3Delight 10.0 User's Manual

A fast, high quality, RenderMan-compliant renderer

Short Contents

Table of Contents

1 Welcome to 3Delight!

3Delight is a fast, high quality, RenderMan compliant renderer. *3Delight* comes as a set of command line tools and libraries intended to help you render production quality images from 3D scene descriptions. At the heart of the rendering tools is the *3Delight* rendering engine. This engine implements a fast scanline renderer, coupled with an on-demand ray tracer. This combination allows *3Delight* to render images quickly while giving you the ability to easily incorporate ray tracing effects whenever you need them.

3Delight was first released publicly in August 2000. Since then numerous releases, both public and internal, have contributed to making it a stable renderer with strong features. In the process, special care has been taken to make sure *3Delight* keeps its performance edge.

Check regularly at `http://www.3delight.com` for the latest release of *3Delight*.

1.1 What Is In This Manual ?

The rest of this chapter gives an overview of the main features of the *3Delight* rendering tools.

The installation procedure and insights about *3Delight*'s working environment are given in Chapter 2 [Installation], page 4.

The complete list of provided tools and libraries, along with operational details, is given in Chapter 3 [Using 3Delight], page 7. RenderMan compliance and *3Delight* specific extensions are discussed in Chapter 5 [3Delight and RenderMan], page 29. Shading Language (SL) support and shadeop details are given in Chapter 6 [The Shading Language], page 62. Rendering tips and useful suggestions can be found in Chapter 7 [Rendering Guidelines], page 120. *3Delight*'s extensible display driver interface and descriptions of all provided display drivers can be found in Chapter 8 [Display Drivers], page 168.

1.2 Features

Here is an overview of *3Delight* features:

RenderMan Compliant
: The `renderdl` program can render any RenderMan Interface Bytestream (RIB) file (binary or text formats) or an application can link with the `lib3delight` library directly and use the RenderMan Application Programming Interface (API), refer to User's Manual Chapter 5 [3Delight and RenderMan], page 29 for details.

RenderMan Shading Language Support
: Programmable shading and lighting with an optimizing shader compiler. RenderMan shaders are fully supported (surface, displacement, light, atmosphere, interior and imager). Matrices, arrays, normals, vectors and all the standard shadeops are supported. DSO shadeops, light categories, message passing and output variables are also supported. Shaders can be either compiled or interpreted. See User's Manual Section 3.2 [Using the shader compiler], page 11.

Rich Rendering Features
: Depth of field, motion blur, level of detail and surface displacement. Standard and deep shadow maps, as well as ray traced shadows. Selective ray tracing, global illumination and point-based graphics. Atmospheric effects.

Textures and Antialiasing
: High quality filtered textures and selectable antialias filters including the quality `sinc` and `catmull-rom` filters;

Rich Geometry Support
: Subdivision Surfaces (catmull-clark), polygons, patches (B-spline, Bezier, Catmull-Rom and others), NURBS (with trim-curves), curves, quadrics and implicit surfaces (blobbies). User defined variables, including vertex variables, attached to geometry are fully supported.

Procedural Geometry
: RenderMan procedurals are fully supported. Goemetry can be specified in a delayed fashion using either `DelayedReadArchive`, `RunProgram` or `DynamicLoad`, refer to User's Manual Section 5.3.8 [procedural primitives], page 57.

Fast and Efficient Rendering
: *3Delight* can handle complex scenes made of millions of primitives. From its initial design stage, rendering speed has been a TOP PRIORITY and it continues to be in its ongoing development.

Extensible Display Drivers
: *3Delight* comes with the following display drivers: `framebuffer`, `tiff`, `bmp`, `zfile`, `shadowmap`, `dsm`, `cineon`, `radiance`, `exr` and `eps`. Since *3Delight*'s uses the "standard" RenderMan display driver interface, third party display drivers are also supported. New extensions to the display system are also supported, including multiple displays per render and display specific quantize parameters. See User's Manual Section 5.1 [options], page 29.

Multi-platform Support with Specific Code Optimization
: *3Delight* is available for Windows, Linux, MacOS X and IRIX.

2 Installation

2.1 MacOS X

An auto-install package is provided for MacOS X users. Administrator privileges are needed to install *3Delight* because everything is copied to `/Applications/Graphics/3Delight-3.0.0/`, which is not accessible to all users. The installation program asks for the appropriate password.

The installer adds a few lines to the `/etc/csh.cshrc`, `/etc/csh.login` and `/etc/bashrc` files. The original files are saved as `/etc/csh.cshrc.3delight.bck`, `/etc/csh.login.3delight.bck` and `/etc/bashrc.3delight.bck` respectively.

To test your installation, perform the following easy steps:

1. Open a Terminal and type the following commands:

   ```
   cd $DELIGHT/examples/rtshadows
   renderdl shadtest.rib
   ```

2. A file named `shadtest.tif` should appear in the current directory. You can view it with any image viewer.

Other examples are available in the `examples` folder. We suggest you run them.

If you encounter problems during installation, write to info@3delight.com and we will be glad to provide assitance.

2.2 UNIX

For UNIX systems (IRIX, Linux), an installation script is provided. It should do everything for you. Here is an example of an installation session (the 3Delight `.gz` file has been downloaded into the `~/downloads` directory and is to be installed in the `~/software` directory).

```
% cd ~/downloads
% gunzip 3delight-3.0.0-Linux-i686-libstdc++-3.tar.gz
% tar xf 3delight-3.0.0-Linux-i686-libstdc++-3.tar
% cd 3delight-3.0.0-Linux-i686-libstdc++-3
% ./install --prefix ~/software/
% cd ~/software/3delight-3.0.0/
% unsetenv DELIGHT
% source .3delight_csh
% cd $DELIGHT/examples/opacity/
% shaderdl fonky.sl
% renderdl cubits.rib
% exit
```

After typing those commands, `cubits.tiff` should appear in your directory.

Note that if you do not specify the `--prefix` option to `install`, *3Delight* is installed in `/usr/local`[1].

Finally, you should add the following line to your `.login` file:

```
source ~/software/3delight-3.0.0/.3delight_csh
```

If you use `bash`, then you should add:

```
source ~/software/3delight-3.0.0/.3delight_bash
```

[1] "root" permissions are required when installing in `/usr/local`.

2.3 Windows

On Windows systems, simply run `setup-3.0.0.exe`. Depending on the user's system privileges, the installer will ask whatever is necessary to install *3Delight* for either all users or for only the current user. In both cases, three different installations are possible:

Full Installation
: Installs everything needed to run *3Delight* and all the files needed for development.

Minimal Installation
: Only installs files needed to run *3Delight* (includes all binaries).

Custom Installation
: User can choose exactly what is installed.

To unninstall *3Delight*, simply run `Uninstall 3Delight` in the startup menu.

Silent Command-Line Installation

It is possible to install *3Delight* from the command line without popping the installation dialog and with no questions asked:

```
C:\> 3delight-7.0.0-setup.exe /VERYSILENT /DIR="c:\Program Files\3Delight"
```

This can be useful to automate network installations. For example, with the PsExec tool and a proper domain setup, a command similar to this can be used to install or upgrade 3Delight on pc1 and pc2:

```
psexec \\pc1,pc2 -i \\server\public\3delight-7.0.0-setup.exe /VERYSILENT
```

2.4 Environment Variables

3Delight needs only two environment variables to run properly and both should be correctly set by the installer.

`DELIGHT` This variable must point to the root of your *3Delight* installation.

`PATH` `$DELIGHT/bin` must be part of this environment variable.

The following environment variables are not necessary for the proper running of *3Delight* but they can be used to enable some special features.

`RISERVER`[2] If defined, and set to a valid filename, this variable forces *3Delight* to output all `Ri` commands to a RIB file. More on RIB output in Section 7.15 [Outputting RIB], page 156.

`RIFORMAT` Specifies the format of the RIB output, can be either '`ascii`' or '`binary`'.

`DL_RIB_ARCHIVEPROCEDURALS`
: Enables the "rib" "archiveprocedurals" option. The main purpose is debugging C API applications.

`DL_OUTPUT_DIR`
: If this variable contains a valid directory, all output file names are post concatenated with it. For example, setting `$DL_OUTPUT_DIR` to `/tmp`, transforms `Display "test.rif" "file" "rgb"` into `Display "/tmp/test.rif" "file" "rgb"`. This also applies to files generated by `RiMakeTexture`, `RiMakeShadow`, `RiMakeCubeFaceEnvironment` and `RiMakeLatLongEnvironment`.

`DL_DUMP_DEFAULTS`
: Forces *3Delight* to dump the contents of the configuration file(s) it reads. More on this subject in Section 5.5 [configuration file], page 59.

[2] The deprecated `DL_RIB_OUTPUT` is also accepted for compatibility with older versions

`DL_SHADERS_PATH`
`DL_TEXTURES_PATH`
`DL_DISPLAYS_PATH`
`DL_ARCHIVES_PATH`
`DL_PROCEDURALS_PATH`
`DL_RESOURCES_PATH`

These variables specify default search paths for different *3Delight* resources. They can be extracted using the '`@`' symbol when specifying search paths through `RiOption`. Note that default search paths can also be specified through the configuration file (see Section 5.5 [configuration file], page 59) but setting them through environment variables *overrides* configuration file defaults.

`RIBDIR` This environment variable is set by *3Delight* in each `RiReadArchive` call to the path of the *current* RIB. This lets the user load RIB archives that have a path relative to the parent RIB without using absolute paths. For example, if the master RIB is `/scene/master.rib` and the archive to be loaded is `/scene/models/car.rib`, one can write the following in `master.rib` to load the archive:

```
ReadArchive "$RIBDIR/models/car.rib"
```

The advantage of using such relative paths is of importance: entire RIB directory structures can be moved elsewhere and still have correct archive paths.

`DL_SEARCHPATH_DEBUG`

When this variable is set to an integer value, *3Delight* prints a trace of how it uses the search paths. A value of 1 or higher displays the files found. A value of 2 or higher displays every location in which a file is searched for. A value of 3 or higher shows how the dirmap option (see [Search Paths], page 39) is applied.

`DL_DUMP_CORE`

When this variable is set, *3Delight* will not install its own signal handlers. This means crashes will result in core dumps instead of a stack trace, among other things. This is only useful for debugging.

3 Using 3Delight

3Delight is a collection of tools and a library. Here is an overview:

- `renderdl`, a RIB file reader. This programs reads a binary or ASCII-encoded RIB file and calls the renderer to produce an image.
- `shaderdl`, a shader compiler. The compiler can produce either object-code shaders (a DSO or a DLL) or byte-code shaders (commonly called interpreted shaders).
- `tdlmake`, a texture optimizer which reads a number of input image files and produces a TIFF optimized for the renderer. The optimized file typically has a `.tdl` extension.
- `shaderinfo`, a utility that gathers information from compiled shaders.
- `dsm2tif`, a utility to convert deep shadow maps into TIFFs.
- `hdri2tif`, a dynamic range compression utility. Converts HDR images to low dynamic range TIFFs.
- `ribdepends`, a utility to list RIB file dependencies and to make *site independent* packages.
- `ribshrink`, a utility to compress a series of RIB files.
- `ptcmerge` a point cloud file merger.
- `ptc2brick` a point cloud to brick map (3D texture) converter.
- `ptcview`, a point cloud file viewer.
- `i-display`, an advanced flip book and display driver, capable of sequence playback.
- `lib3delight`, a library that can be linked to other applications to render images and interrogate shaders. **The library complies to the RenderMan API**.

The next sections describe each of these tools in more detail.

3.1 Using the RIB Renderer - `renderdl`

`renderdl` reads a file containing scene description commands and "executes" them. Such files are commonly called RIB files (RIB stands for RenderMan Interface Bytestream). There are two kinds of RIB files: ASCII encoded RIB files and binary encoded RIB files. A binary RIB file is smaller than its ASCII encoded equivalent, but an ASCII RIB file has the advantage of being editable in any text editor or word processor.

To render a RIB named `file.rib`, just type:

```
% renderdl file.rib
```

It is possible to render more than one file:

```
% renderdl file1.rib file2.rib file3.rib
```

In this case, `renderdl` reads each file one after the other, and the graphic state is retained from one file to another (in other words, the graphic state at the end of one file is the starting graphic state for the next file). If a file cannot be found, it is simply skipped. This behavior is useful to separate the actual scene description from rendering options. For example:

```
% renderdl fast.opt scene.rib
% renderdl slow.opt scene.rib
```

These render the scene `scene.rib` twice but with different rendering options (note that `fast.opt` and `slow.opt` are normal RIB files). `slow.opt` contains options for high quality rendering such as low `ShadingRate` and high `PixelSamples`, and `fast.opt` contains low quality (speedy) option settings.

If no file name is specified, `renderdl` reads scene description commands from the standard in. This feature enables piping commands directly in `renderdl`. For example, to enter scene description commands interactively (which is not really practical), do the following:

```
% renderdl
Reading (stdin)
<enter commands here>
```

If you wish to pipe the content of `file.rib` in `renderdl`, type:

```
% cat file.rib | renderdl
```

`renderdl` options are described in the following sub-section.

Command Line Options

`-t n`

`-p n` Specifies the number of threads to use for the rendering. '`n`' can take any of the following values:

$n > 0$	Use '`n`' processors.
$n = 0$	Use maximum number of processors available.
$n < 0$	Use all but '`-n`' processors.

It is also possible to use this option with a separator, like this:

```
renderdl -t:2 -d cubits.rib
```

`-P n` Specifies the number of processes to use for the render. '`n`' can take the same values as in the `-p` option. The way *3Delight* cuts the image is controlled using the `-tiling` option. Note that each process will run with a single thread unless explicitly requested otherwise on the command line.

`-hosts host1,host1,...,hostn`
Specifies a list of machines to use for rendering. It is also possible to specify a file (instead of a host name) that contains a list of machines to use. The format of the file is very simple: one machine name per line. This option is not yet functional on Windows platforms.

`-tiling` Specifies the tiling mode to use when splitting an image for multiprocess rendering. Four tiling modes are supported:

'`b`'	For balanced tiling (default). Uses feedback from the previous multiprocess render to improve the split.
'`m`'	For mixed tiling. Splits the image into almost identical squares.
'`v`'	For vertical tiling. Splits the image into vertical stripes.
'`h`'	For horizontal tiling. Splits the image into horizontal stripes.

`-ssh` Use `ssh` instead of `rsh` to start remote renders. More about network rendering in Section 7.1 [Multithreading and Multiprocessing], page 120.

`-jobscript script`
Use `script` to start remote renders. The script receives the command to run as its first argument and is responsible for choosing a machine and executing the command there in a proper environment.

`-jobscriptparam param`
When used with `-jobscript`, renderdl will pass 'param' as the first argument of the script. The command to run then becomes the second argument.

`-d` Forces a display to the `framebuffer` display driver. Note that a `framebuffer` display driver is added to the displays declared inside the RIB so those are still called. If there is already a `framebuffer` display driver declared in the RIB it is ignored.

`-D` Has the same effect as `-d` but automatically closes the framebuffer display driver when rendering ends.

`-id` Starts `i-display` and launches the render, enabling the user to send multiple renders into a centralized application. See Section 8.1 [dspy_idisplay], page 168.

`-idf` Same as `-id` but sets the display to output floating point data.

`-nd` Ignores all framebuffer display drivers declared in the RIB file(s). This option overrides `-id`, `-d` and `-D`.

`-displayvar var`
Works in conjunction with `-id`, `-d` or `-D` to specify which display variable to render. For example:

```
renderdl -id -displayvar "color aov_occlusion" test.rib
```

The default value for this option is '`rgba`'.

`-res x y` Specifies the resolution to use when rendering the specified RIB. This overrides any `RiFormat` command specified in the RIB file[1].

`-frames f1 f2`
Renders the frames between f1 and f2, inclusively. This options enables you to render some specific frames inside one RIB file. Frames outside the specified interval are skipped.

`-crop l r t b`
Sets a crop window defined by <l r t b> (left right top bottom). The values should be given in screen coordinates, which means that all values are between 0.0 and 1.0 inclusively. This command line option overrides any `RiCropWindow` command present in the RIB file.

`-noinit` Doesn't read the `.renderdl` file. See [the .renderdl file], page 10.

`-stats[1-3]`
This is equivalent to adding the following line in the RIB file ('`n`' being a number between 1 and 3):

```
Option "statistics" "endofframe" n
```

The higher the level, the more statistics you get at a slight performance cost. The default statistics level is 3. Note that there is no space between `-stats` and the number.

`-statsfile file`
Prints statistics in `file` instead of `stdout`. Has the same effect as putting the following line in the RIB file:

```
Option "statistics" "endofframe" 3 "filename" "file"
```

This option may be combined with the `-stat` in order to reduce the statistic level.

`-progress` Prints a progress status after each bucket. May also be enabled from the RIB file. See [progress option], page 40.

`-catrib` Doesn't render the specified file(s), only outputs the RIB to `stdout`. More about RIB output in Section 7.15 [Outputting RIB], page 156.

`-o file` Output the RIB to the specified `file` instead of stdout. `stdout` and `stderr` are two special file names which output to standard output and error respectively. Only meaningful with `-catrib`.

[1] The aspect ratio is set to 1.0.

`-binary` Output RIB in binary format. Only meaningful with `-catrib`.

