] = v[0]*v[0]*v[0];
}
```

We don't really recommend doing it this way, but every once in a while you may need to do something nonstandard or squeeze out that extra little bit of performance.

14.1.8 Strings

Strings are stored as `const char *`'s. Thus, for a string parameter to a DSO shadeop, you may simply:

```
ShaderArg str = args[...];    // String parameter
for (ShadeGridIter i (exec, rvary); !i.done(); ++i) {
    ...
    const char **s = (const char **) str.data(i);
    printf ("str value was '%s'\n", s[0]);
}
```

However, if you ever *store* a string (i.e., your DSO shadeop returns a string value, or has an `output` parameter that writes a string), you need to *tokenize* the string. Tokenizing converts your string into a unique representation in a special table in the renderer, allowing for minimal storage and very fast comparison. This can be done with the `ShadingExecution::tokenize` function:

```
ShaderArg str = args[0];    // Return value is a string
for (ShadeGridIter i (exec, rvary); !i.done(); ++i) {
    const char **s = (const char **) str.data(i);
    char mystring[100];
    sprintf (mystring, ...);
    s[0] = exec->tokenize (mystring);
}
```

If you alter the string pointed to by a shader argument, or assign your own (non-tokenized) `char *`, you may crash the renderer or get strange results. It is very important that you create new renderer strings using the `tokenize` method.

14.1.9 Thread Safety

Gelato is multi-threaded, which means that multiple grids can be shaded simultaneously by different threads.

Some operations that you may do in a DSO shadeop might fail under such conditions, for example, having shared or `static` data, building data structures that must persist between different calls to the shadeop, or inializations that must happen exactly once (such as filling in a table).

For this reason, by default, Gelato assumes that your DSO shadeop is not necessarily thread-safe, and therefore it will ensure that no two shadeops will be called simultaneously (using a lock called a "mutex").

However, if you are careful to write your shadeop so that it is fully thread-safe[1], there is a way to signal Gelato that so that it will *not* perform the locking, therefore the shadeop may

[1]Please refer to the documentation for your operating system, or a reference of your choice on multithreading, for further details.

execute simultaneously on different grids (and simultaneously with other DSO shadeops that are marked as thread-safe).

To indicate that a shadeop as thread-safe, just declare an `int` variable called *name*`_threadsafe` (where *name* is the name of the shadeop) whose value is nonzero. For example, to declare the `cube` shadeop described above as thread-safe:

```
GELATO_EXPORT int cube_threadsafe = 1;
```

If no *name*`_threadsafe` variable is found in the DSO/SLL, or if the value of the integer is zero, the shadeop is assumed to not be thread-safe, and so the mutex locking will be used to ensure that it does not execute simultaneously with itself or any other non-thread-safe shadeops.

14.1.10 Protocol Check

The header `shadeop.h` defines a value `SHADEOP_PROTOCOL`. If a renderer has a symbol named `shadeop_protocol`, assumed to hold an `int` value, and that value does not match the renderer's protocol, then the DSO will be rejected by the renderer. This protocol version will be incremented any time the number or type of arguments to shadeop implementations changes (although it is anticipated that this will happen rarely, if ever).

Thus, to achieve this checking, your C++ file should contain:

```
GELATO_EXPORT int shadeop_protocol = SHADEOP_PROTOCOL;
```

14.2 Compiling

By convention, we name the file containing source code for the DSO shadeop *op*`.cpp`, and the compiled DSO *op*`.so`.

For users running Linux and compiling with `gcc`, here is the command to turn your C++ file into a DSO:

```
g++ -shared -o cube.so -I$GELATOHOME/include cube.cpp
```

It is okay to put multiple DSO shadeops in the same `.so` file. Just be careful that each shadeop has its own, properly named, declaration table.

14.3 Basic Example

As an illustrative example, below is the full source code for the implementation of the cube function for both `float` and triple values. The source code below would ordinarily be stored in `cube.cpp` and compiled into the file `cube.so` (on Unix or Linux) or `cube.dll` (on Windows).

```
#include "shadeop.h"
using namespace Gelato;

extern "C" {

GELATO_EXPORT int shadeop_protocol = SHADEOP_PROTOCOL;
```

```
GELATO_EXPORT const char *cube_shadeop[] = {
    "float cube (float)",
    "color cube3 (color)",
    "point cube3 (point)",
    "vector cube3 (vector)",
    "normal cube3 (normal)",
    NULL  /* no more variants */
};


GELATO_EXPORT void cube (ShadingExecution *exec, int nargs, ShaderArg *args)
{
    ShaderArg result = args[0];           // Result is in argument 0
    ShaderArg val = args[1];              // Passed-in value is argument 1
    int rvary = exec->isvarying (result); // Check whether result is varying

    // Assign float to float
    for (ShadeGridIter i (exec, rvary); !i.done(); ++i) {
        float *r = (float *) result.data(i);
        float *v = (float *) val.data(i);
        r[0] = v[0]*v[0]*v[0];
    }
}


GELATO_EXPORT void cube3 (ShadingExecution *exec, int nargs, ShaderArg *args)
{
    ShaderArg result = args[0];           // Result is in argument 0
    ShaderArg val = args[1];              // Passed-in value is argument 1
    int rvary = exec->isvarying (result); // Check whether result is varying

    // Assign triple to triple, e.g., color = cube(color)
    for (ShadeGridIter i (exec, rvary); !i.done(); ++i) {
        float *r = (float *) result.data(i);
        float *v = (float *) val.data(i);
        r[0] = v[0]*v[0]*v[0];
        r[1] = v[1]*v[1]*v[1];
        r[2] = v[2]*v[2]*v[2];
    }
}

}; /* extern "C" */
```

14.4 Texture, Noise, Attributes, and Symbols

Gelato allows DSO shadeops to read from texture, shadow, and environment maps; to call noise routines, to retrieve shader symbols, and to get renderer attributes. This section documents the methods of the `ShadingExecution` that enable this access.

14.4.1 Accessing Texture

A DSO shadeop may access texture, shadow, and environment maps using the following methods of the `ShadingExecution` class:

```
void texture (const char *filename, int firstchannel, int nchannels,
              float s, float t, float dsdx, float dtdx,
              float dsdy, float dtdy, float *result,
              int nparams, const char *paramname[], void *paramdata[]);
void shadow (const char *filename, int firstchannel, int nchannels,
             const float *P, const float *dPdx,
             const float *dPdy, float *result,
             int nparams, const char *paramname[], void *paramdata[]);
void environment (const char *filename, int firstchannel, int nchannels,
                  const float *R, const float *dRdx,
                  const float *dRdy, float *result,
                  int nparams, const char *paramname[], void *paramdata[]);
```

For all three of these functions, most of the parameters are used in nearly identical ways to the corresponding GSL functions (see Section 5.8.10). The *firstchannel* parameter indicates the first channel of the file to read (corresponding to the optional `"firstchannel"` parameter to the GSL versions of the functions). The *nchannels* indicates how many channels to read (1 for `float`, 3 for `color`, etc.).

Optional parameters may be passed to the routines, just like in GSL. In this case, *paramname*[0..*nparams*-1] contain the parameter names, and *paramdata*[0..*nparams*-1] contain pointers to the data for each parameter. If no optional parameters are required, you should pass *nparams*=0, *paramname*=NULL, *paramdata*=NULL.

```
bool gettextureinfo (const char *texturename, const char *parmaname,
                     Gelato::ParamType type, void *result);
```

Retrieves information about the given texture file. If the texture exists and has a parameter of the given name and type, its value will be stored in the memory pointed to by *result* and the function will return `true`. If the file does not exist, or no such parameter is found, or if the parameter does not have the type stipulated by the caller, no value will be stored and the function will return `false`. It is the responsibility of the caller to ensure that *result* points to a large enough buffer to hold the data type requested. Valid parameters and types may be found in Table 5.5.

Below is an example of a DSO shadeop that performs an unfiltered shadow lookup:

```
GELATO_EXPORT void
dso (ShadingExecution *exec, int nargs, ShaderArg *args)
{
    ShaderArg result = args[0];
    ShaderArg Psym = args[1];
    int rvary = exec->isvarying (result); // Check whether result is varying
```

```
    float bias = 0.1;
    char *paramname[1] = {"bias"};
    void *paramdata[1] = {&bias};
    float zerovec[3] = {0,0,0};
    for (ShadeGridIter i (exec, rvary); !i.done(); ++i) {
        float *r = (float *) result.data(i);
        float *p = (float *) Psym.data(i);

        exec->shadow ("foo.sm", 0, 1, p, zerovec, zerovec,
                      r, 1, paramname, paramdata);
    }

    float res[2];
    exec->gettextureinfo ("foo.sm", "resolution",
                          Gelato::ParamType(PT_FLOAT,2), &res);
    if (res[0] < 1024 || res[0] < 1024)
        printf ("Warning: you really should use a bigger shadow map\n");
}
```

14.4.2 Noise Functions

You may freely use Gelato's noise functions from within DSO shadeops. The declarations are all in `noise.h`, and are within the namespace `Gelato::Noise`.

```
float noise (float x);
float noise (float x, float y);
float noise (float x, float y, float z);
float noise (float x, float y, float z, float t);

float snoise (float x);
float snoise (float x, float y);
float snoise (float x, float y, float z);
float snoise (float x, float y, float z, float t);

float cnoise (float x);
float cnoise (float x, float y);
float cnoise (float x, float y, float z);
float cnoise (float x, float y, float z, float t);

float pnoise (float x, float xper);
float pnoise (float x, float y, float xper, float yper);
float pnoise (float x, float y, float z, float xper, float yper, float zper);
float pnoise (float x, float y, float z, float t,
              float xper, float yper, float zper, float tper);

void noise (int indim, float *in, int outdim, float *out);
```

```
void snoise (int indim, float *in, int outdim, float *out);
void cnoise (int indim, float *in, int outdim, float *out);
void pnoise (int indim, float *in, float *per, int outdim, float *out);
void psnoise (int indim, float *in, float *per, int outdim, float *out);
```

14.4.3 Shader symbols

In addition to the parameters passed to the DSO shadeop itself, it is also possible for a DSO shadeop to access any shader parameter or "global" symbols (such as `P`, `u`, etc.):

```
bool getsymbol (const char *name, ShaderArg &symbol);
```

> Search for the named symbol of the shader. If found as a global or parameter, store a ShaderArg-like handle in *symbol* and return `true`. If no shader symbol with the given *name* is found, return `false` and do not modify *symbol*.

Shader symbols accessed in this manner are considered read-only — it would generally not be wise to attempt to write new data into them. The `getsymbol` function will only find globals and shader parameters, not local variables of the shader. Globals will be found only if they are actually in the symbol table of the shader; that is, they must be referenced in the shader itself. If a shader does not reference ˇin any way, for example, a DSO shadeop that the shader calls will fail to find symbol `"u"` using `getsymbol`.

14.4.4 Attributes

A DSO shadeop may inquire the values of any renderer attribute, either global or per-object (for the object being shaded) using the following method of `ShadingExecution`:

```
bool getattribute (const char *name, Gelato::ParamType type, void *result);
```

> If there is an attribute of the given name and type, its value will be stored in the memory pointed to by *result* and the function will return `true`. If no such attribute is found, or if the attribute does not have the type stipulated by the caller, no value will be stored and the function will return `false`. It is the responsibility of the caller to ensure that *result* points to a large enough buffer to hold the data type requested.

14.4.5 Other Information

```
int raylevel (void) const;
```

> Returns the recursion level of the grid being shaded — 0 indicates a grid directly visible to the camera, 1 a reflection, 2 a reflection of a reflection, and so on.
>
> Note that shadow rays have the same ray level as the grid that spawned them. For example, the ray-traced shadows of a camera grid have raylevel 0, but `isshadowray()` will return true.

`bool` **isshadowray** `(void) const;`

Returns true if the shading being performed is to determine the opacity of a shadow ray.

`bool` **isindirectray** `(void) const;`

Returns true if the shading being performed is to determine the color of an indirect ray.

15 Examining compiled shaders with `libgsoargs`

Some applications may need to find out the names, types, and default values of the parameters of compiled shaders. The `libgsoargs` library allows C++ applications to query these data from compiled Gelato shaders. If you are not developing such applications, feel free to skip this chapter.

15.1 The `GsoArgs` class

The header file `gsoargs.h` provides a definition for the C++ class `GsoArgs`, which defines the following public member functions:

`bool GsoArgs::`**open** `(const char *`*shadername*`, const char *`*shaderpath*`=NULL);`

> The `open()` method takes the name of the shader to read, and optionally a colon-separated (or semicolon-separated) list of directories in which to search for the named shader. The searchpath will undergo environment variable substitition; that is, `$`*V*`AR`, `${`*V*`AR}`, `$(`*V*`AR)`, and `%`*V*`AR%` are replaced by the value of environment variable `VAR`, if it exists (for any environment variable). If no searchpath is supplied (i.e., `NULL` is passed), the searchpath will be `"$GELATOHOME/shaders"`.
>
> The `open` method returns `true` upon success. Upon failure, it returns `false` and sets an appropriate error message which may be retrieved by the `error()` method.

`const char * GsoArgs::`**shadertype** `( ) const;`

> Returns the type of shader that was read (one of `"surface"`, `"displacement"`, `"light"`, `"volume"`, `"shader"`).

`const char * GsoArgs::`**shadername** `( ) const;`

> Returns the name of the shader that was read.

`int GsoArgs::`**nargs** `( ) const;`

> Returns the number of parameters that the shader accepts.

const Parameter * GsoArgs::**getarg** (const char **name*) const;
const Parameter * GsoArgs::**getarg** (int *index*) const;

Return a pointer to a GsoArgs::Parameter record for one particular shader parameter, either looked up by name or by numeric index (order declared in the shader). If the index was out of range, or if no parameter is found with a matching name, getarg() returns NULL.

The Parameter structure consists of the following, mostly self-explanatory, definition:

```
struct Parameter {
    const char *name;                     // name
    ParamType type;                       // data type
    bool isoutput;                        // true if it's an output param
    bool valid;                           // false if there's no default val
    bool varlenarray;                     // is it a varying-length array?
    std::vector<float> fdefault;          // default float values
    std::vector<const char *> sdefault;   // default string values
    std::vector<const char *> spacename;  // space name for matrices and
                                          //   triples, for each array elem.
    std::vector<Parameter> metadata;      // Meta-data about the param
};
```

The type field is the same ParamType class as is used by Parameter and Attribute, and is defined in the paramtype.h header file (see Section A.2).

NEW!

For an array parameter (type.isarray is true), if the varlenarray field is true, it is a *variable length array*. In this case, the type.arraylen field describes the default length of the array, which can be overridden at runtime (as described in Section 5.3). If the type.arraylen field is false, then the array length may not be overridden.

The valid field of Parameter indicates whether or not a valid default value is present. There is a good chance at discerning the default value if it was a simple assignment (constant numeric value, possibly in a named space). But if the shader's default value for that parameter is the result of a computation or requires information not available at compile time, valid will be false.

The default values of the parameter are stored in the STL vector fdefault (if it is a type comprised of floats), or sdefault (if it is a string or array of strings). If the parameter is a multi-float type (such as a point), the floating point values will simply be concatenated in the fdefault vector. If the parameter is an array, the values for the array elements will be concatenated in the appropriate default vector.

The metadata field is a vector containing the metadata for the parameter. Each metadatum is itself a Parameter record, since it must have a name, type and value(s). If the metadata vector is empty (that is, metadata.size() == 0), no metadata are associated with this parameter.

const std::vector<Parameter> & **metadata** (void) const;

The metadata() method returns a reference to a vector containing the metadata for the shader as a whole (the global metadata, not the parameter-specific metadata which is

stored with each invididual parameter). Each metadatum is itself a `Parameter` record, since it must have a name, type and value(s). If the vector referred to by `metadata()` is empty (that is, `metadata().size() == 0`), no global metadata are associated with the shader.

const char * GsoArgs::**error** ( );

Return the explanation of any error that has occurred since the last call to `error()` as a text string, or NULL if no error has occurred since the last call to `error()`. Note that the string returned belongs to the class, not to the user — do not free or write to the error message string.

15.2 Using gsoargs.h and libgsoargs

Following are the basic steps to interrogate the parameters of a compiled shader:

1. Your program should be sure to include the file `gsoargs.h`:

   ```
   #include "gsoargs.h"
   ```

2. The contents of `gsoargs.h` are contained within the `Gelato::` namespace. The easiest thing to do is to tell the C++ compiler to automatically resolve symbols in this namespace:

   ```
   using namespace Gelato;
   ```

   Of course, if you also need to use libraries from some other source that have unqualified names in common with the Gelato libraries, you may wish to forgo the `using` directive and explicitly prepend `Gelato::` onto all references to symbols defined in the header. For the remainder of this section, we will assume that you are employing the `using namespace Gelato` directive.

3. Create a `GsoArgs` object, and instruct it to open the shader file.

   ```
   GsoArgs argparser;
   char *shadername = "plastic";   // name of your choice
   bool ok = argparser.open (shadername);
   if (! ok)
       exit(1);  // or a better error recovery scheme
   ```

4. You can check the name of the shader and its type as follows:

   ```
   const char *sname = argparser.shadername();
   const char *stype = argparser.shadertype();
   ```

5. You can check the existance of, and retrieve shader metadata using the `metadata()` method:

```
int nmeta = argparser.metadata().size();
for (int i = 0;  i < nmeta;  ++i) {
    const GsoArgs::Parameter &p = argparser.metadata()[i];
    // Info about the one metadatum is now in p
}
```

6. You can determine the total number of shader parameters using the `nargs()` method:

```
int n = argparser.nargs();
```

   For each argument, you can retrieve information about it using the `getarg()` methods, which returns a pointer to a `GsoArgs::Parameter`. You can access by name,

```
const GsoArgs::Parameter *param;
param = argparser.getarg ("Kd");
```

   or by argument number (between 0 and `nargs()-1`),

```
param = argparser.getarg (3);
```

   A `GsoArgs::Parameter` contains information about that parameter: its name, its type (as a `ParamType` structure), its default value, and (where applicable) the name of the space of the default value (e.g., `"shader"`).

7. The default values of the parameter are stored in the STL vector `fdefault` (if it is a type comprised of floats), or `sdefault` (if it is a string or array of strings). If the parameter is an array, the values are simply concatenated in the appropriate default array.

```
if (param->type.basetype == PT_COLOR) {
    printf ("color = ");
    for (unsigned int a = 0;  a < param->type.arraylen;  ++a)
        printf ("%g %g %g ", param->fdefault[3*a],
                param->fdefault[3*a+1], param->fdefault[3*a+2]);
    printf ("\n");
} else if (param->type.basetype == PT_STRING) {
    printf ("string = ");
    for (unsigned int a = 0;  a < param->type.arraylen;  ++a)
        printf ("string = '%s' ", param->sdefault[a]);
    printf ("\n");
}
```

8. Any metadata about an individual parameter may be found in the parameter's `metadata` field, which is itself a vector of `Parameter` records (one for each metadatum).

9. When the `GsoArgs` object (`argparser` in our example above) is destroyed or exists the scope, it will free all resources that it had allocated. There is no cleanup that the user is required to do.

15.3 Example: gsoinfo source code

As an illustrative example of the use of `libgsoargs`, we present the full source code of the `gsoinfo` utility.

Listing 15.1: `gsoinfo.cpp` source code

```
/////////////////////////////////////////////////////////////////////////////
// Copyright 2004 NVIDIA Corporation.  All Rights Reserved.
//
// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions are
// met:
//
// * Redistributions of source code must retain the above copyright
//   notice, this list of conditions and the following disclaimer.
// * Redistributions in binary form must reproduce the above copyright
//   notice, this list of conditions and the following disclaimer in the
//   documentation and/or other materials provided with the distribution.
// * Neither the name of the NVIDIA nor the names of its contributors
//   may be used to endorse or promote products derived from this software
//   without specific prior written permission.
//
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
// "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
// LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
// A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
// OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
// SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
// LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
// DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
// THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
// (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
// OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
//
// (This is the Modified BSD License)
/////////////////////////////////////////////////////////////////////////////


#include <iostream>
#include <string>

#include "gsoargs.h"
using namespace Gelato;



static void
usage (void)
{
    std::cout << "gsoinfo -- list parameters of a compiled Gelato shader\n";
    std::cout << "(c) Copyright 2004 NVIDIA Corp.  All rights reserved.\n";
    std::cout << "Usage:  gsoinfo [options] file0 [file1 ...]\n";
    std::cout << "Options:\n";
    std::cout << "       -v       Verbose\n";
    std::cout << "       -p %s    Set searchpath for shaders\n";
}


static void
```

```
print_default_string_vals (const GsoArgs::Parameter *p, bool verbose)
{
    if (verbose) {
        for (unsigned int a = 0;  a < p->type.arraylen;  ++a)
            std::cout << "\t\tDefault value: \"" << p->sdefault[a] << "\"\n";
    } else {
        for (unsigned int a = 0;  a < p->type.arraylen;  ++a)
            std::cout << "\"" << p->sdefault[a] << "\" ";
        std::cout << "\n";
    }
}



static void
print_default_float_vals (const GsoArgs::Parameter *p, bool verbose)
{
    int nf = ParamBaseTypeNFloats(p->type.basetype);
    for (unsigned int a = 0;  a < p->type.arraylen;  ++a) {
        if (verbose) {
            std::cout << "\t\tDefault value: ";
            if (p->spacename.size() > a && p->spacename[a])
                std::cout << "\"" << p->spacename[a] << "\" ";
        }
        if (nf > 1)
            std::cout << "[ ";
        for (int f = 0;  f < nf; ++f) {
            std::cout << p->fdefault[a*nf+f];
            if (f < nf-1)
                std::cout << ' ';
        }
        if (nf > 1)
            std::cout << " ]";
        std::cout << std::endl;
    }
}



static std::string
elaborate_escape_chars (const char *unescaped)
{
    std::string s = unescaped;
    for (size_t i = 0;  i < s.length();  ++i) {
        char c = s[i];
        if (c == '\n' || c == '\t' || c == '\v' || c == '\b' ||
            c == '\r' || c == '\f' || c == '\a' || c == '\\' || c == '\"') {
            s[i] = '\\';
            ++i;
            switch (c) {
            case '\n' : c = 'n'; break;
            case '\t' : c = 't'; break;
            case '\v' : c = 'v'; break;
            case '\b' : c = 'b'; break;
            case '\r' : c = 'r'; break;
            case '\f' : c = 'f'; break;
            case '\a' : c = 'a'; break;
            }
            s.insert (i, &c, 1);
        }
    }
    return s;
}
```

```
static void
print_metadata (const GsoArgs::Parameter &m)
{
    char typestring[100];
    m.type.tostring (typestring, sizeof(typestring));
    std::cout << "\t\tmetadata: " << typestring << ' ' << m.name << " =";
    for (unsigned int d = 0;  d < m.fdefault.size();  ++d)
        std::cout << " " << m.fdefault[d];
    for (unsigned int d = 0;  d < m.sdefault.size();  ++d)
        std::cout << " \"" << elaborate_escape_chars(m.sdefault[d]) << "\"";
    std::cout << std::endl;
}



static void
gsoinfo (const char *name, const char *path, bool verbose)
{
    GsoArgs g;
    g.open (name, path);
    const char *e;
    if ((e = g.error()) != NULL) {
        std::cout << "ERROR opening shader \"" << name << "\" (" << e << ")\n";
        return;
    }
    if (verbose)
         std::cout << g.shadertype() << " \"" << g.shadername() << "\"\n";
    else std::cout << g.shadertype() << " " << g.shadername() << "\n";
    if (verbose) {
        for (unsigned int m = 0;  m < g.metadata().size();  ++m)
            print_metadata (g.metadata()[m]);
    }

    for (int i = 0;  i < g.nargs();  ++i) {
        const GsoArgs::Parameter *p = g.getarg (i);
        if (!p)
            break;
        char typestring[100];
        p->type.tostring (typestring, sizeof(typestring));
        if (verbose) {
            std::cout << "    \"" << p->name << "\" \""
                      << (p->isoutput ? "output " : "") << typestring << "\"\n";
        } else {
            std::cout << (p->isoutput ? "output " : "") << typestring << ' '
                      << p->name << ' ';
        }
        if (! p->valid) {
            if (verbose)
                 std::cout << "\t\tUnknown default value\n";
            else std::cout << "nodefault\n";
        }
        else if (p->type.basetype == PT_STRING)
             print_default_string_vals (p, verbose);
        else print_default_float_vals (p, verbose);
        if (verbose) {
            for (unsigned int i = 0;  i < p->metadata.size();  ++i)
                print_metadata (p->metadata[i]);
        }
    }
}
```

```
int
main (int argc, const char *argv[])
{
    const char *path = NULL;
    bool verbose = false;
    for (int a = 1;  a < argc;  ++a) {
        if (! strcmp(argv[a],"-") || ! strcmp(argv[a],"-h") ||
            ! strcmp(argv[a],"-help") || ! strcmp(argv[a],"--h") ||
            ! strcmp(argv[a],"--help")) {
            usage();
            exit(0);
        } else if (! strcmp(argv[a], "-p")) {
            if (a == argc-1) {
                usage(); return(-1);
            }
            path = argv[++a];
        } else if (! strcmp (argv[a], "-v")) {
            verbose = true;
        } else {
            gsoinfo (argv[a], path, verbose);
        }
    }
    return 0;
}
```

Part IV

Appendices

A Public API Header Files

This chapter contains code listings of the C++ header files that define Gelato's public API's.

A.1 gelatoapi.h

```
/////////////////////////////////////////////////////////////////////////////
// Copyright 2004 NVIDIA Corporation.  All Rights Reserved.
//
// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions are
// met:
//
// * Redistributions of source code must retain the above copyright
//   notice, this list of conditions and the following disclaimer.
// * Redistributions in binary form must reproduce the above copyright
//   notice, this list of conditions and the following disclaimer in the
//   documentation and/or other materials provided with the distribution.
// * Neither the name of NVIDIA nor the names of its contributors
//   may be used to endorse or promote products derived from this software
//   without specific prior written permission.
//
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
// "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
// LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
// A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
// OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
// SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
// LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
// DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
// THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
// (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
// OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
//
// (This is the Modified BSD License)
/////////////////////////////////////////////////////////////////////////////



// This header file is the definition for the C++ public API of
// NVIDIA's Gelato renderer.
```

```
#ifndef GELATOAPI_H
#define GELATOAPI_H


#include "export.h"
#include "paramtype.h"   /* Needed for ParamType definition */
#include "errormanager.h"



class GELATO_PUBLIC GelatoAPI {
public:
    GelatoAPI (void) {}
    virtual ~GelatoAPI (void) {}

    virtual void Input (const char *filename) {}
    virtual void Input (const char *filename, const float *bound) {}

    virtual void Camera (const char *name) {}
    virtual void Output (const char *name, const char *format,
                         const char *dataname, const char *cameraname) {}
    virtual void World (void) {}
    virtual void Render (const char *cameraname=NULL) {}

    virtual void Motion (int ntimes, float time0, ...) {}
    virtual void Motion (int ntimes, const float *times) {}

    // Inserts a comment
    virtual void Comment (const char *format, ...) {}

    // Executes a command signified by the token command
    virtual void Command (const char *command) {}

    // Set a parameter of the next shader, primitive, camera, or output
    virtual void Parameter (const char *name, Gelato::ParamType t, const void *val) {}
    virtual void Parameter (const char *name, Gelato::ParamType t, int val) {}
    virtual void Parameter (const char *name, Gelato::ParamType t, float val) {}
    virtual void Parameter (const char *name, Gelato::ParamType t, double val) {}
    virtual void Parameter (const char *name, Gelato::ParamType t, const char *val) {}
    virtual void Parameter (const char *name, Gelato::ParamType t, const int *val) {}
    virtual void Parameter (const char *name, Gelato::ParamType t, const float *val) {}
    virtual void Parameter (const char *name, Gelato::ParamType t, const char **val) {}
    // Set a parameter with type info embedded in the name
    virtual void Parameter (const char *typedname, const void *val) {}
    virtual void Parameter (const char *typedname, int val) {}
    virtual void Parameter (const char *typedname, float val) {}
    virtual void Parameter (const char *typedname, double val) {}
    virtual void Parameter (const char *typedname, const char *val) {}
    virtual void Parameter (const char *typedname, const int *val) {}
```

```
    virtual void Parameter (const char *typedname, const float *val) {}
    virtual void Parameter (const char *typedname, const char **val) {}

    // Set an attribute in the graphics state, with name and explicit type
    virtual void Attribute (const char *name, Gelato::ParamType t, const void *val) {}
    virtual void Attribute (const char *name, Gelato::ParamType t, int val) {}
    virtual void Attribute (const char *name, Gelato::ParamType t, float val) {}
    virtual void Attribute (const char *name, Gelato::ParamType t, double val) {}
    virtual void Attribute (const char *name, Gelato::ParamType t, const char *val) {}
    virtual void Attribute (const char *name, Gelato::ParamType t, const int *val) {}
    virtual void Attribute (const char *name, Gelato::ParamType t, const float *val) {}
    virtual void Attribute (const char *name, Gelato::ParamType t, const char **val) {}
    // Set an attribute with type embedded in the name
    virtual void Attribute (const char *typedname, const void *val) {}
    virtual void Attribute (const char *typedname, int val) {}
    virtual void Attribute (const char *typedname, float val) {}
    virtual void Attribute (const char *typedname, double val) {}
    virtual void Attribute (const char *typedname, const char *val) {}
    virtual void Attribute (const char *typedname, const int *val) {}
    virtual void Attribute (const char *typedname, const float *val) {}
    virtual void Attribute (const char *typedname, const char **val) {}

    // Get the value of a graphics state attribute
    virtual bool GetAttribute (const char *name, void *val) { return false; }
    virtual bool GetAttribute (const char *name, int &val) { return false; }
    virtual bool GetAttribute (const char *name, float &val) { return false; }
    virtual bool GetAttribute (const char *name, double &val) { return false; }
    virtual bool GetAttribute (const char *name, char *&val) { return false; }
    virtual bool GetAttribute (const char *name, int *val) { return false; }
    virtual bool GetAttribute (const char *name, float *val) { return false; }
    virtual bool GetAttribute (const char *name, char **val) { return false; }

    virtual void PushAttributes (void) {}
    virtual void PopAttributes (void) {}
    virtual void SaveAttributes (const char *name, const char *attrs=NULL) {}
    virtual void RestoreAttributes(const char *name, const char *attrs=NULL) {}
    virtual void Modify(const char *namepattern=NULL) {}

    // Transformations
    virtual void PushTransform (void) {}
    virtual void PopTransform (void) {}
    virtual void SetTransform (const float *M) {}
    virtual void SetTransform (const char *spacename) {}
    virtual void AppendTransform (const float *M) {}
    virtual void Translate (float x, float y, float z) {}
    virtual void Rotate (float angle, float x, float y, float z) {}
    virtual void Scale (float x, float y, float z) {}

    // Shaders
    virtual void Shader(const char *shaderusage, const char *shadername=NULL,
                        const char *layername=NULL) {}
```

```
virtual void Light (const char *lightid, const char *shadername,
                    const char *layername=NULL) {}
virtual void ShaderGroupBegin (void) {}
virtual void ShaderGroupEnd (void) {}
virtual void ConnectShaders (const char *srclayer, const char *srcparam,
                             const char *dstlayer, const char *dstparam) {}
virtual void LightSwitch (const char *lightid, bool on) {}

// Geometry
// 0-D prims - point clouds
virtual void Points (int npoints) {}

// 1-D prims - lines, curves, hair
virtual void Curves (const char *interp, int ncurves, int nvertspercurve){}
virtual void Curves (int ncurves, int nvertspercurve, int order,
                     const float *knot, float vmin, float vmax) {}

// 2-D prims - rectangular patches (NURBS, bicubics, bilinears), and
// indexed face meshes (polys, polyhedra, subdivs)
virtual void Patch (const char *interp, int nu, int nv) {}
virtual void Patch (int nu, int uorder, const float *uknot,
                    float umin, float umax,
                    int nv, int vorder, const float *vknot,
                    float vmin, float vmax) {}
virtual void TrimCurve (int nloops, const int *ncurves, const int *n,
                        const int *order, const float *knot,
                        const float *min, const float *max,
                        const float *uvw) {}

virtual void Mesh (const char *interp, int nfaces,
                   const int *nverts, const int *verts) {}

virtual void Sphere (float radius, float zmin, float zmax,
                     float thetamax=360) {}
virtual void Sphere (float radius) { Sphere(radius,-radius,radius); }

class GELATO_PUBLIC Generator {
public:
    Generator() {}
    virtual ~Generator() {}
    virtual bool bound (float *bbox) { return false; }
    virtual void run (GelatoAPI *renderer, const char *params) {}
};

// Each generator DSO/DLL should include this statement:
//      GELATO_EXPORT int generator_version = GelatoAPI::API_VERSION;
// Applications using generator DSO/DLL's should check this
// variable, to avoid using DSO/DLL's compiled against
// incompatible versions of this header file.
static const int API_VERSION = 2;
```