`-gzip` Compress RIB using gzip format. This is valid in both ASCII and binary format. Only meaningful with `-catrib`.

`-callprocedurals`
Calls any procedurals found in the RIB when outputting. Only meaningful with `-catrib`. More about procedurals and RIB output in Section 7.15 [Outputting RIB], page 156.

`-archiveprocedurals`
Only meaningful with `-catrib`. Turns any procedurals found in the input RIB into archives named archive0.rib, archive1.rib, etc.

`-maxmessages n`
Print out at most '`n`' warnings and errors to the console. For example, setting '`n`' to five will only show the first five warnings *and* the first five errors.

`-filtermessages m`
Filter out the messages specified in the comma separated list *m*. Note that this simply sets the message filtering option so it can be overriden by a RIB file. See also [filtering messages], page 41.

`-rif plug-in name`
Loads the specified Ri plug-in filter. 3DELIGHT will look for the file in the current directory as well as in paths specified in `DL_RESOURCES_PATH` and `DL_PROCEDURALS_PATH`. More about Ri plug-in filters in Section 7.17 [Using Ri Plug-in Filters], page 162.

`-rifargs ... parameters ... -rifend`
Encloses a set of parameters that are meant for the previously declared Ri plug-in filters (loaded using `-rif`). This set of parameters is not interpreted in any way by `renderdl` and is directly passed to the Ri plug-in filter.

`-beep` Beeps when all RIBs are rendered.

`-beeps` Beeps after *each* RIB is rendered.

`-v` Prints *3Delight*'s version number and name.

`-h` Prints a short help screen.

Return values

The `renderdl` executable can return one of the following values.

`0` No error.

`1` Bad combination of parameters. An error message will explain why.

`99` Option "licensing" "waitforlicense" 0 was used and no license was available.

`255` The RIB file specified on the command line could not be read.

The `.renderdl` File

When started, `renderdl` immediately looks for an initialization file named `.renderdl`. This file is a normal RIB which may contain any standard RIB command, enabling the user to specifiy whatever default options are needed for rendering, such as variable declarations, standard screen format, performance options, etc. The locations in which *3Delight* looks for this file are (in order):

1. The directory of the first RIB file passed on the command line.

2. The current working directory.
3. The user's home directory[2].
4. The directory pointed to by the `DELIGHT` environment variable.

Only the first `.renderdl` file found is loaded. Loading `.renderdl` may be bypassed using the `-noinit` option (see [renderdl options], page 8).

3.2 Using the Shader Compiler - `shaderdl`

3Delight shaders are written in RenderMan Shading Language (SL). *3Delight* also supports most of the common extensions to this language. The shader compiler produces either an object-code shader (a DSO on IRIX and Linux, a DLL on Windows) or a byte-code shader (commonly called interpreted shader). By default, the compiler produces byte-code shaders. If you have a C++ compiler installed on your computer, you can ask `shaderdl` to produce object-code shaders by setting the `--dso` command line switch.

The main advantage of byte-code (interpreted) shaders is their platform independentness: they can be used on IRIX, MacOSX, Linux or Windows computers. This is not the case with object-code shaders. Object-code shaders, however, can be faster than byte-code shaders. We suggest you produce object-code for those shaders that are computationally intensive, such as volume shaders.

To compile a shader, use the command:

```
% shaderdl shader.sl
```

As you may have noticed, you do not have to specify an output file name to `shaderdl`. The output file name is the name of the shader (the identifier following the keyword `surface`, `displacement`, `light`, or `volume` in the SL program) followed by the suffix `.sdl`.

Some renderers' shader compilers append the name of the platform at the end of object-code shader's name. *3Delight*'s shader compiler does not use this convention. An object-code shader always has the same name, regardless of the platform for which it is compiled. We recommend that you keep shaders compiled for different platforms in separate directories (this of course is not necessary for interpreted shaders).

Command Line Options

As with every compiler, `shaderdl` understands a set of command-line options that control the compilation process. These are specified before the input file name:

```
% shaderdl [options] [shader files or directories]
```

There is no need to specify an output file name to `shaderdl` since it is automatically set to the name of the shader[3] followed by the suffix `.sdl`.

The `SHADERDL_OPTIONS` environment variable is read for any option before reading those on the command line. The `-v` and `-h` options do not work if set in the variable.

Valid command line options are:

`-o <directory>`
: Specifies the output file (and possibly the destination directory) for compiled shaders. If the specified file doesn't have the `.sdl` extension, it is not recognized by the renderer.

`-d <directory>`
: Specifies the destination directory for compiled shaders. The default is the current working directory.

[2] As specified by the `HOME` environment variable.

[3] The identifier following the keyword `surface`, `displacement`, `light`, `volume` or `imager` in the SL code.

`--dso` Generates object-code shaders. To use this option, you must have a C++ compiler installed. The compilation script included with *3Delight* is configured to work with `CC` on IRIX, with `g++` on Linux and with `Visual C++` on Windows. If you have a different compiler installed, you have to set an environment variable or use the `--compiler` option. For more information, use the `--help-compiler` option.

`--compiler <compiler>`
This enables you to override the default C++ compiler used by `--dso` . Use the `--compiler-help` to see the different compilers available or how to build your own custom C++ compiler command.

`--help-compiler`
Explains how to pick a C++ compiler using the `--compiler` or by setting the `SHADERDL_CC_COMMAND` environment variable.

`--int` Generates byte-code shaders (default).

`--embed-source`
Adds source code into the `.sdl`. This enables automatic recompilation when backward compatibility is broken. See `--recompile-sdl` for a more complete solution.

`--recompile-sdl`
When backward compatibility is broken, providing the `.sdl` was compiled with the `--embed-source` option, it is recompiled each time a render requires it, therefore slowing down the rendering process. This option makes the shader accept `.sdl` files, recompiles them (provided it contains its own source code) and replaces them. If a path is specified instead, all `.sdl` in it are affected. Only byte-code shaders may be updated with this option.

`-O<n>` Specifies the optimization level, from 0 to 3. Refer to [Optimization Levels], page 13.

`-w<n>` Sepcifies warning level:

0. Disables all warnings (not recommended).
1. Logs important warnings only (default).
2. Logs all warnings.

`-I<directory>`
Specifies a directory to search for `#include`'d files.

`-D<symbol>`
Defines a preprocessor symbol. See also [Predefined Macros], page 13.

`-E` Stops after the preprocessing step.

`-c` Stops after the shading language to C++ translation pass.

`--keep-cpp-file`
`--dont-keep-cpp-file`
Specifies whether or not to keep the intermediate files generated by the preprocessor. They are not kept by default.

`--no-array-check`
Turns off array run time bound checking. Enabled by default.

`--use-shadeops`
`--dont-use-shadeops`
Enables [disables] use of shadeops. Enabled by default.

`--searchpath-debug`
: Displays a trace of where shadeops are searched.

`--strict` Enable strict RSL syntax and semantics (see [Strict RSL Syntax], page 13).

`-v`

`--version` Prints the compiler version number and exits.

`-h`

`--help` Prints a help message and exits.

Predefined Macros

`shaderdl` always defines the following macros:

- `DELIGHT`
- A platform specific macro (`LINUX`, `DARWIN` or `WIN32`).
- `DELIGHT_VERSION` to a numeric value of the form 8501 (for version 8.5.1).

Optimization Levels

`shaderdl` is an optimizing compiler: it implements most of the modern optimization techniques and also adds some special SIMD optimizations. Here is the list of optimizations enabled by each `-O` level.

`-O0` There are no optimizations.

`-O1` Enables some basic optimizations that fight against the tendency of SIMD compilers to generate a lot of temporaries and loops:

- Enables SIMD mechanisms.
- Minimizes number of generated temporary variables.
- Reuses temporary variables and SIMD booleans.
- Regroups SIMD loops.

`-O2` Sets some aggressive optimization, this is the recommended level and also the default. It includes:

- Dead code elimination
- Copy Propagation
- Constant Folding
- Arithmetic operations are reused if possible.
- Removes code from loops or moves them outside, when possible.
- Generates extra code to *force* some non-SIMD calls to become SIMD.
- Converts `varying` variables to `uniform` variables when possible.

`-O3` This level enables even more aggressive optimizations. We tend to put newly introduced optimizations in `-O3` and move them to `-O2` after they have been thoroughly tested. Experimental optimizations, which do not produce faster code in all cases, are also put here.

- Disables array bounds checking.
- Removes empty and useless loops, even if they are infinite.
- Widens the scope of different optimizations. This makes them more thorough at a cost of extra compilation time.

Strict RSL Syntax

The `--strict` option to `shaderdl` disables some non-standard syntax and semantics. By default `shaderdl` is quite permissive in order to be compatible with other RSL shader compilers. By using this option the shader writer can write shaders that are more in-line with the good practices of RSL programming. Follows a list of features that are disabled by the `--strict` option.

- Error instead of a wrning for varying strings. Writing the following code would produce an error:

```
varying string bad = "something";
```

- Error when accessing pre-defined variables without the `extern` keyword. The following surface will print an error in strict mode (only a warning by default):

```
void test()
{
    P = 0;
}

surface strict_test() { test(); }
```

  Note that for RSL 2.0 shaders there is no need to use the `extern` keyword since predefined variables are considered as class members.

- Error when `extern` declaration has detail mismatch. For example,

```
void test()
{
    extern uniform point P;
    P = 0;
}

surface strict_test() { test(); }
```

  Will print an error since *P* should be declared as `varying point`.

- Error when assigning to a non-output function parameter. For example,

```
void test( float in ) { in = 0; }

surface strict_test()
{
    float tmp = 0;
    test( tmp );
}
```

  Will produce and error because *in* parameter to the `test` function is not declared as `output`.

- Error on ambiguous function call. For example,

```
void test() { }
float test() { }

surface strict_test() { test( ); }
```

  Will produce an error because it is not clear which function to call.

3.3 Using the Texture Optimizer - `tdlmake`

`tdlmake` preprocesses TIFF, PNG, JPEG, GIF, IFF[4], SGI, PIC[5], PSD[6], TGA[7], "bake," RADIANCE and *OpenEXR* files to convert them into an efficient texture format suitable to *3Delight*. It can also convert a `zfile` into a shadow map. We recommend running `tdlmake` on all textures before rendering, for two reasons:

- `tdlmake` creates a mipmapped version of the original texture, allowing *3Delight* to produce nicer images more efficiently.
- *3Delight* employs a caching mechanism for texture data which works well with tiled images. Using raw (striped) non-converted TIFFs may degrade overall performance.

Note that a converted file is a normal TIFF that can be viewed with any image viewer. We suggest using a `.tdl` extension for *3Delight* texture files.

Command Line Options

`tdlmake` is invoked by specifying at least two file names and an optional set of command-line switches:

```
% tdlmake [options] input.tif [input2.tif ... input6.tif] output.tif
```

Valid options are:

`-envlatl` Generates a latitude-longitude environment map.

`-envcube` Generates a cubic environment map. Needs six images in input, ordered as follows: `+x`, `-x`, `+y`, `-y`, `+z`, `-z`.

`-twofish` Generates a cubic environment map from two images taken with a circular fisheye lens. The first image is taken to be `+z` and the second `-z`.

`-lightprobe`
Generates a cubic environment map from a 360 degree fov lightprobe.

`-dirtex` Creates a "directory texture". A directory texture (or "dirtex" in short) consists of a series of TIFF files save in a directory, along with an index file. The main advantage of such textures is to have a finer granularity of data transfer when using the network cache. Please refer to Section 7.14 [Network Cache], page 154 for more about directory textures.

`-skymap s` Generates a latlong environment map that is a representation of a sky as seen from the given position and the given time. The parameter to `-skypmap` is a comma separated list of six arguments specifying: '`latitude,longitude,timezone,julian day,time,turbidity`'. Ranges for each parameter are:

'`latitude`'
'`longitude`'
These two coordinates specify the location of the observer, in degrees.

'`timezone`' This is the standard time zone indicator. Starting at 0 and ending at 12.

'`Julian day`'
Day of the year, from 0 to 365.

'`time`' Time of the day. '`6:30 PM`' is '`18.5`'.

[4] Such as those used by *Maya* software.
[5] Such as those used by *SoftImage* software.
[6] Such as those used by *Abobe Photoshop* software.
[7] Native format of *Truevision Inc.*

'turbidity'
This is a measure of "haziness" ranging from 2 to 10. A value of 2 will give a clear day and a value of 10 will give a very hazy skylight.

An example command line would be:

```
% tdlmake -skymap 42,74,5,10,12.5,4 daylight.tdl
```

`-shadow` Generates a shadowmap from a zfile. When generating a shadowmap, only the `-c-` option is functional.

`-lzw` Compresses output texture using LZW algorithm. This option is enabled by default since compressed textures take much less space and there is no noticeable speed penality when accessing them.

`-deflate` Compresses output texture using the Deflate algorithm. Has a better compression ration than LZW.

`-packbits` Compresses output texture using Apple's PackBits algorithm. Compression ratio is not as good as with LZW or Deflate but decompression is very fast.

`-c-` Does not compress output texture.

`-separateplanes`
Writes the output TIFF with separate color planes instead of interleaved colors. This may help compatibility with some software packages.

`-fov n` Specifies a field of view, in degrees, for cubic environment maps. Default is '90' degrees.

`-mode <black|clamp|periodic>`
`-smode <black|clamp|periodic>`
`-tmode <black|clamp|periodic>`
Specifies what action should be taken when accessing a texture (using `texture()` or `environment()`) outside its defined parametric range (s, $t = [0..1]$).

'black' Sets texture to black outside its parametric range.

'clamp' Extends texture's borders to infinity.

'periodic' Tiles texture infinitely.

`-smode` and `-tmode` specify the wrapping modes of the texture in s or t only. Default mode is '`clamp`' for normal textures and '`periodic`' for latitude-longitude environment maps. Note that while the effects of this option are noticeable mostly when using `texture()` or `environment()` from inside a shader, it also slightly changes how the mipmaps of the texture are computed.

`-filter <box|triangle|gaussian|catmull-rom|bessel|mitchell|sinc>`
Specifies a downsampling filter to use when creating mipmap levels. The default filter is '`sinc`'. All supported filters, along with their `filterwidth` and window defaults, are shown in Table 3.1.

filter	filterwidth	window	Comment
box	1.0	–	This filter tends to blur textures. Use only if texture generation speed is an issue.
triangle	2.0	–	`filterwidth` larger than 2.0 is unnecessary.
gaussian	2.50	–	A good filter that might produce slightly blurry results, not as much as the box filter though.
catmull-rom	4.0	–	A better filter (producing sharper textures).
bessel	6.47660	'`lanczos`'	Filter width chosen in order to include 2 roots.
sinc	8.0	'`lanczos`'	We recommend this high quality filter as the best choice. Although it can produce some ringing artifacts on some textures. Using a `filterwidth` smaller than 4 is not recommended.

Table 3.1: `tdlmake` filters.

`-window <lanczos|hamming|hann|blackman>`
Applies a windowing function to '`bessel`' and '`sinc`' filters (which are infinite in width) to achieve a softer cut at the filter's support boundaries. Possible windowing functions are '`lanczos`', '`hamming`', '`hann`' and '`blackman`'. By default, no function is applied.

`-filterwidth n`
Overrides the default filter width in s and t.

Important: Filter's width is the *diameter* of the filter and *not* the radius.

`-sfilterwidth n`
`-tfilterwidth n`
Overrides the default filter width in s or t, respectively.

`-blur n` Blurs or sharpens the output image. Values larger than '`1.0`' make the image more blurry and values smaller than '`1.0`' give sharper looking textures. This function works by scaling the filter function by the specified value. This is *not* the same thing as scaling `filterwidth`. Default is '`1.0`'.

`-scale n` Scales down the output image. Values larger than '`1.0`' or smaller than '`0.0`' are not accepted. Default is '`1.0`'.

`-preview n` Adds a preview image to the texture. The preview is the first image in the TIFF file and is written in a format which is readable by most software, unlike the mipmaps *3Delight* uses. The argument is interpreted as an absolute size for the preview if greater than 1 and as a relative size otherwise.

`-preview8 n`
Does the same as -preview except that the preview image is always 8-bit. This makes it smaller and potentially more compatible with some software.

`-quality <low|medium|high>`
Controls mipmap downsampling strategy: when using '`low`', each mipmap level is created from the previous one. At '`medium`' quality, each level is created from the 2nd previous level. At '`high`' quality, each level is created from up to the 4th previous level. The default '`medium`' setting is more than enough for most applications.

`-flips`
`-flipt` Flips the image horizontally or vertically, respectively.

`-flipst` Flips the image in both directions.

`-bakeres n` Specifies the output image resolution when converting `.bake` files to a texture map. Note that a `.bake` file can be of any resolution but is of the same width and height.

`-forcecolor`
Force greyscale input to be expanded to RGB.

`-gamma n` Indicates the gamma of the input image. This allows `tdlmake` and *3Delight* to convert the texture to a gamma of 1.0 before performing computations on it.

`-rgbagamma n n n n`
Indicates the gamma of each red, green, blue and alpha channel of the input image. This allows `tdlmake` and *3Delight* to convert the texture to a gamma of 1.0 before performing computations on it.

`-colorspace <linear|srgb|BT.709>`
Indicates the color space of the input image (linear, sRGB or BT.709). This allows `tdlmake` and *3Delight* to convert the texture to a linear space before performing computations on it.

`-byte or -ubyte`
Forces the output to be 8-bit unsigned data.

`-sbyte` Forces the output to be 8-bit signed data.

`-ushort` Forces the output to be 16-bit unsigned data.

`-short or -sshort`
Forces the output to be 16-bit signed data.

`-float` Forces the output to be floating point data.

`-pixelaspectratio`
Specifies the aspect ratio of the pixels. This can be retrieved with `textureinfo()` (see [textureinfo shadeop], page 113).

`-imageaspectratio`
Specifies the aspect ratio of the image. A pixel aspect ratio is computed from this value.

`-newer` If the destination texture already exists, regenerates it only if one of the input textures is newer than the destination.

`-nomipmap` By default, `tdlmake` creates mipmapped textures that are suitable for efficient and high quality texture mapping. This option turns this feature off and is not recommended.

`-progress` Shows texture creation progress, only useful for really large textures (or really slow computers!).

`-v` Prints version and copyright information.

`-h` Shows help text.

Supported Input Formats

`tdlmake` supports most of the common formats used in a production environment:

TIFF `tdlmake` supports stripped or tiled TIFFs with 1 to 6 channels. Suppported data types are:

- 1, 4, 8, 16 or 32 bits of unsigned integer data
- 8, 16 or 32 bits of signed integer data
- 32 bits of floating point data

Unsigned single channel images (grayscale or b&w) can have either MINISBLACK or MINISWHITE photometric. Either way, the output file (`.tdl`) is MINISBLACK. Signed files are required to have either RGB or MINISBLACK photometric. Files with less than 8 bits per sample are promoted to 8 bits per sample in the output. Otherwise, the output format is the same as the input. `LogLuv` encoded TIFFs are supported and are kept in their `LogLuv` form when processed. TIFFs with JPEG compression are not supported.

IFF files from Maya software
Both RGB and RGBA formats are supported.

PNG files are supported (all formats and bit depths).

SGI files from Maya software
GRAYSCALE, GRAYSCALE+ALPHA, RGB and RGBA formats are supported. *Note that 16 bits images are not supported.*

PIC files from SoftImage software
Both RGB and RGBA formats are supported.

PSD files from Abobe Photoshop software
Both GRAYSCALE and GRAYSCALE+ALPHA formats without layers are supported; both RGB and RGBA formats with or without layers are supported.

TGA files developped by Truevision Inc.
GRAYSCALE, GRAYSCALE+ALPHA, RGB, RGBA, INDEXED and INDEXED+ALPHA formats are supported.

`.bake` *files* These files are generated by the `bake` (see [bake shadeop], page 110) shadeop and are converted to a floating point, mipmapped, TIFF file. Output resolution can be specified using the `-bakeres` option.

JPEG 8 bit (grayscale) and 24 bit JPEGs are supported.

Radiance Picture Format
Files produced by Radiance[8] are supported. Both XYZE and RGBE encoding modes are recognized by `tdlmake`. Run-length encoded files are also supported. Note that there is a restriction on image orientation: only -Y +X orientation is correctly recognized, any other orientation is reverted to -Y +X. This is not a major problem since it can be corrected by using `-flips` and `-flipt` options (see [tdlmake options], page 15).

OpenEXR ILM's OpenEXR format is supported by `tdlmake` with the restriction that images should be in RGB or RGBA format[9].

zfile zfiles are only used to produce shadow maps.

For cube environment maps (`-envcube`), all six images are required to be of the same file format.

[8] Refer to http://floyd.lbl.gov/radiance/.
[9] Refer to http://www.openexr.org.

Quality and Performance

As with all *3Delight* tools, `tdlmake` was designed to fulfill the tough demands of a production environment. In this case, special attention was given to filtering quality and memory usage, without penalizing execution speed. Among other things:

- All internal filtering algorithms are executed in floating point to preserve as much accuracy as possible.
- An efficient caching system is used to keep memory usage footprint *very* low. `tdlmake` was tested on textures as large as 16K x 16K (1 Gigabyte of disk space) and accomplished its work using less than 12megs of memory.
- The filtering process has been designed to have a linear cost increase relative to `-filterwidth`, unlike some software in which the cost is quadratic. For example, `-filterwidth 8` runs twice as slow as `-filterwidth 4`.

Working with Large Textures

As mentioned, `tdlmake` has been designed to adapt very well to huge textures. One exception is compressed TIFFs that have a large "rows per strip" value. Here is an example output of `tiffinfo` on a large texture file that can cause problems for `tdlmake`:

```
% tiffinfo earth.tif
  Image Width: 43200 Image Length: 21600
  Resolution: 72, 72 (unitless)
  Bits/Sample: 8
  Compression Scheme: AdobeDeflate
  Photometric Interpretation: RGB color
  FillOrder: msb-to-lsb
  Software: "ImageMagick 5.5.7 07/22/03 Q16 http://www.imagemagick.org"
  Document Name: "earth.tif"
  Orientation: row 0 top, col 0 lhs
  Samples/Pixel: 3
  Rows/Strip: 21600
  Planar Configuration: single image plane
  Predictor: horizontal differencing 2 (0x2)
```

There is only one strip, made of 21600 (total image height) rows. This means that accessing any scanline in this TIFF forces the TIFF reading library to uncompress the entire file[10]. In order to lower memory usage for such large files, it is suggested that a lower "rows per strip" count be used. Typically, 16 or 32 scanlines is a good choice. `tdlmake` prints a warning if it encounters a file that has the aforementioned "problem":

```
tdlmake: warning, reading texture 'earth.tif' may take a large
tdlmake: amount of memory. Please refer to user's manual if you are
tdlmake: unable to convert this file.
```

Examples

Here are some examples using `tdlmake` on the command line:

To create a texture named `grid.tdl` from a TIFF named `grid.tif` using a gaussian downsampling filter of width 4:

```
% tdlmake -filter gaussian -filterwidth 4 grid.tif grid.tdl
```

[10] Compression is performed on strips in TIFFs, not scanlines.

To create a cubic environment map in which all cube sides were rendered using 90 degrees field of view:

```
% tdlmake -fov 90 -envcube \
  in1.tif in2.tif in3.tif in4.tif in5.tif in6.tif \
  envmap.tdl
```

or (won't work in a DOS shell) :

```
% tdlmake -fov 90 -envcube in?.tif envmap.tdl
```

To create a texture using the high quality downsampling mode and show progress while doing so:

```
% tdlmake -progress -quality high grid.tif grid.tdl
```

To create a shadow map from a zfile (Section 8.6 [dspyzfile], page 170):

```
% tdlmake -shadow data.z shadowmap.tdl
```

3.4 Using `dsm2tif` to Visualize DSMs

`dsm2tif` is a utility to convert *3Delight*'s proprietary deep shadow map files into viewable TIFF files. A DSM contains visibility data for any given depth in a scene, so one single 2D image (e.g. a TIFF) cannot describe all the information provided by a DSM; that is why command line parameters are provided to specify at which depth the evaluation occurs.

`-z depth` Specifies a relative depth at which the deep shadow map is evaluated. "Relative" means that each pixel in the image is evaluated at its own depth. It is computed as follows: $(pZmax - pZmin) * depth + pZmin$. $pZmin$ and $pZmax$ being the depths of the closest and the furthest features present in the **pixel**. `-z 1` is the default, which shows the amount of light that passes through to infinity at each pixel.

`-Z depth` Specifies an absolute depth (range: [0..1]) to evaluate the deep shadow map. "Absolute" means that all pixels in the output image are evaluated at the same depth, computed as follows: $(Zmax - Zmin) * depth + Zmin$. $Zmin$ and $Zmax$ being the depths of the closest and the furthest features in the DSM.

`-bias n` Specifies a shadow bias for deep shadow map evaluation. This is needed to avoid self-occlusion problems. If the produced TIFF contains noisy areas, consider increasing this parameter. Default is 0.015.

`-opacity` By default, `dsm2tif` assigns a "visibility" value to each pixel in the output TIFF, specifying this option assigns "opacity" values to each pixel ($opacity = 1 - visibility$). This means that dark areas in the image indicate that light passes through unnoccluded.

`-mipmap n` This options specifies which mipmap level is converted. Note that mipmapping can be disabled by an option given to the display driver (see Section 8.8 [dspy_dsm], page 171).

`-8`
`-16`
`-32`
`-float` Chooses data format for the output TIFF. Default is 8 bits per channel.

`-lzw`
`-deflate`
`-packbits`
`-logluv` Selects a compression scheme. Default is '`lzw`'. `-logluv` is only supported with `-float` data type.

`-v` Prints version and copyright information.

`-h` Shows help text.

3.5 Using `hdri2tif` on High Dynamic Range Images

`hdri2tif` uses a range compression algorithm[11] to convert high dynamic range images into displayable, low dynamic range TIFFs. All the important HDRI formats are recognized in input:

1. Floating point TIFFs as well as LogLuv encoded TIFFs.
2. Radiance files, uncompressed or RLE packed.
3. ILM's OpenEXR files.

Note that all formats must have 3 or 4 channels. The fourth channel is considered to be alpha information and stays untouched in the output image.

`hdri2tif` accepts the following options:

`-middle-gray n`
`-key n` Sets the value of the middle-gray in the image, default is 0.18. Range is [0..1]. Specifying a value of 0 enables a histogram based automatic estimation (one can use `-verbose` to display the computed key value).

`-simple` By default, `hdri2tif` uses a "dodging-and-burning" algorithm which is relatively costly[12]. This option uses a simpler algorithm that is well suited for images with a medium dynamic range (lower than 10 "zones"). Note that this method may produce bad results on very high dynamic range images.

`-white n` Sets the luminance value in the image that is mapped to pure white (1.0 in the low dynamic range output image). Setting this parameter to a smaller value produces more "burning". The default action is to set the white point to the maximum luminance in the image, which removes burning. This option is only meaningful when paired with the `-simple` option because the default algorithm automatically computes the white point. Specifying a value of 0 enables a histogram based automatic estimation (one can use `-verbose` to display the computed value).

`-gamma n` Specifies a gamma correction to be applied on the produced image. Gamma correction is performed *after* the range compression algorithm.

`-nthreads n`
`hdri2tif` can run in a multi-threaded mode for increased performance. This option specifies how many threads to use. This option has *no* effect when `-simple` is used.

`-verbose` Displays information while processing the image.

`-h` Displays a brief help text.

The following options may be used to further control the "dodging-and-burning" algorithm. Modifying these options is not recommended.

`-nscales n` Specifies the number of gaussian convolutions to use when computing luminance information for each pixel. More gaussians mean a more precise result but also a slower computation. Default is 8.

`-sharpness n`
Sets the sharpness parameter. Higher values mean sharper images. Default is 8.

The following options control the compression applied to the output:

---

[11] *Erik Reinhard, Mike Stark, Peter Shirley and Jim Ferwerda, 'Photographic Tone Reproduction for Digital Images', ACM Transactions on Graphics, 21(3), pp 267–276, July 2002 (Proceedings of SIGGRAPH 2002).*

[12] Implies frequency domain computations.

`-lzw`
`-deflate`
`-packbits`
`-none` Selects compression method to use for output TIFF. `-lzw` is enabled by default.

3.6 Using `shaderinfo` to Interrogate Shaders

`shaderinfo` is a utility to interrogate shaders compiled with `shaderdl`. It could be used as a quick way to see what parameters are needed for a particular shader or it could be used by third party software to build a graphical interface to manipulate shader parameters.

The general command line syntax is:

```
shaderinfo [options] shader1 shader2 ... shadern
```

And the following list of options is accepted:

`-d` Output shader's parameters in a RIB format (see example below)

`-t` Output shader's parameters in an easy to parse tabular format

`-a` Output shader's annotations, if available

`--source` Output shader's source code if the shader was compiled with the `--embed-source` option as described in [shaderdl options], page 11

`--methods` Output the name of all the methods in the shader.

`-v` Output `shaderinfo` version

`-h` Print a help screen

A sample run on the `matte` shader gives the following output[13]:

```
% shaderinfo matte
surface "matte"
    "Ka" "uniform float"
                Default value: 1
    "Kd" "uniform float"
                Default value: 1
```

If the output is intended for parsing, it might be better to use the `-t` option which gives the parameters in the following format:

`Line 1` <displacement|volume|surface|lightsource|imager>

`Line 2` <shadername>

`Line 3` A number '`n`' specifying the total number of shader parameters

`Lines 4 to n+4`
'n' lines of output, one for each parameter in the shader. Each line is a comma separated list of fields. Fields are, in order: name, storage, class, type, default space, array length, default value. If there is no default space '`<none>`' will be used.

Interrogating the `matte` shader with the `-t` option gives the following output:

```
% shaderinfo -t matte
surface matte
2
Ka,parameter,uniform,float,<none>,0,1
```

[13] To find a shader, `shaderinfo` uses the search paths specified by the `DL_SHADERS_PATH` (or `DL_RESOURCES_PATH`) environment variable.

```
Kd,parameter,uniform,float,<none>,0,1
```

Shader annotation keys are listable using `shaderinfo` but this feature is better described in Section 10.4 [Attaching annotations to shaders], page 245.

3.7 Using `ribshrink`

`ribshrink` is a utility to factorize archives from a series of RIB files, meaning that parts that are common to all RIBs are extracted and put into archives. This tool could dramatically reduce disk usage for automatically generated RIB sequences.

`ribshrink` accepts the following options:

`-d dir` Specify an output directory.

`-prefix dir`
: Specify a prefix for archive names.

`-overwrite`
: Allows overwriting of existing files.

`-archiveminsize n`
: Minimal size (in bytes) of extracted archives. The size is measured before compression so the actual archives can be smaller.

`-binary[-]`
: Enable or disable binary output (default is on).

`-gzip[-]` Enable or disable compression (default is on).

`-progress` Write a trace of what is being done.

`-v` Output version.

`-h` Print this help.

No directories are created by ribshrink. If a directory is specified with `-d` or `-prefix`, it must exist beforehand.

3.8 Using `ribdepends`

`ribdepends` is a utility to list dependencies in a given RIB file as well as make *site independant* RIB packages.

The general command line syntax is:

```
ribdepends [options] [file1 file2 ... filen]
```

And the following options are accepted:

`-package dir`
: Copies specified RIB files and all their dependencies in '`dir`' directory. Paths inside RIB files are modified to point into correct locations in the created package. Conflicting file names are changed in the package directory and the RIB files. Note that this copies all textures, shaders and archives necessary for the RIB to render; this could result in very large pacakges.

`-scale s` If used with the `-package` option, packaged textures are scaled whenever possible to save disk space. Only scaling down is supported. This means that `s` must be smaller than 1. Deep shadow maps and cube environment maps are not scaled.

`-binary` If used with the `-package` option, the copied RIB files will be binary encoded.

`-gzip` If used with the `-package` option, the copied RIB files will be gzipped.

`-noshaders`
Causes ribdepends not to search for shaders.

`-notextures`
Causes ribdepends not to search for texture files.

`-nooutputs`
Causes ribdepends not to process the paths of output files (RiDisplay, RiMakeTexture, etc).

`-noprocedurals`
Causes ribdepends not to search for `DynamicLoad` and `RunProgram` procedurals. Does not affect the `DelayedReadArchive` procedural.

`-noinit` Do not read `.renderdl` file.

`-v` Print version.

`-h` Print this help.

The default behavior (when no options are given) is to list resources required for the proper rendering of the specified RIB file(s) (or standard input when no files are specified). Each dependency is prefixed by a character indicating its type:

`[t]` Texture or shadowmap dependency. 3D textures (point cloud files) are also considered.

`[s]` Shader dependency.

`[x]` Procedural dependency

`[u]` Unresolved dependency (could not find file on filesystem).

`[o]` Output file (such as a file specified to `RiDisplay` or `RiMakeTexture`).

`[c]` Conflicting dependencies that have the same name but different paths. These might be renamed in the package when using the `-package` option.

An example run on the example found in `$DELIGHT/examples/outputvariables` gives the following output:

```
% ribdepends toonStill.rib
[s] /users1/aghiles/software/3delight/shaders/ambientlight.sdl
[s] /users1/aghiles/software/3delight/shaders/pointlight.sdl
[u] matte_output.sdl
[o] toonStill-normals.tif
[o] toonStill-points.tif
[o] toonStill.tif
```

We can see that the `toonStill.rib` depends on two shaders and produces three output files. `matte_output.sdl` is listed as undefined because it has not been compiled.

Note that `RiDelayedReadArchive` is treated like a `RiReadArchive`. If procedurals are to be expanded instead, use `renderdl` with the `-callprocedurals` option as in:

```
% renderdl -catrib -callprocedurals somefile.rib | ribdepends
```

3.9 Using `ptc2brick`

`ptc2brick` converts one or more point cloud files into a single brick map.

The general command line syntax is:

```
ptc2brick [options] <in.ptc> [more input files] <out.bkm>
```

The available options are:

`-maxdepth d`
: Specifies a maximum depth '`d`' for the brick map tree. Smaller depths will give smaller files. The default maximum depth is 15.

`-maxerror e`
: Specifies maximum tolerated error '`e`' in the brick map. This is a trade-off between accuracy and file size.