```
    virtual void Input (Generator *procedure, const float *boundingbox=NULL) {}

    virtual Gelato::ErrorManager &Err (void) = NULL;

    // DEPRECATED - use ErrorManager instead
    virtual Gelato::ErrCode GetError (const char **errstrbuf=NULL) {
        return Gelato::ERR_NO_ERROR;
    }


    // make a new renderer - using an existing ErrorManger
    // if err is NULL then create a new (default) ErrorManager
    static GelatoAPI *CreateRenderer (const char *type=NULL,
                                  Gelato::ErrorManager *p_err=NULL);

    // because the new/malloc used inside CreateRenderer may be different
    // than the one used by the application calling CreateRenderer, we
    // override delete - this way the correct version of free/delete is used
    void operator delete (void *pMem, size_t size);
};


#endif /* !defined(GELATOAPI_H) */
```

A.2 paramtype.h

```
///////////////////////////////////////////////////////////////////////////
// Copyright 2004 NVIDIA Corporation.  All Rights Reserved.
//
// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions are
// met:
//
// * Redistributions of source code must retain the above copyright
//   notice, this list of conditions and the following disclaimer.
// * Redistributions in binary form must reproduce the above copyright
//   notice, this list of conditions and the following disclaimer in the
//   documentation and/or other materials provided with the distribution.
// * Neither the name of NVIDIA nor the names of its contributors
//   may be used to endorse or promote products derived from this software
//   without specific prior written permission.
//
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
// "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
// LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
// A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
// OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
// SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
// LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
// DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
// THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
// (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
// OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
//
// (This is the Modified BSD License)
///////////////////////////////////////////////////////////////////////////


#ifndef GELATO_PARAMTYPE_H
#define GELATO_PARAMTYPE_H

#ifndef NULL
#define NULL 0
#endif

#include "export.h"

namespace Gelato {


// Base data types
enum ParamBaseType {
    PT_UNKNOWN = 0,
    PT_VOID,
    PT_STRING,
```

```
    PT_FLOAT, PT_HALF, PT_DOUBLE,
    PT_POINT, PT_VECTOR, PT_NORMAL,
    PT_COLOR,
    PT_HPOINT, PT_MATRIX,
    PT_INT8, PT_UINT8, PT_INT16, PT_UINT16, PT_INT, PT_UINT,
    PT_POINTER,
    PT_LAST
};


// Return the name, for printing and whatnot, of a ParamBaseType
extern GELATO_PUBLIC const char *ParamBaseTypeNameString (int t);

// Return the size, in bytes, of a single item of a ParamBaseType
extern GELATO_PUBLIC int ParamBaseTypeSize (int t);

// Return the number of floats comprising a ParamBaseType
// (e.g., 3 for PT_POINT).  Return 0 for all types not comprised of floats.
extern GELATO_PUBLIC int ParamBaseTypeNFloats (int t);



// Interpolation types
enum ParamInterp {
    INTERP_CONSTANT = 0,       // Constant for all pieces/faces
    INTERP_PERPIECE = 1,       // Piecewise constant per piece/face
    INTERP_LINEAR = 2,         // Linearly interpolated across each piece/face
    INTERP_VERTEX = 3          // Interpolated like vertices
};



// ParamType is a simple type descriptor.  Contains a base type, array
// length, and other attributes.  This structure is no bigger than an
// int, and so can be very cheaply passed around.
class GELATO_PUBLIC ParamType {
public:
    ParamType (void) { /* Uninitialized! */ }

    // Construct from base type and interp, or base only (assume non-array)
    ParamType (ParamBaseType base, ParamInterp detail=INTERP_CONSTANT) {
        basetype = base;
        arraylen = 1;
        isarray = 0;
        interp = detail;
        reserved = 0;
    }

    // Construct with array length
    ParamType (ParamBaseType base, short array,
               ParamInterp det=INTERP_CONSTANT) {
```

```
        basetype = base;
        arraylen = array;
        isarray = (array != 0);
        if (! isarray)
            arraylen = 1;
        interp = det;
        reserved = 0;
    }

    // Construct from a string (e.g., "vertex float[3]").  If no valid
    // type could be assembled, set basetype to PT_UNKNOWN.
    ParamType (const char *typestring) {
        if (! fromstring(typestring))
            basetype = PT_UNKNOWN;
    }

    // Set *this to the type described in the string.  Return the
    // length of the part of the string that describes the type.  If
    // no valid type could be assembled, return 0 and do not modify
    // *this.  If shortname is not NULL, store the word(s) in the string
    // after the type (presumably the variable name) in shortname.
    int fromstring (const char *typestring, char *shortname=NULL);

    // Store the string representation of the type in typestring.  Don't
    // overwrite more than maxlen bytes of typestring!  Return true upon
    // success, false upon failure (including failure to fit).
    bool tostring (char *typestring, int maxlen, bool showinterp=false) const;

    // Return size of one element of this type, in bytes
    int datasize (void) const { return arraylen*ParamBaseTypeSize(basetype); }

    int nfloats (void) const { return arraylen*ParamBaseTypeNFloats(basetype); }

    bool operator== (const ParamType &t) {
        return *(int*)(this) == *(int*)(&t);
    }

    // Demote the type to a non-array
    void unarray (void) { isarray = false;  arraylen = 1; }

    unsigned int basetype:5;   // Base type of the data -- one of ParamBaseType
    unsigned int arraylen:18;  // Array len (up to 256k), or 1 if not an array
    unsigned int isarray:1;    // 1 if it's an array
    unsigned int interp:3;     // Sometimes used: interpolation type
    unsigned int reserved:5;   // Future expansion
};


}; /* end namespace Gelato */

#endif /* !defined(GELATO_PARAMTYPE_H) */
```

A.3 errormanager.h

```
/////////////////////////////////////////////////////////////////////////////
// Copyright 2005 NVIDIA Corporation.  All Rights Reserved.
//
// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions are
// met:
//
// * Redistributions of source code must retain the above copyright
//   notice, this list of conditions and the following disclaimer.
// * Redistributions in binary form must reproduce the above copyright
//   notice, this list of conditions and the following disclaimer in the
//   documentation and/or other materials provided with the distribution.
// * Neither the name of NVIDIA nor the names of its contributors
//   may be used to endorse or promote products derived from this software
//   without specific prior written permission.
//
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
// "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
// LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
// A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
// OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
// SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
// LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
// DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
// THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
// (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
// OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
//
// (This is the Modified BSD License)
/////////////////////////////////////////////////////////////////////////////


/////////////////////////////////////////////////////////////////////////////
// ErrorManager (with the local instance pointed to by err) is a low-level
// class for error reporting/logging/etc.  At its basic level, it allows other
// classes to issue error or info messages to be presented to the user.  The
// handling of these messages can be easily changed from sending to the
// console to writing to a file, showing dialog boxes, etc.
//
// There is a local err pointer, through which these functions may be
// called.  For example, a program or library may, upon error, call:
//       err->Warning ("\"%s\" did not exist, creating", file);
// or:
//       err->Error ("Syntax error on line %d", line);
//
// With the default handler, warnings are preceeded by "WARNING: ", errors are
// preceeded by "ERROR: ", Severe messages are preceeded by
// "SEVERE ERROR: ", and info messages are preceded by "INFO:".  There are also
// "Messages" that have no prefix, so can be used for general non-error console
```

```
// output.
//
// ErrorManager performs verbosity filtering and filters and counts
// duplicate Severe/Error/Warning messsages.  To see those collected
// messages and their totals, use err->Summary ().
//
/////////////////////////////////////////////////////////////////////////////


#ifndef GELATO_ERRORMANAGER_H
#define GELATO_ERRORMANAGER_H

#include <stdarg.h>
#include <stdlib.h>

#include "export.h"

namespace Gelato {

enum ErrCode {
    ERR_NO_ERROR=0,   // never sent to handler
    ERR_INFO=512,
    ERR_WARNING=1024,
    ERR_ERROR=2048,
    ERR_SEVERE=4096,
    ERR_MESSAGE=8192, // always shown but not an error
};

enum VerbosityLevel {
    VERBOSITY_QUIET =0, // show MESSAGE, SEVERE, ERROR
    VERBOSITY_NORMAL=1, // also show WARNING
    VERBOSITY_INFO  =2, // also show INFO
};



class GELATO_PUBLIC ErrorHandler {
public:
    virtual ~ErrorHandler () {}
    virtual void operator() (ErrCode errcode, const char *msg) = 0;
};



class GELATO_PUBLIC ErrorManager {
public:
    ~ErrorManager ();

    // passing NULL means use the default handler (stderr)
    static ErrorManager *Create (ErrorHandler *handler=NULL,
                                 int verbosity=VERBOSITY_NORMAL);
```

```
    void Info (const char *format, ...);
    void Warning (const char *format, ...);
    void Error (const char *format, ...);
    void Severe (const char *format, ...);
    void Message (const char *format, ...);

    // change verbosity level
    void Verbosity (int verbosity);

    // return verbosity level
    int Verbosity () const;

    // set a new handler - passing NULL means use the default handler (stderr)
    void Handler (ErrorHandler *handler);

    // get the current new handler - NULL means the default handler (stderr)
    ErrorHandler *Handler () const;

    // return string with summary of Severe/Error/Warning messages
    // it is the caller's responsibility to call free on the returned string
    char *Summary ();

    // because the new/malloc used inside Create may be different
    // than the one used by the application calling Create, we
    // override delete - this way the correct version of free/delete is used
    void operator delete (void *pMem, size_t size);

protected:
    // don't call new, use ErrorManager::Create instead
    ErrorManager ();
};

}; /* end namespace Gelato */

#endif /* !defined(GELATO_ERRORMANAGER_H) */
```

A.4 shadeop.h

```
///////////////////////////////////////////////////////////////////////////
// Copyright 2004 NVIDIA Corporation.  All Rights Reserved.
//
// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions are
// met:
//
// * Redistributions of source code must retain the above copyright
//   notice, this list of conditions and the following disclaimer.
// * Redistributions in binary form must reproduce the above copyright
//   notice, this list of conditions and the following disclaimer in the
//   documentation and/or other materials provided with the distribution.
// * Neither the name of NVIDIA nor the names of its contributors
//   may be used to endorse or promote products derived from this software
//   without specific prior written permission.
//
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
// "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
// LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
// A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
// OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
// SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
// LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
// DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
// THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
// (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
// OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
//
// (This is the Modified BSD License)
///////////////////////////////////////////////////////////////////////////


#ifndef SHADEOP_H
#define SHADEOP_H

#include "export.h"
#include "paramtype.h"   /* Needed for ParamType definition */



namespace Gelato {

class GELATO_PUBLIC ShaderArg;   // forward declaration

typedef short ShaderRunFlag;


// Each shadeop DSO/DLL should include this statement:
//      GELATO_EXPORT int shadeop_version = Gelato::SHADEOP_VERSION;
```

```
// Applications using shadeop DSO/DLL's should check this
// variable, to avoid using DSO/DLL's compiled against
// incompatible versions of this header file.
static const int SHADEOP_VERSION = 1;
#define SHADEOP_TYPE "shadeop"


// Prototype for a DSO/DLL shadeop:
//   void shadeop (ShadingExecution *exec, int nargs, const ShaderArg *args)
//
// exec is a semi-blind pointer to the execution context of the shader.
// nargs is the number of arguments to the shadeop, including the result
//    (which, if it exists, is argument 0).
// args[0..nargs-1] are handles for the arguments to the shadeop.