`-radiusscale s`
: Specifies a scale '`s`' that will be applied to all points' radii. For '`s`' higher than 1, blurrier and smaller brick maps are created. Specifying a value smaller than 1 is not recommended and will create large and inefficient brick maps.

`-newer`
: Only builds the brick map if one of the input files is newer than the output. This option has no effect if the output file doesn't exist.

`-h`
: Prints a help screen.

A brick map is a 3D texture file format that is suitable for anti-aliased lookups. Please refer to Section 7.6 [Baking], page 141, [bake3d shadeop], page 110 and [texture3d shadeop], page 111 for further discussions.

3.10 Using `ptcview`

`ptcview` displays a point cloud file in a window[14]. The generl command line syntax is:

```
ptcview [-info] [-h] [file.ptc]
```

When started with an empty command line, `ptcview` will start with an empty window. A point cloud can then be loaded through the '`File`' menu item. When provided with a file, this utility will proceed with the load prior to display.

Here is the list of supported functionalities:

1. All channels contained in the point cloud can be displayed through the '`View -> Channels`' menu item.
2. Point normals can be displayed through the '`View -> Normals`' menu item.
3. The current view port can be copied into the clipboard as a 2D image through the '`Edit -> Copy`' menu item.

When given the `-info` option, `ptcview` will only print useful information about the given point cloud file. An example output looks like this:

[14] Does not work on brick maps.

```
Number of points: 462694
Number of channels: 1
Channel number 1 - Name: _radiosity - Type: color
Bounding box: ( -5.71307 -6.24433 -3.51055 ), ( 1.55966 1.02841 3.76218 )
World to eye matrix:
          1           0           0           0
          0           1           0           0
          0           0           1           5
          0           0           0           1
World to ndc matrix:
    3.73205           0           0           0
          0     3.73205           0           0
          0           0           1           5
          0           0           1           5
Format: 400 400 1
```

3.11 Using `ptcmerge`

`ptcmerge` is a simple utility to merge point cloud files (see Section 7.6 [Baking], page 141). The general command line syntax is:

```
ptcmerge [-h] <in.ptc> [other input files] <out.ptc>
```

Input files can contain different data channels and in different order: `ptcmerge` will only merge channels present in *all* input files.

4 Integration with Third Party Software

4.1 Autodesk's *Maya*

The package comes with a full-featured, production ready, *Maya* plug-in: *3Delight For Maya*. Please refer to:

3Delight For Maya Technical Specifications
: For a brief discussion about plug-in's features.

3Delight For Maya User's Manual
: For a length description and rendering guidelines.

4.2 Softimage's *XSI*

We provide a separate product for *XSI* integration: *3Delight For XSI*. Please contact us at `info@3delight.com` for further information.

4.3 Side Effects Software's *Houdini*

The following *Houdini* integration tips work with later *Houdini* 9 packages.

- Integration in *Houdini* is substantially improved by letting *Houdini* know that *3Delight* is the preferred RenderMan renderer. This is done by setting the `HOUDINI_DEFAULT_RIB_RENDERER` environment variable to '`3Delight8.5`' for Houdini versions up to 10.0.473, or to '`3Delight9.0`' for Houdini versions after 10.0.473. This variable will instruct *Houdini* to use *3Delight* for shader compilation and for rendering, and also provides relevant default properties for the RIB output driver.
- *Houdini* comes with a '`houdini`' display driver that can be used with *3Delight*. This display driver is installed using the `proto_install` *Houdini* command.
- *3Delight* provides a digital asset library of all the shaders included in the package. Once this file is imported in *Houdini*, all *3Delight* shaders will exist as SHOP nodes. The digital asset library file can be found in the '`shaders`' directory of the *3Delight* installation (`$DELIGHT/shaders`).
- Custom shaders can also be packaged in a digital asset library file which can then be imported in *Houdini*. The `sdl2otl.py` Python script that comes with *Houdini* will parse the output of `shaderinfo` to create the digital asset library file from any compiled shader file.
- While the RIB output driver comes with the most common rendering parameters, several more can be added through the parameter pane's *tool* menu. It is possible to add new parameters, remove unwanted ones and save the custom parameter set as the default set to be used for future RIB output drivers. This is done through the *Save as Permanent Default* option of the parameters pane's *Tools* menu.
- The same mechanism can be used to modifiy any object's rendering properties. Again, it is possible to define a new default set of attributes, better suited for rendering with *3Delight* than the default properties that *Houdini* creates by using the *Save as Permanent Default* option of the parameters pane's *Tools* menu.

5 3Delight and RenderMan

There are two ways to describe a scene to *3Delight*: the first is by using the RenderMan Application Programming Interface (API) and the second is by using the RenderMan Interface Bytestream (RIB) files. The RenderMan API and the RIB format are well described in *RenderMan Interface Version 3.2*. This document is available electronically at `https://renderman.pixar.com/products/rispec/index.htm`.

A RIB file is a set of commands meant to be interpreted by a RIB reader such as `renderdl`. There is almost a one to one correspondence between the API calls and RIB commands.

For in depth documentation about RenderMan and the RenderMan API, consult the following two classics:

- *The RenderMan Companion*. Steve UPSTILL. Addison Wesley.
- *Advanced RenderMan: Creating CGI for Motion Pictures*. Larry GRITZ and Anthony A. APODACA. Morgan Kauffman.

Check `http://www.3delight.com/links.htm` for a direct link to those references.

5.1 Options

Options are parameters of the graphic state that apply to the entire scene. All options are saved at each `RiFrameBegin` and restored at the corresponding `RiFrameEnd` call. It is important to set these options *before* the `RiWorldBegin`...`RiWorldEnd` block since it is a RenderMan assumption that rendering may begin any time after `RiWorldBegin`; all options must be fixed by then. Also, and contrary to transforms, the order in which options appear is not important.

The RenderMan API declares a standard set of options (which are used to control camera and image parameters) and opens a door to implementation specific options through the `RiOption` call. *3Delight* supports all of the standard options and has some specific calls; both groups are described in the following sections. For added flexibility, many default option values are re-definable through a special initialization file as described in Section 5.5 [configuration file], page 59.

5.1.1 Image and Camera Options

All the standard options are supported and are listed in Table 5.2 along with their default values. Some of those options have an extended behavior in *3Delight*:

`RiDisplay` Since the renderer uses floating point numbers internally, the colors are exposed and quantized using the values specified with the `RiExposure` and `RiQuantize` calls. However, one may want to expose and quantize colors differently for each `RiDisplay`. This is made possible by specifying exposure, quantization and dithering values directly to `RiDisplay`. Other settings can also be overridden for each display; see Table 5.1 for the full list. An extended `RiDisplay` command would look like:

```
Display "file" "driver" "variable"
    "exposure" [gain gamma]
    "quantize" [zero one min max]
    "dither" [amplitude]
    "filter" "filtername"
    "filterwidth" [xwidth ywidth]
```

Note that there are four values given to `quantize` compared to three when calling `RiQuantize`. This extended syntax allows specifying a value for *zero* (black), whereas `RiQuantize` assumes this value to be '`0`'. Using this fourth parameter, colors are quantized using the following formula:

```
value = round(zero + value * (one - zero) + dithervalue)
value = clamp(value, min, max)
```

The "filter" parameter accepts all the same filters as RiPixelFilter but also "zmin" and "zmax". These are special filters as they also affect the compositing of visible surfaces: only the frontmost surface with an opacity exceeding that specified by the zthreshold option is retained (see [zthreshold], page 38). The samples are then filtered by always picking the value of the one with the smallest z over the filtering area (for zmin, greatest z for zmax). These filters are useful for output variables which should not be composited, such as surface normals or object identifiers.

Parameter	Default Value	Description
"float exposure[2]"	RiExposure	Overrides global exposure.
"float quantize[4]"	RiQuantize	Overrides global quantization.
"float dither"	RiQuantize	Overrides global dithering.
"string filter"	RiPixelFilter	Overrides global filter. Can also be 'zmin' or 'zmax' when outputting the 'z' variable.
"float filterwidth[2]"	RiPixelFilter	Overrides global filter width and height.
"int associatealpha"	1	When set to 0, color is divided by alpha to provide an image with an unassociated alpha channel. With AOVs, an alpha channel should be explicitely added if it is not equal to the default alpha.
"int matte"	1	When set to 0, the output will ignore the matte attribute (as set by RiMatte). Only works for AOVs.
"int exclusive"	0	When set to 1 for an AOV, objects which do not output the variable will be transparent instead of the usual black.
"string subset"	"-"	Allows each display to show only a subset of the scene objects. It has the same semantics as trace subsets. See Section 7.3.4 [Trace Group Membership], page 128.
"string lightcategory"	""	Allows each display to show contribution only from lights which match the given category specification. See [Light Categories], page 74. This feature works only with the raytrace hider. A special "__environment" category is supported and will display only the environment map contribution from the trace() shadeop. Each different value for this parameter will add a little to the render time.

Table 5.1: Implementation specific RiDisplay parameters.

Another useful feature supported by RiDisplay is file name formatting. It uses special codes starting with the special '#' character to indicate where to insert useful information in the file name.

The accepted special codes are:

'#f' Is replaced by current frame number, specified by RiFrameBegin

```
Display "frame#f.tif" "tiff" "rgb"
```

'`#s`' Is replaced by the sequence number. This number is incremented at each `RiFrameBegin`, regardless of the frame number provided. Sequence numbering starts at "1".

'`#n`' Is replaced by the "running sequence number", a counter that is incremented at each `RiWorldBegin`. First value is "1".

'`#d`' Is replaced by the display type. The file name in the following example is set to `image.tiff`:

```
Display "image.#d" "tiff" "rgb"
```

'`#r`' Is replaced by "3delight" (renderer's name).

'`##`' Is replaced by a '`#`'.

Special codes which print numerical values can contain an alignment specifier after the '`#`', as in '`frame#5f.tif`' which would yield '`frame00001.tif`'.

It is also possible to output only a single element of an output variable by using the following syntax:

```
Display "file.tif" "tiff" "color many_colors[6]:2"
```

This would output the element at position 2 in the array of 6 colors named *many_colors*. To compute an alpha channel specific to a given AOV, simply add the standard *alpha* variable:

```
Display "+foreground.tif" "tiff" "color foreground,alpha"
```

`RiHider` `RiHider` supports the '`hidden`' hider as well as the '`raytrace`' and '`photon`' hiders (refer to Section 7.4.2 [Photon Mapping], page 131). The optional parameters are as follows:

```
Hider "hidden|raytrace"
    "integer jitter" [1]
    "integer samplemotion" [1]
    "float aperture[4]" [nsides angle roundness density]
    "integer extrememotiondof" [0]
    "string depthfilter" "min"
    "int maxvpdepth" [-1]
    "float midpointratio" [0.5]
    "int progressive" [0]
    "float samplelock" [1]
```

A description for each hider parameter follows (note that some of the parameters apply only to one particular hider):

`Hider "hidden|raytrace" "jitter" [1]`
Selects the sampling pattern to use. '`0`' uses a regular sampling grid and '`1`' uses a "jittered" grid.

`Hider "hidden|raytrace" "samplemotion" [1]`
Disables motion blur sampling when set to 0. Unlike setting the shutter open and close times to the same value by using `RiShutter`, this method will preserve the *dPdtime* vectors for use in shaders or as output variables.

`Hider "hidden|raytrace" "float aperture[4]" [nsides angle roundness density]`

This feature can be used to specify a polygonal bokeh in both REYES and ray-tracing modes. *nsides* specifies the number of sides (blades) in the aperture and *angle* is used to specify the rotation, in degrees, of the aperture. *roundness* and *density* are not supported yet. Note that if *nsides* is smaller than 3 then the bokeh is round. The default value for nsides is 0 which means that aperture is round.

`Hider "hidden" "extrememotiondof" [0]`

The default algorithm used to sample depth of field and motion blur may cause sampling artifacts (commonly known as *ghosting*) when objects are out of focus *and* moving. Setting this parameter to 1 will use an alternate algorithm which produces higher quality for this particular case. This algorithm is slower so it should be used only if necessary. Values of 2 or greater allow for a speed vs. quality tradeoff, with larger values increasing quality.

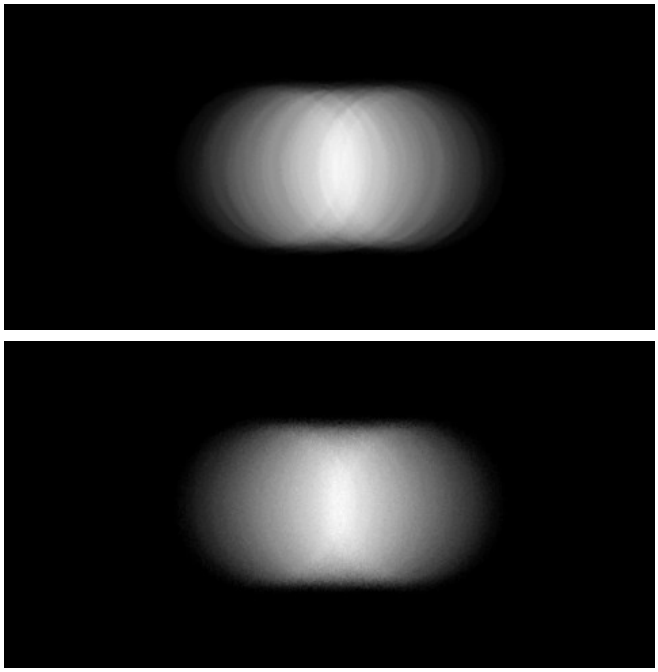


Figure 5.1: An out of focus moving disk rendered with `extrememotiondof` set to 0 (top) and 1 (bottom).

`Hider "hidden" "depthfilter" "`*`filter name`*`"`

Sets a filter type for depth values filtering. The *filter name* may be one of the following:

- ‘`min`’ The renderer takes the minimum z value of all the sub samples in a given pixel.
- ‘`max`’ The renderer takes the maximum z value of all the sub samples in a given pixel.
- ‘`average`’ The renderer averages all sub samples’ z values in a given pixel.

'`midpoint`' For each *sub sample* in a pixel, the renderer takes the average z value of the two closest surfaces (or as dictated by the '`midpointratio`' hider paramter). Depth associated with the pixel is then computed using a *min* filter of these averages. Note that midpoint filtering may slow down renders slightly.

`Hider "hidden" "int maxvpdepth" [-1]`
Sets a limit on the number of visible points for each sub-samples. This means that no more than '`maxvpdepth`' surfaces will be composited in the hider or included in deep shadow map creation. Putting a limit on the number of visible points can accelerate deep shadow map creation for *depth complex* scenes. The default value is '`-1`' which means that there is no limit on visibility lists lengths.

`Hider "hidden" "float midpointratio" [0.5]`
Allows control over the blending of the z values of the first two samples when using the midpoint depthfilter. A value of zero will use the depth of the first sample while a value of one will use the depth of the second sample. Values in between blend both depths.

`Hider "raytrace" "int progressive" [0]`
Setting this to 1 enables progressive rendering with the raytrace hider. The effect will only be visible with some display drivers (such as idisplay).

`Hider "raytrace" "int samplelock" [1]`
This option affects how the random number generator is initialized for some sampling shadeops (e.g. `trace()`). When '`samplelock`' is set to 1, the seed of the random number generator will be the same for any render, which means that any noise perceived in an animation will tend to look *static*. When set to 0, the seed of the random number generation is dependent on `RiFrameBegin` so the sampling noise will change from frame to frame.

More detail about the ray trace hider can be found in Section 7.3.7 [The Ray Tracing Hider], page 130. The '`photon`' hider accepts the '`emit`' parameters as described in Section 7.4.2.3 [Two Pass Photon Mapping], page 132.

Option	Default Value	Comments
`RiFormat`	640 480 1	–
`RiFrameAspectRatio`	1.0	Square pixels
`RiScreenWindow`	-1.3333 1.3333 -1 1	Default computed using `RiFormat` (640/480 = 1.333...).
`RiCropWindow`	0 1 0 1	No cropping.
`RiProjection`	"orthographic"	–
`RiClipping`	0 1e30	–
`RiPixelSamples`	2 2	If motion blur or depth of field are used, increase these values to at least 4 4.
`RiPixelFilter`	"box" 2 2	A high quality setting would be "sinc" 6 6.
`RiExposure`	1.0 1.0	Gain = 1, Gamma = 1
`RiDisplay`	"default.tif" "tiff" "rgba"	Create a RGB TIFF named `default.tif`.

Table 5.2: Standard RenderMan options and their default values.

5.1.2 Implementation Specific Options

3Delight uses implementation specific options only when no standard option provides the desired functionnality. Even if we call them "implementation specific," most of the options described in the following section are commonly used and well defined in other RenderMan implementations. All the supported implementation specific options are listed in Table 5.3 along with their default values.

Rendering Options

`Option "shadow" "float bias0" [0.225]`
`Option "shadow" "float bias1" [0.3]`
`Option "shadow" "float bias" [0.225]`

When using *shadow maps*, a surface may exhibit self-shadowing. This problem is caused by precision errors and quantization and appears as gray spots on the surface. This problem occurs when the distance of an object to a light source computed while rendering a shadow map is slightly less than the distance computed while rendering the image. To prevent self shadowing, specify a bias, which is a value that is added to the shadow map value. If '`bias0`' is different from '`bias1`', the renderer chooses a random value between them for each sample. Note that those parameters do not affect ray traced shadows. The value of '`bias0`' and '`bias1`' should be set to '`0`' if the `"midpoint"` algorithm is used to compute the shadow map, see Section 5.1 [options], page 29 for more details on how to use the `"midpoint"` algorithm.

`Option "render" "integer nthreads" n`

Specifies how many threads to use for the render. Specifying '`0`' tells *3Delight* to use as many threads as there are processors. If the value is negative, it is added to the number of processors to compute the number of threads to use. Refer to Section 7.1 [Multithreading and Multiprocessing], page 120 for a more thorough discussion about multi-threading.

`Option "render" "string bucketorder" "horizontal"`

In order to save memory, the image is rendered progressively in small groups of pixels refered to as "buckets." This option specifies the rendering order of the buckets. Valid values are:

'`horizontal`'
: Rendered left to right, top to bottom (this is the default).

'`zigzag`'
: Rendered left to right on even rows, right to left on odd rows. This ordering is slightly more memory efficient than '`horizontal`'.

'`vertical`'
: Rendered from top to bottom, left to right.

'`spiral`'
: Rendered in a clockwise spiral starting at the center of the image.

'`circle`'
: Rendered in concentric circles starting at the center of the image.

'`random`'
: Rendered in no particular order. Should not be used since it is not memory efficient.

`Option "render" "integer standardatmosphere" [1]`

Traditionnally, volume shaders are applied on top of surface shaders on a per surface basis. While this approach is fine for basic effects like the depthcue shader, it is very limiting for more complex volumetric effects which make use of ray marching. It suffers from a well known issue with transparent surfaces: parts of the volume are integrated several times leading to undesirable artifacts. It also becomes very slow as the geometric complexity of the scene increases.

This option, when set to '0', allows the use of an alternate algorithm better suited for volumetric effects. It handles transparent surfaces properly and provides better performance characteristics with complex scenes. However, it requires that volume shaders follow a few rules for proper composition.

1. The volume shader must be built so that it can be evaluated independently of the surface and later composited. Volume shaders are normally given the color and opacity of the surface as input but with this option they will receive only black. Thus, they must do their work without surface values.

```
volume lazy_fog()
{
    color Cv = 0; /* volume's color */
    color Ov = 0; /* volume's opacity */

    /* Main code here to compute Cv and Ov. */

    /* Composite the volume with the surface. */
    Ci = Cv + (1 - Ov) * Ci;
    Oi = Ov + (1 - Ov) * Oi;
}
```

   That last composition step allows the same shader to work in both modes.

2. The volume shader must compute the atmospheric effect from P to P-I, **not** from P to E as is often done. Also, the shader must be built such that splitting the evaluation range in two at any point and compositing the result of the evaluation of both subranges yields the same values. This will enable partial evaluation of the volume when it is split by transparent surfaces. This condition can be ignored if no transparent surfaces are ever used. This requirement means that volume marching shaders are usually best written as a compositing loop:

```
volume volumetric_smoke()
{
    color Cv = 0; /* volume's color */
    color Ov = 0; /* volume's opacity */

    /* Main volume marching/integration loop. */
    while( ... )
    {
        color Ce = 0; /* volume element color */
        color Oe = 0; /* volume element opacity */

        /* Compute Ce and Oe here for one volume element. */

        /* Composite the volume element in front of the previous ones. */
        Cv = Ce + (1 - Oe) * Cv;
        Ov = Oe + (1 - Oe) * Ov;
    }

    /* Composite the volume with the surface. */
    Ci = Cv + (1 - Ov) * Ci;
    Oi = Ov + (1 - Ov) * Oi;
}
```

`Option "render" "float volumeshadingrate" [1.0]`

This options controls the frequency of shading for interior shaders. It also applies to atmosphere shaders if the `"standardatmosphere"` option is set to 0. Its meaning is similar to the `ShadingRate` attribute: it represents the area in pixels of a shading element for volumes.

`RiOption( "render", "stopcallback", &function_ptr, RI_NULL )`
: This option, only available through a direct `RiOption` call, accepts a pointer to a function which will be called at each rendered bucket or more often to allow user code to stop the render. The function prototype and option call are as follows:

```
int StopRender( void *data )
{
    ...
}

RtPointer callback = (RtPointer) &StopRender;
RiOption( "render", "stopcallback", &callback, RI_NULL );
```

Returning a value of 0 will let the render continue while a value of 1 will stop it. The function may be called from a different thread than the one which started the render. Some care should be taken to make the function execute quickly (eg. avoid calling into a UI toolkit). It should ideally simply test a flag which the application will set when it wants the render to abort.

"stopper_function" is an alias to "stopcallback" for backward compatibility. It is also possible to use this callback to pause a render by delaying the return from the function.

`RiOption( "render", "stopcallbackdata", &data, RI_NULL );`
: This option allows specification of the data parameter passed to the callback set with the "stopcallback" option above.

`Option "shutter" "efficiency" [1.0 1.0]`
: These two values control the shape of the shutter opening and closing for motion blur. Smaller values provide a softer look.

`Option "shutter" "offset" [0.0]`
: The specified value will be added to all subsequent `RiShutter` and `RiMotionBegin` calls. The main utility of this option is to write "canonical" RIB archives that always have the same motion range but are loaded from master RIBs that specify the current shutter time using this option. That way, an RIB archive is independent from shutter time and thus re-usable in each frame: no need to regenerate an archive because of shutter time change, only specify the current "offset" in the master RIB before loading the archive.

`Option "trace" "specularthreshold" [-1]`
: This angle, specified in degrees, is used to determine if ray sampling patterns are of specular or diffuse nature. Patterns with an angle smaller than the value of this option are considered specular. This is used by the `occlusion()` and `indirectdiffuse()` shadeops as well as the `gather` statement. A value of -1 (default) disables this attribute completely.

  IMPORTANT: *Don't use this attribute* since it can affect the visibility of objects since varying the angle in the tracing functions can change ray's type and hence the selection of visibility attributes. More about this attribute in [visibility], page 45. The safe way to override the default ray type for each ray-tracing function is to use the '`raytype`' option as shown in Table 6.10. The only reason this attribute is provided is to keep compatibility with another RenderMan-compliant renderer.

`Option "photon" "emit" [n]`
: Enables automatic photon map generation and specifies the total number of photons to cast in the scene. In this mode, the photon maps are not saved to disk by default. More about this option in Section 7.4.2.2 [Single Pass Photon Mapping], page 132.

`Option "photon" "writemaps" [n]`
: If automatic photon map is generated, photon maps can still be written to disk if this option is set to '`1`'. Default is '`0`'.

`Option "photon" "maxdiffusedepth" [10]`
`Option "photon" "maxspeculardepth" [10]`
: Specifies the maximum diffuse and specular bounces for photons *when using the automatic photon map approach*. In the two-pass approach (see Section 7.4.2.3 [Two Pass Photon Mapping], page 132) the renderer will honour the maximums specified by "maxdiffusedepth" and "maxspeculardepth" trace attributes.

Quality vs. Performance Options

`Option "limits" "integer bucketsize[2]" [16 16]`
: This option specifies the dimension of a bucket, in pixels. Using smaller buckets may reduce memory use at the expense of slight performance loss. In our experience, this default value is the best compromise for *3Delight*.

`Option "limits" "integer gridsize" [256]`
: During the rendering process, geometry is split into small grids of micropolygons. Shading is performed on one grid at a time. This option specifies the maximum number of such micropolygons per grid. The default value ensures that one grid covers approximately one bucket. Since by default `RiShadingRate` is one and buckets are 16 by 16 pixels wide, the default grid size is 16*16/1 = 256.

`Option "limits" "integer eyesplits" [6]`[1]
: Specifies the number of times a primitive crossing the eye plane is split before being discarded.

`Option "limits" "integer texturememory" [131072]`
: The memory needed to hold the texture for a scene may exceed the amount of core physical memory available. To render such scenes efficiently, *3Delight* uses a memory caching system to keep the texture memory footprint below a predefined threshold. This option specifies the amount of memory, in kilobytes, dedicated to texture mapping algorithms. Increasing this parameter may improve texture, shadow and environment map access performance. Note that *3Delight* also offers a texture file caching mechanism over networks, see Section 7.14 [Network Cache], page 154.

`Option "limits" "integer texturesample" [1]`
: Specifies the default oversampling value for `texture()` and `environment()` lookups. Note that `texture()` honors this value only if the '`box`' filter is selected (see Table 6.16) since the '`gaussian`' filter is an analytical anisotropic filter which doesn't use super sampling. Also, specifying a '`samples`' parameter to either `texture()` or `environment()` overrides this value.

`Option "limits" "color othreshold" [0.99608 0.99608 0.99608]`
: Sets the limit opacity value above which surfaces or layers of surfaces are considered fully opaque. This cutoff value enables the renderer to cull geometry more aggressively. *This threshold can provide significant time and memory savings* especially for scenes with a large number of layered and semi-translucent surfaces. An example is shown in Figure 5.2. Setting this threshold to [1 1 1] disables opacity culling.

[1] See *Advanced RenderMan: Creating CGI for Motion Pictures*, p.151, for a complete discussion about eye splits.

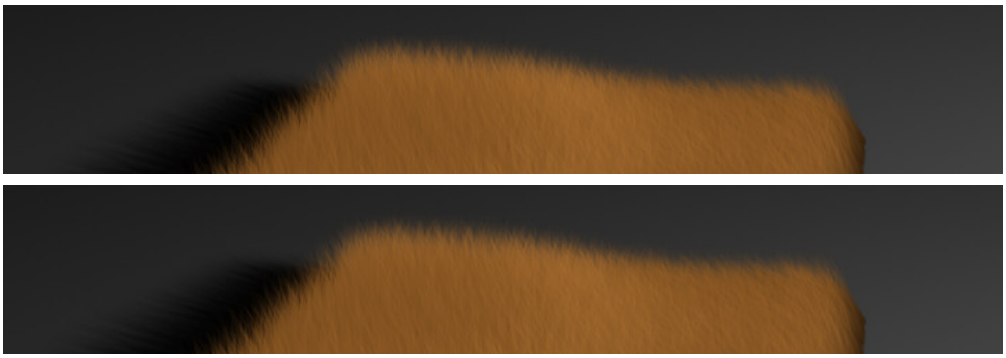


Figure 5.2: A patch of 50000 curves rendered with (top) and without (bottom) opacity culling. The difference is not perceivable (less than 0.4%). The full top image took 119 seconds and 63 MB of RAM to render. The bottom image took 151 seconds and 80 MB of RAM.

`Option "limits" "color zthreshold" [0.99608 0.99608 0.99608]`
: Sets the limit opacity value above which surfaces are considered for depth calculations. For any given surface, the threshold is checked against the accumulated opacity from the camera up to that surface. The default value will cause surfaces to appear in shadow maps only when surfaces behind it are nearly hidden from the camera (ie. the surface itself or combined with surfaces in front of it is nearly opaque). Setting a lower value will make partially transparent surfaces also visible in shadow maps. If an object is marked as a matte, it will only cause depth to be set to infinity if it meets this threshold.

`Option "limits" "integer volumegroupsize" [256]`
: Controls how many points are shaded together for interior shaders and atmosphere shaders if the `"standardatmosphere"` option is set to 0. There should not be any need to change this except for debugging shaders.

`Option "shadow" "integer sample" [16]`
: Specifies the default oversampling value for shadow map and deep shadow map lookups. This only applies if the `shadow` call does not contain an oversampling value (see Section 6.4.6 [texture mapping shadeops], page 107). This option does not affect ray traced shadows.

`Option "trace" "integer maxdepth" [2]`
: This option sets the maximum recursion level for the ray tracer. Valid values are between 0 and 32, inclusively. Any value outside of this range is interpreted as 32 and a value of 0 turns off ray tracing. The default value is 2.

`Option "trace" "float approximation"`
: This option sets a relative detail threshold below which ray intersections may be approximated. The number specifies the ratio between the area of the intersected primitive and the arrea of the ray at the intersection point[2]. If the ratio is smaller or equal to the specified value, the intersection is performed on *approximate geometry*. The default value of 4 is a safe compromise between quality and speed (meaning that an object which is only 4 times larger than the *footprint* of the intersection point will be replaced

[2] Rays in *3Delight* carry *ray differentials* so that it becomes possible to compute the area of the hit point.

by a more efficient *proxy*). A value of 0 can be used to turn this feature off completely. This optimization is especially beneficial for higher order surfaces such as subdivisions and NURBS.

`Option "trace" "float depthaffectsptc" [1]`
: Specifies if ray depths should be checked when using point-based occlusion and color bleeding. For example, setting `Option "trace" "maxdepth" [0]` will, somewhat counter-intuitively, also disable *point-based* occlusion. To enable occlusion/indirectdiffuse even in this case one should set this option to 0.

Search Paths Options

A search path is a colon or semi-colon separated list of directories. The directories should be separated using a slash ('`/`') character. On WINDOWS platforms, it is possible to use `cygwin`'s convention: `//c/dir1/...`.

Some symbols have a special meaning in a search path string:

- The '`@`' symbol specifies the default path. Default paths for each resource are settable through the configuration file (see Section 5.5 [configuration file], page 59) or through environment variables (see Section 2.4 [Environment Variables], page 5). Environment variables *overrides* any value set in the configuration file.
- The '`&`' symbol specifies the previous path list.
- A word starting with a '`$`' and followed by uppercase letters (e.g. `$HOME`) is considered as an environment variable and is replaced by its value. If the specified variable does not exist, it is left untouched in the string.
- The '`~`' symbol, when placed at the beginning of a path, is equivalent to `$HOME`.

Search paths are specified using the following commands:

`Option "searchpath" "string` **shader**`" "`*path-list*`"`
: Sets '`shader`' and DSO shadeops search path.

`Option "searchpath" "string` **texture**`" "`*path-list*`"`
: Sets '`texture`' search path.

`Option "searchpath" "string` **display**`" "`*path-list*`"`
: Sets '`display`' search path.

`Option "searchpath" "string` **archive**`" "`*path-list*`"`
: Sets '`archive`' search path.

`Option "searchpath" "string` **procedural**`" "`*path-list*`"`
: Sets '`procedural`' search path.

`Option "searchpath" "string` **resource**`" "`*path-list*`"`
: Sets '`resource`' search path. This is equivalent to adding this path to all the above search paths.

`Option "searchpath" "string` **dirmap**`" "[\"`*zone*`\" \"`*map-from*`\" \"`*map-to*`\"] [\"`*zone*`\" ... ] ... "`
: Sets a mapping table for all paths. Only entries where *zone* matches the current dirmap zone are used. Then, whenever a path begins with one of the *map-from* entries, this part is replaced by the corresponding *map-to*. The current dirmap zone defaults to UNC on Windows and NFS on all other platforms. It can be overridden in `rendermn.ini` (see Section 5.5 [configuration file], page 59). Note the quotes which are escaped here for the RIB syntax. In the `rendermn.ini` file, this is not necessary.

`Option "searchpath" "string rslpluginmap" "[\"map-from\" \"map-to\"] [\"map-from\" \"map-to\"] ... "`
Sets a mapping table for rsl plugin names. This can be used to make a shader use a different plugin than the one it was compiled with, as long as it contains the required functions.

Default values are shown in Table 5.3.

An environment variable can be set to track down searchpath related problems (see Section 2.4 [Environment Variables], page 5).

Statistics Options

Statistics can prove useful when fine-tuning the renderer's behavior (for quality or performance reasons).

`Option "statistics" "integer endofframe" [0]`
Specifies the desired statistics level. The level ranges from 0 to 3, with 0 meaning no statistics at all and 3 meaning all possible statistics. A level of '2' is enough for most uses.

`Option "statistics" "string filename" [""]`
By default, statistics are dumped to the console (more specifically, to `stdout`); this option can be used to specify an output file name instead. If the file name has the `.html` or `.htm` extension, statistics are dumped in HTML format. **Note:** If specified, the file name should be on the same line as the "endofframe" statistic option in order to get statistics at all.

`Option "statistics" "int progress" [0]`
Turning this option on forces *3Delight* to output a progress status[3] to the console. Output text looks like this:

```
0.33 %
```

Each line is printed after a bucket is done rendering.

`RiOption( "statistics", "progresscallback", &function_ptr, RI_NULL )`
This option, only available through a direct `RiOption` call, accepts a pointer to a function which will be called at each rendered bucket to indicate renderer's progress. The function prototype and option call are as follows:

```
void RenderProgress( float i_progress, void *data )
{
    ...
}

RtPointer callback = (RtPointer) &RenderProgress;
RiOption( "statistics", "progresscallback", &callback, RI_NULL );
```

The *i_progress* parameter will hold a real value in the range [0..100]. The function may be called from a different thread than the one which started the render.

`RiOption( "statistics", "progresscallbackdata", &data, RI_NULL );`
This option specifies the data parameter passed to the progress callback function set with the "progresscallback" above.

[3] Approximate, since it is computed as follow : $\frac{numberofbucketsrendered}{totalnumberofbuckets}$.

Messages Options

The following options can be used to control the output of various messages by *3Delight*.

`Option "messages" "string filter" ""`
: This option allows some error messages to be filtered out. It should be set to a comma separated list of unwanted error codes, such as "A2037,P1168". The values are not cumulative, meaning that setting the option to "" will cause all messages to appear again.