// The first parameter to a shadeop is a pointer to a ShadingExecution.
// The public interface for this is just an interface class.  Because
// there are no data members or virtual functions, we should be able to
// add additional (non-virtual) helper functions in the future without
// breaking linkage.

class GELATO_PUBLIC ShadingExecution {
public:
    // Return the indices of the first, and one past the last active
    // points in the grid.
    int begin (void);
    int end (void);

    // Return a pointer to the array of runflags for the grid.  Each
    // runflag[i] is nonzero if grid point #i should be executing.
    const ShaderRunFlag *active (void);

    // Return the type of a particular variable of the shading execution.
    ParamType type (ShaderArg arg);

    // Return true if the indexed shading variable is varying.
    bool isvarying (ShaderArg arg);

    // Force a symbol to be varying, if at all possible.  Return true
    // if successful.
    bool forcevarying (ShaderArg arg);

    // Return a pointer to the actual data of a particular variable of the
    // shading execution, for a particular gridpoint.  If no gridindex
    // parameter is supplied, it defaults to point 0, which is just the
    // beginning of the data area for that variable.
    void *data (ShaderArg arg, int gridindex=0);

    // The stride (in number of floats, or number of char*'s) between
```

```
// successive grid points' data for the indexed variable.  For
// uniform variables (that is, one value for the whole grid),
// stride() returns 0.
int stride (ShaderArg arg);

// Tokenize a user-generated string (that is, make it unique and in
// the renderer's token table) so that it can be assigned to a string
// variable.
const char *tokenize (const char *userstring);

// Pass along an error message for the renderer to display, with printf
// conventions.
void Error (const char *format, ...);

// Single 2D texture lookup.
void texture (const char *filename, int firstchannel, int nchannels,
              float s, float t, float dsdx, float dtdx,
              float dsdy, float dtdy, float *result,
              int nparams, const char *paramname[], void *paramdata[]);
// Shadow map lookup
void shadow (const char *filename, int firstchannel, int nchannels,
             const float *P, const float *dPdx,
             const float *dPdy, float *result,
             int nparams, const char *paramname[], void *paramdata[]);
// Environment map lookup
void environment (const char *filename, int firstchannel, int nchannels,
                  const float *R, const float *dRdx,
                  const float *dRdy, float *result,
                  int nparams, const char *paramname[], void *paramdata[]);

// Get info about the named texture.  If found, store it in result and
// return true.  If not found or type doesn't match, return false.
bool gettextureinfo (const char *texturename, const char *paramname,
                     Gelato::ParamType type, void *result);

// Get attribute of the given name and type.  If found, store it
// in result and return true.  If not found or type doesn't match,
// return false.
bool getattribute (const char *name, Gelato::ParamType type, void *result);

// Search for the named symbol of the shader.  If found as a global or
// parameter, store a ShaderArg-like handle in symbol and return true.
// Return false and do not modify symbol if the name is not found.
bool getsymbol (const char *name, ShaderArg &symbol);

// Return the "ray level" for this shading.  Zero means that this is for
// camera visibility.  One is a reflection or shadow.  And so on.
int raylevel (void) const;

// Is this shading being done just to compute opacity for a shadow ray?
bool isshadowray (void) const;
```

```
    // Is this shading being done just to compute the contribution of an
    // indirect ray?
    bool isindirectray (void) const;
};




// A ShadeGridIter is used to iterate through all the active (turned on)
// points in a "grid" (collection of points all shaded at once).
class GELATO_PUBLIC ShadeGridIter {
public:
    // Initialize the shade iter.  It needs a pointer to the
    // ShadingExecution and it needs to know if we vary (run once for
    // each active point) or not (run once for the whole grid).
    ShadeGridIter (ShadingExecution *exec, bool vary) : exec(exec) {
        active = exec->active();
        if (vary) {
            i = exec->begin();
            end = exec->end();
        } else {
            i = 0;
            end = 1;
        }
    }

    // The () operator -- iter() -- returns the index of the currently
    // active grid point.
    int operator() (void) const { return i; }

    // iter++ advances to the next grid point that is active
    void operator++ (void) {
        do {
            ++i;
        } while (i < end && active[i] == 0);
    }

    // iter.done() is true if there are no more grid points left to process
    bool done (void) const { return (i >= end); }

private:
    ShadingExecution *exec;
    const ShaderRunFlag *active;
    int i, end;
    friend class ShaderArg;
};



// ShaderArg is a handle of an argument to a shader.  Its only
```

```
// purpose is to be passed into a DSO shadeop in args[], and be used
// as a parameter to various ShadingExecution methods.  But it can
// also be used as a convenient shorthand: for ShaderArg var and
// ShadeGridIter i, you can point to the current grid point's data as
// var.data(i).
class GELATO_PUBLIC ShaderArg {
public:
    // Point to this shader arg's data for the grid point specified by iter.
    void *data (ShadeGridIter &iter) const {
        return iter.exec->data(*this,iter());
    }
private:
    int index;
    friend class ShadingExecution;
};


};  /* end namespace Gelato */


#endif /* SHADEOP_H */
```

A.5 imageio.h

```
///////////////////////////////////////////////////////////////////////////
// Copyright 2004 NVIDIA Corporation.  All Rights Reserved.
//
// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions are
// met:
//
// * Redistributions of source code must retain the above copyright
//   notice, this list of conditions and the following disclaimer.
// * Redistributions in binary form must reproduce the above copyright
//   notice, this list of conditions and the following disclaimer in the
//   documentation and/or other materials provided with the distribution.
// * Neither the name of NVIDIA nor the names of its contributors
//   may be used to endorse or promote products derived from this software
//   without specific prior written permission.
//
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
// "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
// LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
// A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
// OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
// SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
// LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
// DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
// THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
// (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
// OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
//
// (This is the Modified BSD License)
///////////////////////////////////////////////////////////////////////////


#ifndef GELATO_IMAGEIO_H
#define GELATO_IMAGEIO_H

#include <vector>

#include "export.h"
#include "paramtype.h"   /* Needed for ParamBaseType definition */


namespace Gelato {


// Each imageio DSO/DLL should include this statement:
//      GELATO_EXPORT int imageio_version = Gelato::IMAGEIO_VERSION;
// Applications using imageio DSO/DLL’s should check this
// variable, to avoid using DSO/DLL’s compiled against
// incompatible versions of this header file.
```

```
//
// Version 3 added supports_rectangles() and write_rectangle() to
// ImageOutput, and added stride parameters to the ImageInput read
// routines.
const int IMAGEIO_VERSION = 3;


struct ImageIOFormatSpec {
    int x, y, z;                // image origin (0,0,0)
    int width, height, depth;   // width, height, depth (depth>1 for volume)
    int full_width;             // width of entire image (not just cropwindow)
    int full_height;
    int full_depth;
    int tile_width;             // tile size (0 if tiles are not supported)
    int tile_height;
    int tile_depth;
    int nchannels;              // e.g., 4 for RGBA
    ParamBaseType format;       // format of data in each channel
    std::vector<const char*> channelnames;  // e.g., {"R","G","B","A"}
    char unused[256];           // for future expansion
};



struct GELATO_PUBLIC ImageIOParameter {
    const char *name;
    ParamBaseType type;         // data type
    int nvalues;                // number of elements
    const void *value;          // array of values

    ImageIOParameter () {};
    ImageIOParameter (const char *name, ParamBaseType type,
                      int nvalues, const void *value)
        : name(name), type(type), nvalues(nvalues), value(value) {}
};


class GELATO_PUBLIC ImageOutput {
 public:
    ImageOutput ();
    virtual ~ImageOutput ();

    // Overrride these functions in your derived output class
    // to inform the client which formats are supported

    // Does this format know how to write tiled images?
    virtual bool supports_tiles (void) const { return false; }

    // Does this format know how to write tiles/scanlines in any order?
    // (false means that the client must send scanlines/tiles in image order)
    virtual bool supports_random_access (void) const { return false; }
```

```
// Does this format know how to write multiple subimages in a single file?
virtual bool supports_multiimage (void) const { return false; }

// Does this format know how to write volumetric images?
virtual bool supports_volumes (void) const { return false; }

// Does this format accept the same scanline/tile more than once?
// The main use is for interactive output.  supports_rewrite implies
// that supports_random_access must also be true.
virtual bool supports_rewrite (void) const { return false; }

// Does this format support the passing of a NULL data pointer
// in the write_scanline or write_tile functions to indicate
// that the entire data block is zero?
virtual bool supports_empty (void) const { return false; }

// Does this format plugin support write_rectangle calls giving
// totally arbitrary rectangles of pixels?
virtual bool supports_rectangles (void) const { return false; }


// Open file with given name, with resolution and other format data
// as given in spec.  Additional param[0..nparams] contains additional
// params specific to the format/driver (valid parameters should be
// enumerated in the documentation for the output plugin).  Most
// plugins should understand, at a minimum:
//      "quantize"  int[4]       quantization black, white, min, max
//      "gain"      float        gain multiplier
//      "gamma"     float        gamma correction before quant
//      "dither"    float        dither amount
// Open returns true for success, false for failure.
// Note that it is legal to call open multiple times on the same file
// without a call to close(), if it supports multiimage and the
// append flag is true -- this is interpreted as appending images
// (such as for MIP-maps).
virtual bool open (const char *name, const ImageIOFormatSpec &spec,
    int nparams, const ImageIOParameter *param, bool append=false) = 0;

// Write the scanline that includes pixels (*,y,z).  (z==0 for
// non-volume images.)  The stride value gives the data layout:
// one pixel to the "right" is xstride floats away.
// Return true for success, false for failure.  It is a failure to
// call write_scanline with an out-of-order scanline if this format
// driver does not support random access.
virtual bool write_scanline (int y, int z, const float *data, int xstride)
    { return false; }

// Write the tile with (x,y,z) as the upper left corner.
// (z==0 for non-volume images.)  The three stride values give the
// data layout: one pixel to the "right" is xstride floats away,
```

```
    // one pixel "down" is ystride floats away, one pixel "in" (the
    // next volumetric slice) is zstride floats away.  Return true for
    // success, false for failure.  It is a failure to call write_tile
    // with an out-of-order tile if this format driver does not
    // support random access.
    virtual bool write_tile (int x, int y, int z,
        const float *data, int xstride,int ystride,int zstride) {return false;}

    // Write pixels whose x coords range over xmin..xmax (inclusive),
    // y coords over ymin..ymax, and z coords over zmin...zmax.
    // Stride measures the distance (in floats) between successive
    // pixels in x, y, and z.  Return true for success, false for
    // failure.  It is a failure to call write_rectangle for a format
    // plugin that does not return true for supports_rectangles().
    virtual bool write_rectangle (int xmin, int xmax, int ymin, int ymax,
                                  int zmin, int zmax, const float *data,
                                  int xstride, int ystride, int zstride)
        { return false; }

    // Close an image that we are totally done with.
    virtual bool close () = 0;