`Option "messages" "float mindispbound" [50]`
: When a displacement bound is specified for an object and that object is displaced by far less than the specified value, *3Delight* emits an informational message. This option controls the percentage below which the message is shown. The default value causes it to be shown when an object uses less than half its displacement bound.

Network Cache Options

3Delight has a network caching system for texture files located on "slow access" media (such as NFS and CD-ROM drives). Refer to Section 7.14 [Network Cache], page 154 for a detailed description.

`Option "netcache" "string cachedir" [""]`
: Specifies a directory for data caching. The directory should be locally mounted, such as `/tmp/3delight_cache` and should be dedicated solely to *3Delight*. Do not use `/tmp` as a cache directory (`/tmp/3delight_cache/` is a better choice).

`Option "netcache" "integer cachesize" [1000]`
: Specifies cache size in megabytes with one megabyte being 1024 Kbytes. To fully take advantage of this performance feature, specify a size big enough to hold many textures, ideally all the textures used in a given scene.

`Option "netcache" "integer minfreespace" [0]`
: Specifies a minimum amount of free space in megabytes to leave on the drive where the netcache is located. This can effectively reduce the set cache size.

`Option "netcache" "integer writecache" [0]`
: This can be set to 1 to enable write caching of the various output files produced by *3Delight*.

`Option "netcache" "string includepaths"`
: Specifies a colon or semicolon separated list of paths which are always cached even if they are not considered slow access.

`Option "netcache" "string excludepaths"`
: Specifies a colon or semicolon separated list of paths which are never cached even if they are considered slow access.

User Options

`Option "user" "type variable" value`
: Sets a user defined option. Such an option can then be read from inside a shader using the `option()` shadeop. See [option shadeop], page 116. A simple example:

```
Option "user" "float shadow_pass" 1  # rendering a shadow
```

: It is also possible to define user attributes, as explained in Section 5.2.6 [user attributes], page 51.

RIB Output Options

These options are only meaningful when using the C API. More details and code snippets can be found in Section 7.15 [Outputting RIB], page 156.

`RiOption( "rib", "archiveprocedurals", namepattern, RI_NULL )`
: When this is set to a non-empty string and RIB output is used, all `RiProcedural` calls are modified to use `RiProcDelayedReadArchive`. The required archive is generated and named according to *namepattern* which should contain the `"#a"` string to be replaced by a unique number for each archive.

`RiOption( "rib", "callprocedurals", onoff, RI_NULL )`
: Enables or disables procedurals expansion during RIB output. *onoff* is a string and can be set to either '`on`' or '`off`'.

`RiOption( "rib", "compression", compression, RI_NULL )`
: Specifies a compression for RIB output. The only supported *compression* is '`gzip`'. This option must be called before `RiBegin`.

`RiOption( "rib", "format", format, RI_NULL )`
: Specifies the format of the output stream. *format* can either be '`ascii`' or '`binary`'.

`RiOption( "rib", "header", onoff, RI_NULL )`
: Specifies if a header is written at the beginning of RIB output. This option must be called before `RiBegin`. *onoff* is a string and can be set to either '`on`' or '`off`'.

`RiOption( "rib", "ident", onoff, RI_NULL )`
: Specifies if ascii RIB output is to be formatted nicely. *onoff* is a string and can be set to either '`on`' or '`off`'.

`RiOption( "rib", "pipe", fd, RI_NULL )`
: Specifies a file descriptor to be used when outputing RIB. This option must be called before `RiBegin`.

Other Options

`Option "licensing" "integer waitforlicense" 1`
: When this option is set to 1, *3Delight* will wait for a license when none is available to render. If the option is set to 0, *3Delight* will return immediately if no license is available and renderdl will return an error code of 99 (see [renderdl return values], page 10). This is useful when managing a renderfarm and other work could be scheduled instead of the waiting *3Delight* process.

`Option "licensing" "integer holdlicense" 0`
: By default, *3Delight* will get new licenses for every render and release them once the render is done. This can be undesirable if several frames are rendered from the same RIB or application session. If this option is set to 1, the licenses obtained for the first render are held until RiEnd() is called. For renderdl, this means until the last RIB is done processing.

`Option "licensing" "string server" ""`
: Allows specification of the license server to use as an option. If this is left empty, the /3delight/licserver key from `rendermn.ini` is used instead. Note that this option must be set before the first world block to have any effect.

`Option "limits" "float processmemory" 0`
: The render will abort with error R5060 if the process memory use in bytes exceeds the specified value. A value of zero disables the feature.

Option	Default Value	Comments
"shadow" "bias0"	0.225	–
"shadow" "bias1"	0.300	–
"render" "bucketorder"	["horizontal"]	–
"render" "standardatmosphere"	1	old behavior by default
"render" "volumeshadingrate"	1.0	shades for every pixel
"shutter" "efficiency"	[1.0 1.0]	The shutter opens and closes sharply.
"shutter" "offset"	0.0	No offset.
"limits" "bucketsize"	[16 16]	–
"limits" "gridsize"	256	One grid is approximately as large as a bucket when `RiShadingRate`=1.
"limits" "eyesplits"	6	Enough for most cases, larger values may slow down rendering.
"limits" "othreshold"	[0.99608 .. ..]	–
"limits" "texturememory"	131072	128 megabytes
"limits" "texturesample"	1	–
"limits" "zthreshold"	[1.0 1.0 1.0]	Consider only opaque objects for z.
"limits" "volumegroupsize"	256	–
"shadow" "sample"	16	–
"trace" "maxdepth"	1	–
"trace" "specularthreshold"	10	–
"searchpath" "shader"	`$DELIGHT/shaders`	–
"searchpath" "texture"	`$DELIGHT/textures`	–
"searchpath" "display"	`$DELIGHT/displays`	–
"searchpath" "archive"	`$DELIGHT/archive`	–
"searchpath" "procedural"	""	–
"searchpath" "resource"	""	–
"searchpath" "dirmap"	""	–
"searchpath" "rslpluginmap"	""	–
"netcache" "cachedir"	[""]	No network caching.
"netcache" "cachesize"	1000	1 gigabyte of disk space. Has no effect if caching is disabled.
"netcache" "minfreespace"	0	–
"netcache" "writecache"	0	Output files are not cached.
"netcache" "includepaths"	""	–
"netcache" "excludepaths"	""	–
"statistics" "endofframe"	0	No stats.
"statistics" "file"	""	Outputs statistics to console (‘`stdout`’).
"statistics" "mindispbound"	50.0	–

Table 5.3: Implementation specific options and their default values.

5.2 Attributes

Attributes are components of the graphic state that are associated with elements of the scene, such as geometric primitives and light sources. As with options, there are two kinds of attributes: the standard and the implementation specific. Standard attributes, along with their default values, are listed in Table 5.4. Implementation specific attributes are passed through the `RiAttribute` command and are described in the subsections sections below. All attributes are saved at `RiAttributeBegin` and restored at `RiAttributeEnd`. Attributes are also implicitely saved at each `RiFrameBegin` and `RiWorldBegin` and restored at the enclosing `RiWorldEnd` and `RiFrameEnd`.

Attribute Name	**Default**	**Comments**
Color	[1 1 1]	Default is white.
Opacity	[1 1 1]	Default is opaque.
TextureCoordinates	[0 0 1 0 0 1 1 1]	This option only applies to parametric surfaces such as `RiPatch` and `RiNuPatch`.
Surface	`defaultsurface`	This default surface shader does not need a light-source to produce visible results.
Atmosphere	–	–
Interior	–	–
Displacement	–	–
LightSource	–	–
ShadingRate	1	Default is approximately one shading computation at every pixel.
GeometricApproximation "motionfactor"	1	ShadingRate is scaled up by motionfactor/16 times the number of pixels of motion. This also affects depth of field.
GeometricApproximation "focusfactor"	1	ShadingRate is scaled proportionally to the radius in pixels of the circle of confusion for depth of field, multiplied by this constant.
Matte	0	–
Sides	2	Objects are visible from both sides by default.
Orientation	"outside"	Do not invert transform handedness by default.
TrimCurve	–	No trim curves defined by default.

Table 5.4: Standard RenderMan attributes and their default values

5.2.1 Primitives Identification

`Attribute "identifier" "name" [""]`
: Subsequent objects are named as specified by this attribute. This is useful when *3Delight* reports warnings or errors about a primitive.

`Attribute "grouping" "string membership" [""]`
: Specifies group membership of subsequent primitives. Beginning the group name with "+" or "-" adds or removes subsequent primitives from the specified group name. Specifying "" as the group name resets any group memberships. More about grouping in Section 7.3.4 [Trace Group Membership], page 128 and Section 7.9 [Display Subsets], page 148.

`Attribute "grouping" "string tracesubset" [""]`
Sets the default group subset for primitives that are *shooting* a ray. This attribute sets the default value if ray tracing functions do not provide any value. More about grouping in Section 7.3.4 [Trace Group Membership], page 128.

5.2.2 Primitives Visibility and Ray Tracing

`Attribute "visibility" "integer camera" [1]`
Specifies if an object is visible to primary camera rays.

`Attribute "visibility" "integer trace" [0]`
Specifies if an object is visible to secondary rays traced in the scene. It affects rays traced by `environment()`, `trace()`, `gather()` and `indirectdiffuse()`. This attribute is obsolete; `"visibility" "diffuse"` and `"visibility" "specular"` should be used instead.

`Attribute "visibility" "integer diffuse" [0]`
`Attribute "visibility" "integer specular" [0]`
Specifies if an object is visible to diffuse or specular rays. Diffuse rays are traced by `gather()`, `indirectdiffuse()` and `occlusion()`. Specular rays are traced by `gather()`, `trace()` and `environment()`. Note that these attributes must be specified with inline declarations and take precedence over `"visibility" "trace"`. The way objects are shaded when hit by a specular or diffuse ray is specified using "hitmode" attributes described in [hitmode], page 46.

> IMPORTANT: `gather()`, `indirectdiffuse()` and `occlusion()` functions can trace *both* specular and diffuse rays depending on the *specular threshold* as explained in [specularthreshold], page 36.

`Attribute "visibility" "string transmission" ["transparent"]`
Specifies how an object appears to transmission-like rays. In particular, controls how a specific surface casts shadows on other surfaces. Possible values for this attribute are:

`"transparent"`
: The object does not cast shadows on any other object since it is completely transparent.

`"opaque"`
: The object casts a shadow as a completely opaque object.

`"Os"`
: The object casts a shadow according to the opacity value given by `RiOpacity` or by the *Os* variable attached to the geometry, if provided.

`"shader"`
: The object casts shadows according to the opacity value computed by the surface shader.

This attribute affects rays traced by `shadow()`, `transmission()` and `occlusion()`. This attribute is obsolete; `"visibility" "integer transmission"` should be used instead.

`Attribute "visibility" "integer transmission" [0]`
Specifies if an object is visible to transmission rays. See also [hitmode], page 46. Transmission rays are those traced by `shadow()` and `transmission()`. Note that this attribute must be specified with an inline declaration and takes precedence over `"visibility" "string transmission"`.

`Attribute "visibility" "integer photon" [0]`
Specifies if an object is visible to photons.

`Attribute "visibility" "string subsurface" [""]`
: Specifies that geometry is visible to "subsurface" rays. This means that affected primitives are included in subsurface scattering computations and that they belong to the specified subsurface group name. Primitives that are in the same subsurface group are considered as forming one closed object. A special group name of `"*"` must be specified for procedural geometry if its contents is visible to several different subsurface groups. More on subsurface scattering in Section 7.4.5 [Subsurface Scattering], page 136.

`Attribute "shade" "string diffusehitmode" ["primitive"]`
`Attribute "shade" "string specularhitmode" ["shader"]`
`Attribute "shade" "string transmissionhitmode" ["shader"]`
: These attributes determine if the object's shader is run when it is hit by various types of rays. A value of '`shader`' runs the shader to compute Ci and Oi while '`primitive`' uses the object's Cs and Os attributes directly. The later is of course much faster.

`Attribute "shade" "string photonhitmode" ["shader"]`
: Specifies the action to take when a photon hits a surface. The default action is to run the shader and use the resulting color and opacity in combination with the photon shading model as specifyed in [photon shading model], page 47. Using '`primitive`' instead of '`shader`' will use object's Cs and Os quantities without any shading.

`Attribute "shade" "string camerahitmode" ["shader"]`
: If this is set to '`primitive`' instead of '`shader`' the object is assumed to be opaque by the camera. This allows the renderer to compute visibility faster without running the surface shader.

`Attribute "shade" "viewdependent" [0]`
: When using multi-camera rendering (see Section 7.8 [Multi-Camera Rendering], page 148) this will make the renderer shade each grid one per camera.

`Attribute "cull" "integer hidden" [1]`
: Specifies if primitives hidden by other primitives are culled or not. Specifying '`0`' to this attribute is only meaningful when "baking" a shader and could have a **significant** impact on performance if not used properly.

`Attribute "cull" "integer backfacing" [1]`
: Specifies if primitives backfacing the camera are shaded or not if `RiSides` is set to '`1`'. Only meaningful when "baking" a shader. By default, cull primitives that are backfacing the camera.

`Attribute "hider" "integer composite" [1]`
: Enables or disables compositing of transparent surfaces together. Any object which has this attribute set to 0 will cause compositing to act is if all the objects behind it did not exist. This allows the object to be semi-transparent without other objects showing through, giving direct control over the alpha channel of the final image.

`Attribute "trace" "float bias" [.01]`
: This bias affects all traced rays. The bias specifies an offset added to ray's starting point (in ray's direction) so as to avoid intersecting with the emitting surface.

`Attribute "trace" "int samplemotion" [0]`
: By default rays are all launched at the object's shading time. However, rays traced from an object which has this attribute set, are spread over the object's visibility time span so they can see motion blur.

`Attribute "trace" "maxdiffusedepth" [1]`
`Attribute "trace" "maxspeculardepth" [2]`
: Specifies the maximum depth for reflections and diffuse bounces.

`Attribute "light" "string shadows" ["off"]`
Turns automatic shadow calculation ["on"] or ["off"] for `RiLightSource`s specified after this directive.

`Attribute "light" "integer samples" [1]`
Controls the oversampling of automatic shadows. Larger values improve shadow accuracy but slow down rendering.

`Attribute "light" "string samplingstrategy" ["lightsource"]`
Controls how *3Delight* samples area lights. Two values are accepted:

`"lightsource"`
When this sampling strategy is selected (it is the default), *3Delight* will run each area light as many times as there are area light samples and will average the result. Surface shaders will be provided with `L`, `Cl` and `Ol` that are the result of the averaging. This method will generally produce correct *soft shadows* (since in most cases light sources are responsable of shadow computation) but incorrect lighting[1].

`"illuminance"`
This is the "physically correct" area light sampling that will run *illuminance based shadeops*[2] in all surface shaders as many times as there are area light samples so to ensure a proper BRDF sampling. This mode is recommended if:

- Accurate specular highlights are important.
- Shadows are generated from inside surface shaders (and not light shaders).

In terms of performance, the '`lightsource`' mode is obviously slightly faster since it doesn't trigger multiple evaluation of `illumiance` constructs.

5.2.3 Global Illumination Attributes

`Attribute "irradiance" "nsamples" [64]`
Specifies the default number of samples to use when calling `occlusion()` or `indirectdiffuse()` shadeops. This default value can be modified by passing a parameter to those shadeops. See Section 7.3.6 [Oversampling and Derivatives], page 129.

`Attribute "irradiance" "shadingrate" [4]`
3Delight can use a different shading rate for irradiance computations to speed up `occlusion()` and `indirectdiffuse()` computations. Using higher shading rates enables irradiance interpolation across shaded grids, and therefore speeds up rendering.

`Attribute "irradiance" "maxerror" [0.1]`
Controls a maximum tolerable error when using interpolation (for an irradiance shading rate higher than 1). Parameter range is [0..1].

`Attribute "photon" "color reflectance" [1 1 1]`
This is a multiplier over the reflectance given by the surface shader (or the primitive color as specified by the shading model below). This can be used to exaggerate or to attenuate the color bleeding effect produced by a particular object.

[1] Although the difference with the correct "illuminance" sampling is in many case unnoticeable.
[2] This include explicit and implicit illuminance loops that are *hidden* inside shadeops such as `specular()`.

`Attribute "photon" "string shadingmodel" ["matte"]`
Sets the shading model of the surface when interacting with a photon. Accepter shading models are '`matte`', '`chrome`', '`water`', '`glass`' and '`transparent`'.

`Attribute "photon" "causticmap" [""]`
`Attribute "photon" "globalmap" [""]`
Specifies a caustic or a global photon map for the current surface. This value be retrieved from inside a shader using the `attribute()` shadeop. See [attribute shadeop], page 114.

`Attribute "photon" "int estimator" [50]`
Specifies the total number of photons to use when estimating radiance or irradiance. This will be the default value when calling the `photonmap()` shadeop. See [photonmap shadeop], page 105.

5.2.4 Subsurface Light Transport

`Attribute "subsurface" "color absorption" [`*absorption*`]`
`Attribute "subsurface" "color scattering" [`*scattering*`]`
Specifies reduced absorption and reduced scattering coefficients for the subsurface simulation. These two parameters should always be specified in pairs:

```
# marble reduced scattering + absorption parameters
Attribute "subsurface" "color scattering" [2.19 2.62 3.00]
  "absorption" [0.0021 0.0041 0.0071]
```

`Attribute "subsurface" "color meanfreepath"`
`Attribute "subsurface" "color reflectance"`
Another way to specify subsurface parameters, using translucency and reflectance (mean free path and diffuse reflectance). These two parameters should be specified in pair. If "reflectance" is not specified it is set to the `RiColor` of the surface.

```
# marble reduced meanfreepath + diffuse reflectance
Attribute "subsurface"
  "meanfreepath" [8.50 5.566 3.951] "reflectance" [ 0.83 0.79 0.75]
```

`Attribute "subsurface" "refractionindex"`
This attribute specifies the third parameter of the subsurface simulation. It is a single float indicating the relative index of refraction of the simulated material (commonly referred to as n).

`Attribute "subsurface" "scale" [1]`
The values for subsurface scattering use a millimeter scale. Since most scenes are modeled on a different scale, this attribute compensates for that instead of adjusting the parameters themselves. It is a single float which defaults to 1.0.

`Attribute "subsurface" "shadingrate" [1]`
This attribute controls how many irradiance samples should be taken during the precomputation phase. It is a single float which defaults to 1.0 and has a meaning similar to `RiShadingRate`.

`Attribute "subsurface" "referencecamera" ["cameraname"]`
Specifies a dicing camera to use when evaluating the subsurface light transport. The camera must be declared with `RiCamera` first as explained in Section 7.12 [Dicing Cameras], page 153.

More on subsurface scattering in Section 7.4.5 [Subsurface Scattering], page 136.

5.2.5 Displacement

Whenever a displacement shader is applied to a primitive, one must specify a displacement bound to inform the renderer about maximum displacement amplitude. This is done using one of the attributes described below.

`Attribute "displacementbound" "float sphere" [`*radius*`] "string coordinatesystem" "`*ss-name*`"`

This tells the renderer that a displaced point does not move outside a sphere of a given radius in the given coordinate system. The following coordinate systems are supported:

- `"surface"` which is the surface shader's coordinate system.
- `"displacement"` which is the displacement shader's coordinate system.
- `"shader"` which means `"displacement"` if a displacement shader is attached to the object and `"surface"` if not.
- `"current"` which is the shading language "current" space.
- `"null"` which specifies the coordinate system active when the attribute call is made.
- Any other named coordinate system which will be accessible to the object's shaders.

If no coordinate system is specified then the `"object"` space is used. Do not forget to specify a displacement bound when using displacement shaders.

`Attribute "displacementbound" "float sphere" [`*radius*`] "matrix transform" [`*matrix*`]`

This is very similar to the previous attribute except that the coordinate system is specified using a matrix.

`Attribute "bound" "float displacement" [`*radius*`]`

This is provided for backward compatibility and specifies a sphere of a given radius living in the object's coordinate system.

`Attribute "trace" "int displacements" [0]`

If true (`[1]`), objects are displaced prior to ray-primitive intersection tests. See [Raytracing Displacements], page 128.

`Attribute "trace" "float displacementshadingrate" [1]`

Controls at which frequency the displacement shader is run on the geometry before intersecting it with rays. Larger values will be faster and use less memory while smaller values will yield more accurate results.

`Attribute "trace" "float inflategrids" [0]`

This setting allows the grids used to trace displacements to be slightly inflated to hide cracks which may occur between them. Values between 0 and 1 will usually hide most problems. Turning this on can have a significant effect on performance so use it only when needed.

`Attribute "displacement" "int concatenate" [0]`

When this is set to 1 and a displacement shader is instantiated, its effect will be concatenated to that of existing displacement shader(s) in the current attribute state. Any number of shaders can be linked together in this fashion. When the object is displaced, the innermost shaders (ie. the ones declared last) are run first.

`Attribute "displacement" "string detail" ["default"]`
: This attribute controls how displacement shaders are run on geometric primitives. The default value causes them to be run at the finest possible detail level for smooth, accurate displacement. It is also possible to specify "vertex" which will cause the displacement shaders to be run on the primitive's vertices for less accurate but faster displacement. Note that in that case, the `calculatenormal` shadeop will not work so the normals need to be displaced explicitely.

3Delight also allows the displacement bound to be specified as parameters of the displacement shader. This is especially useful when using stacked displacement shaders. The "sphere" and "coordinatesystem" parameters of the "displacementbound" attribute are simply replaced by two parameters.

EXAMPLE

```
displacement bumpy(
    float Km = 1, amplitude = 1;
    string texturename = "";
    float __displacementbound_sphere = Km * amplitude;
    string __displacementbound_coordinatesystem = "current" )
{
    if( texturename != "" )
    {
        float amp = Km * amplitude * float texture( texturename, s, t );

        P += amp * normalize(N);
        N = calculatenormal( P );
    }
}
```

Note that the parameters can be either computed by the shader (as in the example) or set when the shader is instantiated. The displacement bound specified with this method and the one specified with the attribute are added together.

As an optimization, the shader may also have an extra parameter which tells the render if it is actually doing any displacement, based on its specific parameter values. This allows *3Delight* to render more efficiently when a displacement shader is used but set to do no real displacement. Here is an example with a simple shader object:

```
class bumpy_surface(
    float Km = 1, amplitude = 1;
    string texturename = "";
    float Ka = 0.2, Kd = 0.8 )
{
    public constant float __displacementbound_sphere;
    public constant string __displacementbound_coordinatesystem = "current";
    public constant float __displacement_enabled;

    public void construct()
    {
        __displacementbound_sphere = Km * amplitude;
        if( __displacementbound_sphere > 0 && texturename != "" )
        {
            __displacement_enabled = 1;
        }
```

```
        else
        {
            __displacement_enabled = 0;
        }
    }

    public void displacement( output point P; output normal N )
    {
        if( texturename != "" )
        {
            float amp = Km * amplitude * float texture( texturename );
            P += amp * normalize(N);
            N = calculatenormal( P );
        }
    }

    public void surface( output color Ci; output color Oi )
    {
        float diffuse = I.N;
        diffuse = (diffuse*diffuse) / (I.I * N.N);
        Ci = Cs * ( Ka + Kd*diffuse );
        Oi = Os;
        Ci *= Oi;
    }
}
```

Note that public members are being used here instead of parameters.

5.2.6 User Attributes

`Attribute "user" "uniform type variable" value`

Specifies a user defined attribute. User attributes can be read in the shading language using the `attribute()` shadeop. See [attribute shadeop], page 114.

EXAMPLE

```
Attribute "user" "float self_illuminated" [1]
```

Note that it is also possible to declare user defined options, as described in [User Options], page 41.

5.2.7 Other Attributes

`Attribute "sides" "float backfacetolerance" [90]`

This value is the angle, in degrees, which specifies how much a surface must be backfacing so it can be discarded before displacement shading occurs. This is to account for the fact that the displacement shader may change the orientation of the surface. A value of 90 causes no backface culling to take place before displacement (the safe choice and the default). A value of 0 causes all backfacing surfaces to be discarded before displacement.

`Attribute "sides" "integer doubleshaded" [0]`

When this is set to 1, each surface is shaded twice: once as usual and once with the normals reversed. This allows thickness to be given to a surface with displacement or to avoid shading artifacts where a transparent surface changes orientation.

`Attribute "trimcurve" "string sense" ["inside"]`
: Specifies whether the '`interior`' or the '`exterior`' of a trim curve should be trimmed (cut) on a NURBS surface. The default value is "inside". Specifying "outside" reverses the behavior.

`Attribute "dice" "rasterorient" [1]`
: When raster oriented dicing is off (`[0]`), surfaces are diced as if they will always face the camera. This is most useful when rendering surfaces that lose displacement details when turned sideways to the camera since micropolygons become big in camera space while they stay small in raster space. This feature **increases** the number of diced and shaded micropolygons, which leads to longer render times.

`Attribute "dice" "hair" [0]`
: This attribute allows direct control over the tesselation in the u parametric direction of curves. When set to 0, the width of the curve determines tesselation like other primitives. When set to a greater value, it is taken as the number of micropolygons to create in the u direction. A value of 1 is typically used to render hair.

`Attribute "dice" "hairumin" [0]`
: This attribute is an alternate form of control over the tesselation in the u parametric direction of curves. It specifies the minimum number of micropolygons to create in the u direction. Unlike "dice" "hair", the actual tesselation can be greater than the specified value. It is useful for shaders which require more than one micropolygon to produce the correct look. Note that if both this attribute and "dice" "hair" are specified, the actual number of micropolygons will be the largest of both.

`Attribute "dice" "lodreferencecamera" ""`
: Specifies a camera (or a list of cameras, separated by commas) to use in selecting the detail level to render. The cameras should be declared using `RiCamera`. If this is not specified, the dice reference cameras are used instead. If those are not specified, the main camera is used.

`Attribute "dice" "referencecamera" ""`
: Selects a camera (or a list of cameras, separated by commas) to use for dicing. The camera should be declared using `RiCamera` as explained in Section 7.12 [Dicing Cameras], page 153.

`Attribute "derivatives" "integer centered" [1]`
: Centered derivatives help remove artifacts due to abutting patches having a different parametric orientation. They are automatically turned on with smooth shading interpolation and should never be turned off.

`Attribute "derivatives" "integer forcesecond" [1]`
: By default, *3Delight* attempts to provide shaders with values which allow second derivatives to be computed, even for very fine geometry. This means, for example, that the derivative of the surface normal will usually not be null. It also improves shading of curved surfaces. When the geometry is really too detailed, this may be unnecessary and have a significant impact on rendering times. Setting this attribute to 0 can then improve performance without a visible degradation of the image.

`Attribute "derivatives" "integer smooth" [1]`
: Smooth derivatives help reduce shading artifacts at grid boundaries by providing a smooth variation of `du` and `dv` across the grid. They do so by computing ideal values as if the `ShadingRate` attribute had been exactly met. This can also improve performance when micropolygons are much smaller than they need to be due to excessive geometric

detail in the scene. However, the very same situation can cause issues with texture filtering if the overpaint zone is not large enough.

`Attribute "procedural" "integer reentrant" 1`
When run with multiple threads, *3Delight* may choose to call the same procedural function with different data from more than one thread at once. If some procedural is not thread safe, this attribute can be set to 0 to disable that behavior[3]. The default should be used for better performance.

`Attribute "shade" "string frequency" "motionsegment"`
This controls how often primitives with multi-segment motion blur are shaded. The default value of "motionsegment" shades at the start of every segment. A value of "frame" specifies shading only once per frame, at the first segment. This is less accurate but will usually be faster and use less memory. Note that some primitives only support "motionsegment", in which case the attribute is silently ignored.

`Attribute "shade" "volumeintersectionstrategy" ["exclusive"]`
Dictates what to do with intersecting interior volumes. By default, only one of the interior volume is run. If set to "additive", all volumes in an intersecting region will be evaluated.

`Attribute "shade" "integer smoothnormals" 0`
Setting this attribute to 1 tells *3Delight* to render polygon meshes with smooth normals instead of geometric normals. It will have no effect if the N primitive variable was already supplied with the mesh.

`Attribute "shade" "shadingrate" [1]`
This is exactly equivalent to the `ShadingRate` command.

`Attribute "shade" "relativeshadingrate" [1]`
`Attribute "shade" "resetrelativeshadingrate" [1]`
"relativeshadingrate" sets the value of the relative shading rate. This will be multiplied by all the relative shading rates found "upward" in the attribute stack. "resetrelativeshadingrate" will reset the current value of the relative shading rate to the provide value (meaning that relative shading rates won't be considered from that point upwards). Final shading rate of the surface is computed by multiplying the shading rate by the cumulated relative shading rates. The following example illustrates how this feature works.

```
ShadingRate 2.0
AttributeBegin
  Attribute "shade" "relativeshadingrate" 10.0
  # shading rate is now 20
  AttributeBegin
    Attribute "shader" "relativeshadingrate" 0.2
    # shading rate is is now 4
    AttributeBegin
      Attribute "shader" "resetrelativeshadingrate" 0.1
      # shading rate is now 0.2 since 0.1 x 2 = 0.2
    AttributeEnd
  AttributeEnd
AttributeEnd
```

[3] Calls to the procedural will then be *serialized*.

5.3 Geometric Primitives

3Delight supports the entire set of geometric primitives defined by the RenderMan standard. Since the primitives are explained in great detail in the RenderMan specification, the following chapters give only a brief description and some useful implementation details.

5.3.1 Subdivision Surfaces

Catmull-Clark subdivision surfaces are supported by *3Delight*[1]. All standard tags are recognized:

`"hole"` A per-face value, tagging faces that are considered as holes. The parameter is an array of integers listing the indices of the hole faces.

`"corner"` A per-vertex value, tagging vertices that are considered semi-sharp. The tag should be specified with an array of integers, giving the indices of the vertices as well as an array of floating point values giving their sharpness. Corners are smooth at sharpness 0.0 and grow sharper as the value is increased.

`"crease"` A per-edge value, tagging edges that are considered as semi-sharp. This tag is specified with an array of integers, indicating the list of vertices that form an edge chain. An array of floating point values specifies the sharpness value for each edge in the chain.

`"interpolateboundary"`
A per-surface value, specifying that boundary edges and vertices are infinitely sharp. This tag takes an optional integer parameter with the following possible values:

- 0 - No interpolation, the default behavior.
- 1 - Full interpolation: infinitely sharp creases are added to boundary edges and infinitely sharp corners are added to boundary vertices of valence 2. This is the behavior obtained when the tag is specified without a parameter.
- 2 - Interpolation of edges only: infinitely sharp creases are added to boundary edges.

`"facevaryinginterpolateboundary"`
A per-surface value which allows using either true facevarying (set to 0) or a facevertex-like version (set to 1) which smooths out the boundaries. The default value is 1 and can be changed in `rendermn.ini` with the `"/3delight/subdivisionmesh/facevaryinginterpolateboundary"` key.

The following are 3Delight specific extensions:

`"smoothcreasecorners"`
This tag requires a single integer parameter with a value of 0 or 1 indicating whether or not the surface uses enhanced subdivision rules on vertices where more than two creased edges meet. With a value of 0, such a configuration uses the conventional rules which treat the vertex as a sharp corner. With a value of 1, the vertex is subdivided using an extended crease vertex subdivision rule which yields a smooth crease. Note that sharp corners can still be explicitly requested using the "corner" tag.

Unlike other rendering packages, *3Delight* does not attempt to tesselate the entire subdivision surface into many small polygons (thus taking a large amount of memory); instead, a lazy and adaptive process is used to generate only those portions of the surface that are needed for some specific bucket.

All standard variable types are supported: `constant`, `uniform`, `varying`, `vertex`, `facevarying` and `facevertex`. `facevertex` is used exactly as `facevarying` but interpolates the variables according to surface's subdivision rules. This eliminates many distortion artifacts due to bilinear interpolation in `facevarying`s.

[1] Other subdivision schemes (such as Loop and Butterfly) are rendered as polygonal meshes.

5.3.2 Parametric Patches

All parametric patches are supported:

`RiPatch` Specifies a single bicubic or bilinear patch. All standard basis matrices are supported: `"bezier"`, `"bspline"`, `"catmull-rom"` and `"hermite"`.

`RiPatchMesh`
Specifies a rectangular mesh of bilinear or bicubic patches. As with `RiPatch`, all basis matrices are supported. It is recommanded to use `RiPatchMesh` instead of `RiPatch` when specifying connected surfaces since this is more efficient and more suited to *3Delight*'s crack elimination algorithm.

`RiNuPatch` Specfies a NURBS surface. Trimming is supported.

The following variable types are allowed for parametric patches: `constant`, `uniform`, `varying`, `vertex` and `facevarying`.

5.3.3 Curves

Both '`linear`' and '`cubic`' curves are supported. Attaching a normal to the curve geometry modifies the curve's orientation depending on the normal's direction; this can be used to render features such as grass. The following variables are supported: `constant`, `uniform`, `varying`, `vertex` and `facevarying`.

Note: *uniform*s are specified per curve and not per curve segment as was the case in old versions.

5.3.4 Polygons

All polygonal primitives are supported, this includes: `RiPolygon`, `RiGeneralPolygon`, `RiPointsPolygons` and `RiPointsGeneralPolygons`. All polygonal primitives support the following variable types: `constant`, `uniform`, `varying`, `vertex` and `facevarying`.

Note: *vertex* and *varying* interpolation over a polygon is ill-defined since it depends on how the polygon is tesselated by the renderer.

5.3.5 Points

Points are supported in *3Delight* through the standard RenderMan `RiPoints` primitive. Points' size is controlled using either `"width"` or `"constantwidth"` variable (`RI_WIDTH` or `RI_CONSTANTWIDTH` in the API). The way *3Delight* renders points is selectable during primitive declaration using the `"uniform string type"` variable which can take one of the following values: '`particle`', '`blobby`', '`patch`', '`sphere`' or '`disk`'.

`"particle"`
The default mode is to render points using a lightweight primitive, suitable for clouds of tiny particles (such as clouds of dust, stars, etc ...). Particles are rendered as circles using a specialized algorithm which is suitable for such small primitives.

Note: Displacement and volume shaders, as well as output variables, are not supported for lightweight particles.

`"blobby"` Renders the particles as "blobbies" (implicit surfaces). In this case, the energy field radius is defined by *width* or *constantwidth*. Particles created as blobbies will blend together smoothly and will also blend all primitive variables.