    // General message passing between client and image output server
    virtual int send_to_output (const char *format, ...);
    int send_to_client (const char *format, ...);

    // Helper routines to compute quantization and exposure
    static int quantize (float value, int black, int white,
                         int clamp_min, int clamp_max, float ditheramp);
    static float exposure (float value, float gain, float invgamma);

    // Error reporting
    void error (const char *message, ...);
    const char *error_message () { return errmessage; }


 private:
    char *errmessage;
};



class GELATO_PUBLIC ImageInput {
 public:
    ImageInput ();
    virtual ~ImageInput ();

    // Open file with given name.  Various file attributes are put in
    // newspec and a copy is also saved in this->spec.  From these
    // attributes, you can discern the resolution, if it's tiled,
    // number of channels, and native data format.  Return true if the
```

```
// file was found and opened okay.
virtual bool open (const char *name, ImageIOFormatSpec &newspec,
    int nparams, const ImageIOParameter *param) = 0;

// Return the subimage number of the subimage we're currently reading.
// Obviously, this is always 0 if there is only one subimage in the file.
virtual int current_subimage (void) const { return 0; }

// Seek to the given subimage.  Return true on success, false on
// failure (including that there is not a subimage with that
// index).  The new subimage's vital statistics are put in newspec
// (and also saved in this->spec).  The reader is expected to give
// the appearance of random access to subimages -- in other words,
// if it can't randomly seek to the given subimage, it should
// transparently close, reopen, and sequentially read through
// prior subimages.
virtual bool seek_subimage (int index, ImageIOFormatSpec &newspec) {
    return false;
}

// Read the scanline that includes pixels (*,y,z) into contiguous
// floats beginning at data.  (z==0 for non-volume images.)  The
// stride value gives the data layout: one pixel to the "right" is
// xstride floats away.  The reader is expected to give the
// appearance of random access -- in other words, if it can't
// randomly seek to the given scanline, it should transparently
// close, reopen, and sequentially read through prior scanlines.
// A good default implementation exists that calls read_native_scanline
// and then does appropriate format conversion, so there's no reason
// for each format plugin to override this method.
virtual bool read_scanline (int y, int z, float *data, int xstride);

// Read the tile that includes pixel (x,y,z) into contiguous
// floats beginning at data.  (z==0 for non-volume images.)  The
// three stride values give the data layout: one pixel to the
// "right" is xstride floats away, one pixel "down" is ystride
// floats away, one pixel "in" (the next volumetric slice) is
// zstride floats away.  The reader is expected to give the
// appearance of random access -- in other words, if it can't
// randomly seek to the given tile, it should transparently close,
// reopen, and sequentially read through prior tiles.
// A good default implementation exists that calls read_native_tile
// and then does appropriate format conversion, so there's no reason
// for each format plugin to override this method.
virtual bool read_tile (int x, int y, int z, float *data,
                        int xstride, int ystride, int zstride);

// No supplied stride implies contiguous pixels.
bool read_scanline (int y, int z, float *data) {
    return read_scanline (y, z, data, spec.nchannels);
}
```

```
    bool read_tile (int x, int y, int z, float *data) {
        return read_tile (x, y, z, data, spec.nchannels,
                          spec.nchannels*spec.tile_width,
                          spec.nchannels*spec.tile_width*spec.tile_height);
    }

    // The two read_native routines are just like read_scanline and
    // read_tile, except that they keep the data in the native data
    // format of the disk file, without conversion to float, and
    // always read into contiguous memory (no strides).  It's
    // up to the user to know what to do with the data.
    // THESE ARE THE MOST IMPORTANT ROUTINES FOR EACH FORMAT TO OVERRIDE.
    virtual bool read_native_scanline (int y, int z, void *data) = 0;
    virtual bool read_native_tile (int x, int y, int z, void *data) {
        return false;
    }

    // Try to find a parameter from the currently opened image (or
    // subimage) and store its value in *val.  The user is responsible
    // for making sure that val points to the right type and amount of
    // storage for the parameter requested.  Caveat emptor.  Return
    // true if the plugin knows about that parameter and it's in the
    // file (and of the right type).  Return false (and don't modify
    // *val) if the param name is unrecognized, or doesn't have an
    // entry in the file.
    virtual bool get_parameter (const char *name, ParamType t, void *val) {
        return false;
    }

    // Close an image that we are totally done with.
    virtual bool close () = 0;

    // General message passing between client and image input server
    virtual int send_to_input (const char *format, ...);
    int send_to_client (const char *format, ...);

    // Error reporting
    void error (const char *message, ...);
    const char *error_message () { return errmessage; }

 private:
    char *errmessage;
 protected:
    ImageIOFormatSpec spec;  // spec of current subimage
};



// Utility functions

// Create an ImageOutput or ImageInput for the given format.  The
```

```
// searchpath is the path to find the plugin DSO's, not the searchpath
// to find the images themselves.  If MakeImageOutput isn't given a
// format name, it assumes that the file extension is the format name.
// MakeImageInput will try all imagio plugins to find one that reads
// the format of the input file.
GELATO_PUBLIC ImageOutput *MakeImageOutput (const char *filename,
                                     const char *searchpath);
GELATO_PUBLIC ImageOutput *MakeImageOutput (const char *filename,
                                     const char *format,
                                     const char *searchpath);
GELATO_PUBLIC ImageInput *MakeImageInput (const char *filename,
                                   const char *searchpath);

// If MakeImageInput/Output fail, there's no ImageInput/Output to use to
// call error_message(), so call ImageIOErrorMessage().
GELATO_PUBLIC const char *ImageIOErrorMessage ();

// Helper routines, used mainly by image output plugins, to search for
// entries in an array of ImageIOParameter.
// If the param[] array contains a param with the given name, type, and
// nvalues, then store its index within param[], and return its value.
// Else, set index to -1 and return NULL.
GELATO_PUBLIC const void *IOParamFindValue (const char *name, ParamBaseType type,
                                     int count, int& index, int nparams,
                                     const ImageIOParameter *param);
// If the param[] array contains a param with the given name, and a
// single string value, return that value.  Else return NULL.
GELATO_PUBLIC const char *IOParamFindString (const char *name, int nparams,
                                      const ImageIOParameter *param);

// to force correct linkage on some systems
GELATO_PUBLIC void _ImageIO_force_link ();


}; /* end namespace Gelato */


#endif  // GELATO_IMAGEIO_H
```

A.6 gsoargs.h

```
///////////////////////////////////////////////////////////////////////////
// Copyright 2004 NVIDIA Corporation.  All Rights Reserved.
//
// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions are
// met:
//
// * Redistributions of source code must retain the above copyright
//   notice, this list of conditions and the following disclaimer.
// * Redistributions in binary form must reproduce the above copyright
//   notice, this list of conditions and the following disclaimer in the
//   documentation and/or other materials provided with the distribution.
// * Neither the name of NVIDIA nor the names of its contributors
//   may be used to endorse or promote products derived from this software
//   without specific prior written permission.
//
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
// "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
// LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
// A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
// OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
// SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
// LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
// DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
// THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
// (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
// OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
//
// (This is the Modified BSD License)
///////////////////////////////////////////////////////////////////////////


#ifndef GSOARGS_H
#define GSOARGS_H

#include <cstring>
#include <vector>
#include "export.h"
#include "paramtype.h"


namespace Gelato {


class GELATO_PUBLIC GsoArgs {
public:
    // Parameter holds all the information about a single shader parameter.
    struct Parameter {
        const char *name;                  // name
```

```
    ParamType type;                      // data type
    bool isoutput;                       // true if it's an output param
    bool valid;                          // false if there's no default val
    bool varlenarray;                    // is it a varying-length array?
    std::vector<float> fdefault;         // default float values
    std::vector<const char *> sdefault;  // default string values
    std::vector<const char *> spacename; // space name for matrices and
                                         //   triples, for each array elem.
    std::vector<Parameter> metadata;     // Meta-data about the param
    Parameter() : name(NULL), isoutput(false), valid(false) { }
};

GsoArgs ();
~GsoArgs ();

// Get info on the named shader with optional searcphath.
// Return true for success, false if the shader could not be found
// or opened properly.
bool open (const char *shadername, const char *searchpath=NULL);

// Return the shader type: "surface", "displacement", "volume", "light",
// or "generic."
const char *shadertype (void) const { return _shadertype; }

// Get the name of the shader.
const char *shadername (void) const { return _shadername; }

// How many parameters does the shader have?
int nargs (void) const { return args.size(); }

// Retrieve a parameter, either by index or by name.  Return NULL if the
// index is out of range, or if the named parameter is not found.
const Parameter *getarg (int i) const {
    if (i < 0 || i >= nargs())
        return NULL;
    return &(args[i]);
}
const Parameter *getarg (const char *name) const {
    for (unsigned int i = 0;  i < args.size();  ++i)
        if (! strcmp (args[i].name, name))
            return &(args[i]);
    return NULL;
}

// Retrieve a reference to the metadata about the shader
const std::vector<Parameter> &metadata (void) const { return meta; }

// Return error string, or NULL if there was no error.  Reset the error
// code.
const char *error (void) {
    const char *e = _error;
```

```
        _error = NULL;
        return e;
    }

private:
    const char *_shadername;
    const char *_shadertype;
    const char *_error;
# if defined(_MSC_VER) && (defined(WIN32) || defined(_Windows) || \
     defined(_WINDOWS) || defined(_WIN32) || defined(__WIN32__) || \
     defined(WINNT))
#pragma warning (push)
#pragma warning (disable: 4251)
#endif
    std::vector<Parameter> args;
    std::vector<Parameter> meta;     // Meta-data about the shader
# if defined(_MSC_VER) && (defined(WIN32) || defined(_Windows) || \
     defined(_WINDOWS) || defined(_WIN32) || defined(__WIN32__) || \
     defined(WINNT))
#pragma warning (pop)
#endif
};


}; /* end namespace Gelato */


#endif /* GSOARGS_H */
```

B Glossary

Aliasing. Undesirable image artifacts related to the rendering process inadequately sampling and representing high frequencies in the image.

Alpha. An extra channel in an image giving a measure of *coverage* of a pixel, to aid in image compositing. An alpha of 1.0 means fully opaque; 0.0 means fully transparent.

Ambient occlusion. A technique that uses ray-tracing to compute, for each point in the scene, how much of the hemisphere above the point is exposed to the sky (and thus should be illuminated by "ambient" light) versus how much of the hemisphere is occluded by local objects (and is thus not ambiently illuminated).

Anisotropic. Something that has a directional dependence. When talking about surface reflectivity, it refers to a BRDF that depends on the rotational orientation of a surface, as well as the angles of the incoming and outgoing light.

Antialiasing. Ways of combating aliasing artifacts. Generally encompasses capturing geometric edges without "jaggies," motion blur, depth of field, and otherwise adequately sampling high frequencies in the image.

API. Applications Programming Interface. An API is a set of data types and procedures that define the public interface to a library or program.

Artifact. A visible imperfection in a computer graphics image, particularly one that betrays the fact that the image is CG and is not a real photograph or physical artwork. Aliasing, polygonal silhouettes, oversimplistic shading, and Mach bands are typical examples of artifacts.

Associated alpha. For pixels represented by RGB and alpha (coverage), the practice of *pre-multiplying* the RGB values by the alpha values. This makes the computations for image compositing simpler.

Attribute. Properties that apply to either the scene as a whole or to individual geometric primitives. Examples of attributes include image resolution, object color, shader assignments, and so on.

Backfacing. Surfaces whose surface normals face away from the camera viewpoint.

Bake / Baking. To turn something that would be computed on-the-fly (perhaps repeatedly or expensively) into a static representation that could cheaply used repeatedly at runtime.

For example, *texture baking* usually means taking shader computations that would expensively be done on every frame, computing the results once and storing them in a texture map, then replacing the expensive shader operations with a simple texture lookup.

Beauty pass. The ultimate rendering *pass* that pulls all the prior passes together to form the final image.

Blinn's law. The observation, common in the early days of CGI but first stated by Jim Blinn, that an artist is willing to wait a fixed amount of time for an image to render, and that faster hardware or algorithms simply results in more complex images that take the same amount of time to render.

BRDF. Bidirectional reflectance distribution function. A BRDF is a formula whose inputs are the incoming (L) and outgoing (V) directions on a surface, and its output is the portion of light coming from L that is scattered toward V. A BRDF is the heart of a *local illumination model*.

Caustics. Areas of intense light that result from curved reflectors or refractors (objects that act like mirrors or lenses, respectively) that focus light.

CGI. Computer graphics imagery, especially computer graphics imagery that is produced for use in motion pictures (such as for special effects).

Channel. A single "pane" of data in an image. For example, an RGBA image consists of four channels: red, green, blue, and alpha.

CPU. Central processing unit — traditionally, the main computational unit that executes software on a computer. Examples of CPUs include the Intel Pentium 4 and the AMD Opteron. See also *GPU*.

CTM. Current transformation matrix — at any stage of reading scene input, the 4×4 matrix that describes the current "local" coordinate system (relative to the world). Transformation routines (such as `AppendTransform`, `Rotate`, etc.) modify the CTM. The CTM may be temporarily saved and restored with `PushTransform` and `PopTransform`, and also implicitly by `PushAttributes` and `PopAttributes`.

Deformation blur. Motion blur of the shape of an object, by blurring the positions of the object's control vertices. This can describe movement that is not "rigid." (See *transformation blur*.)

Depth of field. The property of physical cameras that only a limited range of distances can be in focus at any one time.

Displacement bound. Extra space added to the bounding box of an object to account for the fact that displacement might make the primitive "poke out" of the original bounds.

Displacement shading. Allowing shaders to alter the shape of surface geometry, usually to add fine detail.

DLL. Dynamically linked library – a library file that may be loaded dynamically by a program at runtime. Gelato's ImageIO, Generator, and shadeop plugins are implemented as DLLs. "DLL" is a name specific to MS Windows; in the Unix world, these are called "DSOs."

DSO. Dynamic shared object – a library file that may be loaded dynamically by a program at runtime. Gelato's ImageIO, Generator, and shadeop plugins are implemented as DSO's. Note that DSO's are called DLL's under MS Windows.

Fill light. A dim, usually nonspecular, light that fills in areas of a scene not illuminated by a key light.

Frame-parallel rendering. Using multiple machines on a network to each render a separate frame of an animation. Contrast to *network-parallel rendering* and *multithreading*.

Frontfacing. Surfaces whose surface normals face toward the camera viewpoint.

Gamma correction. A nonlinear scaling of the values in an image to compensate for the property of all physical display devices that they react to input values in a non-linear way.

Geometric primitive. An individual piece of geometry, such as a `Patch`, `Mesh`, `Sphere`, and so on.

Geometry set. A named collection of geometric primitives, either corresponding to objects visible to a camera, or a group that may be ray traced. An object may be present in any number of geometry sets. The `Attribute "geometrysets"` controls which geometry sets are active.

Global illumination. Calculation of how light affects an entire scene, especially the contribution of light reflected between surfaces (as opposed to coming straight from a light source). In Gelato, this refers to ray-traced shadows and reflections, indirect illumination, caustics, and ambient occlusion.

Global variable. In GSL, any of the built-in variables describing the shading situation, for example, `P`, `N`, `u`, and so on.

GPU. Graphics processing unit — the main computational unit of a programmable graphics card, such as the NVIDIA Quadro FX line. In addition to quickly drawing shaded triangles, GPU's are very good at performing mathematical computations in a highly parallelized manner. See also *CPU*.

GSL. Gelato Shading Language.

HDRI. Short for High Dynamic Range Imagery, it refers to images that can capture the entire dynamic range of a real scene. In short, a floating-point image or environment map, as opposed to using 8- or 16-bit integers to represent light levels in an image.

IBL. Short for *Image-Based Lighting*.

Image-based lighting. Lighting a scene by data captured from a real-world scene, usually in the form of an *HDRI* environment map.

Indirect illumination. Light that reflects diffusely off an object before illuminating another object in the scene.

Isotropic. Means "the same in all directions." When referring to surface reflectivity, it means a BRDF that depends only on the angles of the incoming and outgoing light relative to the surface normal, without regard to the rotational orientation of the surface about its normal.

Key light. A major source of illumination in a scene, usually resulting in hard shadows and specular highlights.

Local illumination model. A formula that, given the directions and intensities of light impinging on a surface, computes the amount of light scattered away from the surface in a particular direction (such as toward the camera). Synonyms: Local reflection model, BRDF.

Local reflection model. Same as **Local Illumination Model**.

Metadata. Annotations embedded in a shader that do not change the operation of the shader code itself, but describe its contents. Gelato allows metadata about a shader as a whole, as well as metadata specific to each of the shader's parameters.

MIP-map. A texture map for which the results of filtering the texture with a series of larger and larger filters (typically sized in powers of two) has been precalculated and stored with the map, in order to speed render-time texture access. "MIP" actually stands for "multum in parvo" (Latin for "much in little").

Modeler. A program that allows users to specify the shape of geometric objects and to place objects, lights, and cameras in a virtual scene.

Moore's law. The observation that computing power (as measured by the time it takes to perform certain fixed benchmark calculations on new computers) increases exponentially over time, and that historically it has doubled every 18 months or so over a very long time span. Contrast with **Blinn's law**.

Motion block. A transformation or geometric primitive that changes over time. Specifically, a `Motion` statement (with *n* time values passed to it), followed by exactly *n* identical transformations or by exactly *n* geometric primitives (of the same time), each corresponding to the position or shape at one of the time values, respectively.

Motion blur. The property of physical cameras that objects that move relative to the camera leave a streak in photographs, proportional to the length of time the camera shutter is open.

Multithreading. Using multiple CPU's or GPU's in a single computer to contribute to the computation of a single rendered frame. Contrast to *frame-parallel rendering* and *network-parallel rendering*.

Network-parallel rendering. Using multiple machines on a network to contribute simultaneously to rendering a single frame. Contrast to *frame-parallel rendering* and *multithreading*.

Parametric coordinates. The (typically) two values that uniquely specify a point on a parametric surface, such as a Patch.

Pass. A final image may require several separate invocations of the renderer — *passes* — which may include generating shadow depth maps, reflection maps, ambient occlusion images, caustic photon maps, diffuse databases for subsurface scattering, etc. The ultimate pass that pulls all the prior passes together to form the final image is called the *beauty pass*.

Primitive. Short for *geometric primitive*.

Primitive Variables. Data attached to a geometric primitive (per primitive, facet, corner, or vertex), interpolated by the renderer, and that can be accessed in a shader.

Projection. A transformation which "flattens" space by removing one dimension, for example, converting points in a 3D space into positions on a 2D object (such as a plane or the surface of a sphere).

Radiosity. A global illumination method involving solution by finite element methods. Radiosity solutions usually make the assumption that all surfaces are perfectly diffuse. Sometimes *radiosity* refers colloquially to any global illumination algorithm.

Ray casting. A method of global visibility determination, that computes the intersection of viewing "rays" with scene geometry for any purpose.

Ray tracing. A method of rendering. Ray tracing solves hidden surfaces, shadows, and reflections by computing the intersection of viewing "rays" and scene geometry.

Reference geometry. A description of geometry in a canonical pose. As the "real" geometry is deformed by animation, the reference geometry can be used for shading calculations to ensure that any patterns computed by the shader will stick to the surface as it deforms.

Renderer. A program that takes a description of a scene (camera, objects, materials, lights) and produces an image.

Scanline rendering. A family of rendering methods that involve projecting geometry into screen space, and handling geometric primitives in image order.

Shadeop. Short for *shading operation*, shadeops are the built-in operators and functions in GSL (in other words, the operators and functions that the SL compiler already knows about).

Shader. A computer program that describes the appearance of a surface, light, or volume.

Shader type. One of `surface`, `displacement`, `volume`, or `light`, or `shader` (indicating a generic shader). The type of a shader is given in its declaration (in the shader source) and determines which global variables it may access and which operations it may legally

perform (for example, light shaders may not alter `P`, and volume shaders may not `emit` light).

Shader usage. One of `surface`, `displacement`, `volume`, or `light`, the shader usage explains in what stage of the rendering pipeline a shader is executed. The shader loaded for a given usage must have a matching shader type (or, if a generic shader, must still only perform operations legal for the usage).

Shading quality. A measure of how frequently color values are computed on surfaces. Larger values imply that shading is computed more frequently, therefore more total shading calculations will be performed (with the expected increase in cost). Smaller values will result in fewer total invocations of the shader, thus rendering in less time and memory, but with lower image quality.

Space. A synonym for "coordinate system."

Subsurface scattering. Illumination that scatters internally through a translucent material such as marble or skin, often re-emerging quite far from where it entered the material.

Texel. TEXture ELement. A texel is one pixel in a texture map (including shadow and environment maps).

Texture mapping. Taking colors (or other data) from a stored image file and applying the pattern to a surface to give added detail.

Transformation. The placement of an object (or light, camera, etc.). Transformation includes translation, rotation, and scaling of an object, and is accomplished with the API routines described in Section 2.5.

Transformation blur. Motion blur of the position/orientation of an object. This can describe "rigid" motion of an object but does not allow an object do bend or deform. (See *deformation blur*.)

Index

`aastep()`, 129
`abs()`, 120
`acos()`, 119
ambient occlusion, 94, 140, 228–231
`ambient()`, 138
antialiasing, 69, 181
`AppendTransform`, 30, 55
arbitrary output variables, 176
area light source, 33
`arraylength()`, 149
`asin()`, 119
asynchronous rendering, 19
`atan()`, 119
`Attribute`, 21, 55
attributes, 21
    copy-on-write semantics, 10
    modify mode, 11
    per-object, 27
    renderer information and status, 84
    scene-wide, 25
    user attributes, 21

baking, 91
Basic Gelato, 5
binary Pyg, *see* Pyg, 85
bucket block size, 70
bucket order, 69
bucket size, 69
`bump()`, 143

`C`, 88
`Camera`, 15, 55
camera, 165–172
    attributes, 67
    clipping planes, 68
    crop window, 68
    depth of field, 68
    field of view, 67
    motion blur, 68
    multiple cameras, 185–186
    pixel aspect ratio, 68
    positioning, 165
    projection, 67, 166
    resolution, 68
    screen window, 67
    shutter, 68
caustics, *see* global illumination
`ceil()`, 120
`clamp()`, 121
clipping, 68
clipping planes, 172
`cnoise()`, 127
`color()`, 125
`Command`, 19, 55, 95
`Comment`, 20, 55
`concat()`, 131
`ConnectShaders`, 32, 55
`cos()`, 119
`cosh()`, 120
`CreateRenderer`, 10, 14, 49
crop window, 68
crop windows, 173
cross product, 122
`cross()`, 122
CTM, 29, 87
cube-face directions, 206
cube-face environment maps, *see* environment maps
cube-face shadow maps, *see* shadows
`cull:occlusion`, 91
`Curves`, 41, 56

`debug:filesread`, 82
`debug:shadernan`, 82
debugging, 81

degrees(), 119
deltau(), 129
deltav(), 129
depth of field, 68, 69, 171–172, 182
determinant(), 125
dice:binary, 89
dice:curvature, 90
dice:fixed, 90
dice:fixeduname, 79
dice:fixedvname, 79
dice:highcurvature, 90
dice:keepcreases, 90
dice:motionfactor, 90
dice:rasterorient, 90
dice:thincurve, 91
dicing
  binary dicing, 89
  curvature thresholds, 90
  fixed dicing rates, 79, 90
  keepcreases, 90
  motionfactor, 90
  raster oriented dicing, 90
  ray tracing displacements, 93
  thin curve dicing, 91
diffuse(), 138
displace(), 143
displace:maxradius, 91
displace:maxspace, 91
displacement bound, 91
displacements
  ray traced, 93
distance(), 122
dofquality, 69, 182
dot product, 122
dot(), 122
DSO Shadeops, 277–288
dynamic shadow maps, *see* shadows

emit, 110
encrypting shaders, 189
environment maps, 199, 200, 206, 220–222
  cube-face environment maps, 199, 204–205
  lat-long environment maps, 199
  latitude-longitude environment maps, 204
  light probes, 199
environment variables, 159
environment(), 133
erf(), 121
erfc(), 121
Err, 49
error log, 81
error(), 131
errormanager.h, 307
exit(), 149
exp(), 120

fabs(), 120
Face Mesh, 39
faceforward(), 122
far, 68
field of view, 67
filter
  pixel, 73
filter width, 73
filtering, 183
filterwidth(), 129
finite(), 121
floor(), 120
format(), 130
format:binary, 85
fprintf(), 131
fresnel(), 123

gain, 73
gamma, 73
gamma correction, 73
Gelato Pro, 5
GELATO_LICENSE_MODE, 159
gelatoapi.h, 299
GELATOHOME, 159
GELATOTEMP, 159
generators, 44
geometric primitives
  curves, 41
  mesh, 39
  NURBS, 36, 37
  patch, 36, 37
  points, 41
  polygon meshes, 39
  procedural geometry, 44

sphere, 43
subdivision surfaces, 39
trim curves, 38
geometry set, 133, 136
geometry sets, 87–88
GetAttribute, 22, 56
getattribute(), 146
getmessage(), 147
gettextureinfo(), 148
global illumination, 93
ambient occlusion, *see* ambient occlusion
caustics, 235–237
indirect illumination, 225–227
global illumination attributes, 80
global variables, 116
table of, 117
grid dump, 82, 83
grid size, 78
gridany(), 130
gridindex(), 130
gridmax(), 130
gridmin(), 130
gridn(), 130
gridnu(), 130
gridnv(), 130
gslc, 187–189
command line arguments, 188
gsoargs.h, 322
gsoinfo, 190, 293–296

holdout matte geometry, 88
hypot(), 120

Image I/O Plugins, 174, 177–180, 257–276
image resolution, 68
image-based lighting, 233–234
imageio.h, 315
indirect illumination, *see* global illumination
indirect(), 138
indirect:maxerror, 93
indirect:maxpixeldist, 93
indirect:minsamples, 93
indirect:spatialdb, 93
Input, 44, 56
inversesqrt(), 120
isindirectray(), 149
isinf(), 121
isnan(), 121
isshadowray(), 148
iv, 158, 160, 178

key words, 153

lat-long environment maps, *see* environment maps
layered shaders, 32–34
length(), 122
libgsoargs, 289
libsleargs, 289–296
Light, 33, 56
light categories, 111
light probes, *see* environment maps
light source, 33, 34
lights, 111
LightSwitch, 34, 56
limits:autopassreduction, 70
limits:bucketblocksize, 70
limits:bucketsize, 69
limits:gridsize, 78
limits:inputlevels, 86
limits:opacitythreshold, 79
limits:texture memory, 78
limits:texturefiles, 78
limits:threads, 78
limits:transparentgrids, 79
limits:transparentlayers, 79
limits:trimcurvememory, 79
LM_LICENSE_FILE, 159
log(), 120
log10(), 120
log2(), 120
luminance(), 125

maketx, 197–201
match(), 131
matrix(), 125
max(), 121
Mesh, 56
metadata, 106, 190, 191, 290–292
min(), 121
mix(), 121
mod(), 120

Modify, 11, 23, 57
modify mode, 11
Motion, 57
motion blur, 68, 69, 168–171, 181–182
    motion-blurred ray tracing, 92
    motionfactor, 90
multithreaded rendering, 78, 162

named attribute states, 23
named geometry, 87
NaN, 82
near, 68
network-parallel rendering, 85, 162
noise(), 126
non-blocking render, 19
normal(), 121
normalize(), 122
normals
    interpolated, 36
NVIDIAD_LICENSE_FILE, 159

occlusion, 94, 140
occlusion culling, 91
occlusion(), 140
occlusion:maxerror, 94
occlusion:maxpixeldist, 94
occlusion:minsamples, 94
occlusion:spatialdb, 94
opacity, 88
opacity threshold, 79
OpenEXR, 179, 205
orientation, 89
Output, 17, 57
Output, 174–177

Parameter, 13, 57
paramtype.h, 304
pass, 76
Pass name, 76
Patch, 36, 37, 57
paths, 80
photon mapping, 235
pixel aspect ratio, 68
pixel filter, 73
pnoise(), 127
point(), 121
Points, 41, 57
PopAttributes, 22, 57
PopTransform, 29, 58
pow(), 120
preview mode, 70
preview modes, 160
printf(), 131
procedural geometry, 44
projection, 67, 166
psnoise(), 127
PushAttributes, 22, 58
PushTransform, 29, 58
Pyg, 53–65, 85, 86
    binary pyg, 62, 85
    Pyglet, 61
pyg:binary, 85
pyg:indent, 86
pyg:separateparams, 86
Pyglet, 61
Python binding, *see* Pyg

quantization, 73, 175

radians(), 119
random(), 128
randomgrid(), 128
ray tracing, 92
    maximum ray depth, 80
    motion-blurred ray tracing, 92
    per-object controls, 92
    ray-traced reflections, 223–224
    shadows, 218–219
ray tracing attributes, 80
ray:displace, 93
ray:maxdepth, 80
ray:maxdepthcolor, 80
ray:motion, 92
ray:opaqueshadows, 92
raylevel(), 148
Re-render mode, 77, 94
re-rendering, 11, 245–249
reflect(), 123
reflections, 220–224
    environment maps, 199, *see* environment maps
refract(), 123

remote display with `iv`, 160
remote rendering, 14
`Render`, 19, 58
renderer information attributes, 84
`rerender`, 77
`rerender:cachelights`, 94
`rerender:filepattern`, 77
`rerender:locked`, 94
`rerender:memory`, 77
`rerender:reshaderays`, 77
reserved words, 153
resolution, 68
`resolution`, 172
`RestoreAttributes`, 23, 58
`Rotate`, 30, 58
`rotate()`, 124, 126
`round()`, 120

`samplearea()`, 129
`SaveAttributes`, 23, 58
`Scale`, 30, 58
`scale()`, 126
screen window, 67
search paths, 80
`setmessage()`, 146
`SetTransform`, 29, 58
`shadeop.h`, 310
`Shader`, 32, 59
shader compiler, *see* gslc
shader metadata, *see* metadata
Shader parameters, 289
`ShaderGroupBegin`, 32, 59
`ShaderGroupEnd`, 32, 59
shaders, 32–34
Shading Language
  comments, 100
  conditionals, 109
  data types, 100
    color, 101
    float, 100
    matrix, 103
    point, vector, normal, 101
    string, 104
  derivatives, 129
  expressions, 114
  formal syntax, 150
  function definitions, 113
  identifiers, 99
  loops, 109
  parameter declarations, 104
  preprocessor directives, 100
  procedure calls, 108
  scoping, 113
  syntax, 107
  variable declarations, 108
shading quality, 89
`shading:view`, 91
shadow bias, 93
shadow maps, *see* shadows, *see* shadows
`shadow()`, 136
`shadow:bias`, 93
`shadow:opacitythreshold`, 79
shadows, 207–219
  cube-face shadow maps, 200, 203, 206, 216–217
  dynamic shadow maps, 217–218
  ray traced, 218–219
  shadow bias, 209, 210, 218
  shadow maps, 200, 207–218
    cube-face shadow maps, 200
    dynamic shadow maps, 18, 75
  volume shadow maps, 17, 74, 200, 202–203, 213–216
  Woo shadow maps, 212–213
shutter, 68
sidedness, 89
`sign()`, 120
`sin()`, 119
`sinh()`, 120
`smoothstep()`, 126
`snoise()`, 127
Sorbetto, 245–249
spatial databases, 83
`spatialdb:read`, 83
`spatialdb:readonly`, 84
`spatialdb:write`, 83
`spatialdb:writeonly`, 84
`spatialquality`, 69, 181
`specular()`, 138
`specularbrdf()`, 138

`Sphere`, 43, 59
`spline()`, 128
`sqrt()`, 120
statistics, 81
`step()`, 126
stereo rendering, 18, 71, 74, 184–185
`substr()`, 131
subsurface scattering, 142, 238–243
`subsurface()`, 142
surface color, 88
surface opacity, 88
`surfacenormal()`, 129

`tan()`, 119
`tanh()`, 120
`temporalquality`, 69, 181
texture
    maximum open texture files, 78
    texture memory, 78
texture formats, 201–205
texture mapping, 132–137
texture maps, 198
texture substitution, 83
`texture()`, 132, 133
threads, 78
TIFF texture formats, *see* texture formats
`topyg`, 63
`trace`, 112
`transform()`, 123, 124
transformations, 29–31
`transformc()`, 125
`transformn()`, 123, 124
`transformu()`, 124
`transformv()`, 123, 124
`Translate`, 30, 59
`translate()`, 126
transparency, 79
`transpose()`, 126
trim curves
    API for trim curves, 38
    quality settings, 78, 92
    sense, 92
    trim memory, 79
`TrimCurve`, 38, 59
`trimcurve:curvature`, 92
`trimcurve:quality`, 78
`trimcurve:sense`, 92
`trunc()`, 120
`twosided`, 89

units, 76, 124, 194
`units:fps`, 76
`units:length`, 76
`units:time`, 76
user attributes, 21

`vector()`, 121
volume shadow maps, *see* shadows
`volz` shadows, *see* shadows

`World`, 18, 59

`zchannels`, 74