`"sphere"`
`"disk"`
`"patch"` Points are rendered using the `RiSphere`, `RiDisk` or `RiPatch` primitive, respectively. This is useful when shading over the surface of the particle is important. It is also

more efficient to use this feature to declare large groups of spheres, disks and patches instead of declaring each primitive individually. It is not recommended to use these high level primitives for small, particle-like, points.

Note: Since spheres are two sided primitives, they will render differently from disks, patches and particles when the material is not fully opaque and `RiSides` is set to '2'.

When using the '`patch`' particle type, two additional variables are understood by *3Delight*:

`"constant|varying float patchaspectratio"`
: Controls the "aspect ratio" of the patch. This is defined as *width*/*height* and will only modify the patch's *height* (the width being specified by *width* or *constantwidth*). When `constant`, the aspect ratio applies to all particles; `varying` aspect ratio gives one value per particle.

`"constant|varying float patchrotation"`
: Gives the rotation, in degrees, of the patch around its center.

Additionally, *width*, *constantwidth*, *patchaspectratio* and *patchrotation* are correctly motion blurred.

EXAMPLE

```
WorldBegin

  Translate 0 0 10
  Color 1 1 1

  AttributeBegin
    Rotate -30 0 1 0
    Rotate 60 1 0 0
    Scale 0.3 0.3 0.3
    # Render some points using the 'sphere' primitive ...
    Points
      "P" [ -3 2 1 3 2 1 -3 0 1 -1 0 1 1 0 1 3 0 1
            -1 -4 1 1 -4 1 -3 2 -1
             3 2 -1 -3 0 -1 -1 0 -1 1 0 -1 3 0 -1 -1 -4 -1 1 -4 -1 ]
      "constantwidth" [0.2]
      "uniform string type" "sphere"
  AttributeEnd
WorldEnd
```

Listing 5.1: Using sphere-shaped `RiPoints`.

5.3.6 Implicit Surfaces (Blobbies)

`RiBlobby` is implemented. It supports spheres and segments[2]. All standard operators are supported, including: add, subtract, min, max, multiply, divide and negate. Variables specified to `RiBlobby` are interpolated over the resulting isoparametric surface by blending. Note that `vertex` and `varying` variables are specified *per node*; `uniform` variables are specified for the entire primitive.

3Delight uses an adaptive algorithm to render implicit surfaces, meaning that only small parts of the primitive are tessellated at a given time. Tessellation resolution is controlled by `RiShadingRate` and should be decreased if very small implicit surfaces (in raster space) are rendered. In addition,

[2] Repulsion planes are not supported yet.

setting `RiShadingInterpolation` to '`smooth`' (the default) greatly improves the rendering quality of implicit surfaces.

5.3.7 Quadrics

All six quadrics (plus the torus) are supported. This includes: `RiSphere`, `RiDisk`, `RiHyperboloid`, `RiParaboloid`, `RiCone`, `RiCylinder` and `RiTorus`. Each of these primitives accept the following variable types: `constant`, `uniform`, `varying` and `vertex`. Note that `vertex` variables behave *exactly* as `varying`s on quadrics.

5.3.8 Procedural Primitives

Procedural primitives from the C binding using `RiProcedural` are supported. Built-in procedures for the RIB binding (`RiProcDelayedReadArchive`, `RiProcRunProgram` and `RiProcDynamicLoad`) are also fully supported.

The following two RIB bindings were added to provide functionality equivalent to calling `RiProcRunProgram` or `RiProcDynamicLoad` directly from a C program:

```
ProcDynamicLoad [ "dsoname" "dsodata" ] detail
ProcRunProgram [ "progname" "progdata" ] detail
```

They invoke the corresponding procedural immediately when encountered in a RIB. Unlike their counterparts used through the `Procedural` RIB binding, they can be used to alter the current attribute and transform stacks. Note that the corresponding binding for `RiProcDelayedReadArchive` is simply `ReadArchive`.

3Delight also supports a `RiProceduralV` C API which allows the following parameters to be specified:

`"string instancekey"`
: Setting this makes *3Delight* handle procedurals for which it has the same value as instances of the same geometry. This means the procedural may be expanded only once and its output reused internally. This can improve both performance and memory use.

Limitations

Procedural primitives cannot be used inside a `RiMotionBegin`/`RiMotionEnd` block. They can still be subject to transformation motion blur as well as contain motion blocks themselves. When a procedural primitive contains a motion block, it is the user's responsability to take it into account when providing the renderer with a bounding box for the procedural primitive.

Procedural primitives cannot be used inside a `RiObjectBegin`/`RiObjectEnd` block. They can however contain such a block and use objects declared before themselves (with the caveats mentioned below).

The following parts of the graphics state are not saved and restored for procedural primitives:

1. Named coordinate systems created with `RiCoordinateSystem`
2. Retained geometry objects (`RiObjectBegin`/`RiObjectEnd`)

This means that declaring one of the above, inside a procedural primitive, may affect the graphics state for later procedural primitives. Likewise, a declaration made after the procedural primitive declaration (but before it was needed by the renderer) may affect it. It is therefore necessary to declare the required elements before the procedural primitives that require them and not to replace them until after the end of world block, when rendering is complete. It also wise to avoid using these declarations inside the procedural primitive itself. This behavior is subject to change in future releases depending on user feedback about it. Contact us at `info@3delight.com` if these features are important to you.

5.3.9 Object Instances

Object instancing is supported with the `RiObjectBegin/RiObjectEnd` and `RiObjectInstance` APIs. *3Delight* also supports a `RiObjectBeginV` API which allows the following parameters to be specified:

`"string __handleid"`
: This specifies the handle *3Delight* should use for the object. RtObjectHandle and RtString are actually equivalent in *3Delight*.

`"string scope"`
: This specifies at which scope the object should be inserted. The default is `"local"` which is wherever `RiObjectBegin` is called. Other possible values are `"world"` which makes the object available for the entire render and `"global"` which makes the object available until `RiEnd`. These two require that the handle be specified with `"__handleid"`. Their purpose is to allow a procedural primitive to generate objects lazily and reuse them in other instances of the procedural. Note that this will require some synchronization in the procedural's code.

5.3.10 Constructive Solid Geometry

CSG is fully supported. This includes all defined operations on solids: '`union`', '`difference`' and '`intersection`'. One limitation of the CSG algorithm is that primitives spanning the eye plane might give incorrect results when used in a CSG tree.

As an extension, *3Delight* allows putting procedural primitives inside any kind of solid block. This behaves as if the content of the procedural was immediately expanded except if the procedural is the first element in a '`difference`' block. In that case, the procedural will act as an implicit '`union`' block.

5.4 External Resources

5.4.1 Texture Maps

3Delight supports the following optional parameters to the `RiMakeTexture` API:

`"string compression"`
: Specifies the compression to apply to the texture. Possible values are "lzw", "on" (lzw), "1" (lzw), "zip", "deflate" and "off" (uncompressed). The default is "lzw".

`"int bakeres"`
: Specifies the resolution when converting a bake file.

`"int progress"`
: If set to 1, a progress indicator will be printed to the console.

`"int separateplanes"`
: If set to 1, enables the -separateplanes flag of tdlmake (see [tdlmake options], page 15).

`"int flips"`
: If set to 1, enables the -flips flag of tdlmake (see [tdlmake options], page 15).

`"int flipt"`
: If set to 1, enables the -flipt flag of tdlmake (see [tdlmake options], page 15).

`"int newer"`
: If set to 1, enables the -newer flag of tdlmake (see [tdlmake options], page 15).

`"float gamma"`
: Sets the -gamma flag of tdlmake (see [tdlmake options], page 15).

`"float rgbagamma[4]"`
: Sets the -rgbagamma flag of tdlmake (see [tdlmake options], page 15).

`"string colorspace"`
: Sets the -colorspace flag of tdlmake (see [tdlmake options], page 15).

5.4.2 Archives

5.4.2.1 RiArchiveRecord

In addition to what is required by the specification, *3Delight* will attempt to parse the "verbatim" type requests as RIB when used in immediate rendering mode. This is for consistency with the results obtained when RIB commands are injected with this function in RIB output mode. This will only work correctly if complete commands are output in a single `RiArchiveRecord` call.

5.4.2.2 Inline Archives

The following two commands can be used to define inline archives:

```
RiArchiveBegin( RtToken archivename, ...parameterlist... )
RiArchiveEnd()
```

Any commands issued between that pair will be recorded as part of the archive *archivename*. They can then be replayed using `RiReadArchive` or the `RiProcDelayedReadArchive` procedural.

5.5 Configuration file (`rendermn.ini`)

Some of the default options and attributes are settable through the `rendermn.ini` initilization file. The renderer looks for it in four different locations, in order:

1. In *3Delight*'s installation directory, as specified by the `$DELIGHT` environment variable.
2. In the directories specified by the `$DELIGHT_CONF` environment variable (a colon separated list).
3. In user's home directory, as specified by the `$HOME` environment variable. It looks also for an `.rendermn.ini` file in the same directory.
4. Finally, *3Delight* tries to load `rendermn.ini` in the current directory.

If *3Delight* encounters the file in more than one location, it uses the latest keys and values it finds. This enables the user to have a site specific initialization file in the `$DELIGHT` directory and user specific settings in their home directory.

Note that this file has nothing to do with the `.renderdl` file that is read by the `renderdl` executable (see Section 3.1 [Using the RIB renderer], page 7). The main difference (other than the file format) is that `rendermn.ini` is read by *3Delight*'s core library which means that applications linked with this library (including `renderdl`) are affected by the content of `rendermn.ini`. On the other hand, `.renderdl` is only read by `renderdl`.

Each line in the configuration file is of the form "**key** value1 [value2 ... value_n]"; '`key`' being the option or attribute, in lowercase. As illustrated in Listing 5.2, the naming conventions for '`key`' are closely related to its RIB binding.

Additionally, each key can be prefixed by '`/3delight`' which will make it meaningful to *3Delight* only. This can prove useful if more than one RenderMan compliant renderer is used in a facility. Note that such keys will override the keys that do not have the '`/3delight`' prefix (as shown in Listing 5.2). Recognized options and attributes are:

```
/exposure %f %f
/geometricapproximation/motionfactor %f
/geometricapproximation/focusfactor %f
/shadingrate %f
/shadinginterpolation %s
```

All attributes and options which can be set with `RiAttribute` (see Section 5.2 [attributes], page 44) and `RiOption` (see Section 5.1 [options], page 29) can also be set this way. A few examples:

```
/attribute/dice/hair 1
/attribute/limits/eyesplits 5
/option/limits/othreshold 0.96 0.96 0.96
/option/netcache/cachedir /tmp/3delight_cache/
/option/netcache/cachesize 2000
```

Default resolution and aspect ratio can be specified for any display driver using the following three keys:

```
/display/displaydrivername/xres %d
/display/displaydrivername/yres %d
/display/displaydrivername/par %f
```

It is possible to specify a translation for a display driver name using the following key:

```
/displaytype/displaydrivername %s
```

It is also possible to directly specify the library to use for a given display:

```
/display/dso/displaydrivername %s
```

Note that the translation is still applied before looking up this key.

Default search paths for different resources are also settable in the `rendermn.ini` file. Those could then be extracted using the '`@`' symbol when specifying search paths using `Option "searchpath"`. Environment variables are accepted and specified using the `$VAR` convention.

```
/defaultsearchpath/texture %s
/defaultsearchpath/shader %s
/defaultsearchpath/display %s
/defaultsearchpath/archive %s
/defaultsearchpath/procedural %s
/defaultsearchpath/resource %s
```

The current directory mapping zone is set using:

```
/dirmap/zone %s
```

The timeout when waiting for rendering hosts to connect to the master in multi-process rendering (see Section 7.1.3 [Network Rendering], page 121) is set in seconds with:

```
/3delight/networkrender/jobtimeout 600
```

```
# Yes, '#' at the _beginning_ of a line indicates a comment.
# Specify a gamma correction.
/exposure                                1.7 1

# 3delight specific options
/option/limits/texturememory             100000
/option/limits/bucketsize                16 16
/option/limits/othreshold                0.99608 0.99608 0.99608

# curves are diced as hair
/attribute/dice/hair                     1

# default shading rate becomes 0.5
/shadingrate                             1.0

# override the shading rate for 3Delight
/3delight/shadingrate                    0.5

# set default shader path
/3delight/defaultsearchpath/shader       $HOME/shaders:$DELIGHT/shaders


# the file display drivers defaults to tiff
/displaytype/file tiff

# set default resolution for the tiff display driver
/display/tiff/xres                      800
/display/tiff/yres                      600

# specify the library to use for a custom display driver
/display/dso/stereo                     /lib/foo/stereo_dd.so
```

Listing 5.2: An example `rendermn.ini` file.

Finally, setting the `$DL_DUMP_DEFAULTS` environment variable forces *3Delight* to print all read keys and their values from the configuration file(s).

6 The Shading Language

6.1 Syntax

6.1.1 General Shader Structure

6.1.1.1 Traditional Structure

Traditional shaders are pre-RSL 2.0 shaders, they exhibit a very simple structure consisting of one main entry point and a certain number of supporting functions (a structure clearly inspired by the C programming language).

```
{.. Function declarations .. }

[surface|displacement|volume|imager|light] shadername( Parameters )
{
    .... shader body ...
}
```

The output of such shaders depends solely on their type and is passed back to the renderer through a standard output variable (see Section 6.2 [Predefined Shader Variables], page 84). Here is a summery for each shader type:

surface Surface shaders define the color and the opacity (or transparency) of the material by setting the *Ci* and *Oi* output variables.

displacement Displacement shaders can perturb the geometry of the material by modifying the *P* and *N* variables.

volume Volume shaders are executed in-between surfaces or between the eye and some surface (for atmosphere shaders) to simulate various atmospheric effects. They modify *Ci* and *Oi* as in surface shaders.

imager Imager shaders run on the image plane, on every pixel. They modify *Ci* and *alpha* variables.

light Light shaders are slightly more complicated than other shaders in that they also declare an `illuminate()` construct[1] (see [The Illuminate Construct], page 76). They operate by setting *Cl* and *L* output variables.

6.1.1.2 Shader Objects

The modern structure, introduced in RSL 2.0 specifications, closely resembles C++ or Java classes. The general structures is as follows:

```
class shadername( ... shader parameters ... )
{
    ... member variables declarations ...

    ... member methods declarations ...
};
```

In a nutshell this new structure (an example is shown in Listing 6.1) provides interesting capabilities and some performance benefits:

[1] Unless they are ambient lights, in which case they don't need such a construct.

1. Shaders can call methods and access *member variables* of other shaders (only public variables and methods are accessible, as expected). This feature allows, finally, to build comprehensive shader libraries that are pre-compiled and re-usable. Intra-shader data and method access is performed using the `shader` variable type (see Section 6.1.2.6 [Co-shaders], page 68).
2. Member variables can retain shader state. This avoids ad-hoc message passing and other "tricks" widespread in traditional shaders.
3. Access to co-shaders encourages a modular, layer-based, development of looks. For example, illumination models can be passed as co-shaders to surface shaders which can take care of the general structure.
4. The separation of displacement, surface and opacity methods (see below) provides the render with greater opportunities to optimize renderings.
5. Last but not least, shader objects provide a greater flexibility overall and *promising future development perspectives*.

Variables and method declarations have the usual form of traditional shaders but can be prefixed by the `public` keyword. So how does such a class shader define its function (surface, displacement, ... etc) ? By implementing predefined methods as shown in Table 6.1.

`public void surface ( output color Ci, Oi; ... )`	Declaring this method makes the class usable as a valid surface shader (can be called by `RiSurace()`).
`public void displacement ( output point P; output normal N; ... )`	Declaring this method makes the class usable as a valid displacement shader (can be called by `RiDisplacement()`).
`public void volume ( output color Ci, Oi; ... )`	Declaring this method makes the class usable as a valid volume shader (can be called by `RiInterior()` and `RiAtmopshere`).
`public void light(output vector L; output color Cl; ...)`	Declaring this method makes the class usable as a valid light shader (can be called by `RiLightSource()` and `RiAreaLight`).
`public void opacity ( output color Oi; ... )`	If this function is declared, *3Delight* will call it when evaluating transmission rays (see [transmission shadeop], page 98). This can lead to a good performance benefit since computing opacity is usually much less expensive than evaluating the entire surface shader.

Table 6.1: Predefined class shader methods.

This shader structure allows the inclusion of *both* a surface and a displacement entry points. Listing 6.1 demonstrates how this is done and shows the clear benefits of retaining the state of variables in shader objects. In the context of traditional shaders, sharing data between different methods had semantics or was resolved using ad-hoc methods:

1. Share the result of the computation using message passing (see Section 6.4.8 [Message Passing and Information], page 113). This works but complicates the data flow.
2. Re-do the same computation in both the surface and displacement shaders. Also works but time is lost in re-computation.
3. Displacement is performed inside the surface shader. This is the most practical solution but sadly it doesn't give the renderer a chance to do some important render-time optimizations.

```
class plastic_stucco(
    float Ks = .5;
    float Kd = .5;
    float Ka = 1;
    float roughness = .1;
    color specularcolor = 1;
    float Km = 0.05;
    float power = 5;
    float frequency = 10;
    color dip_color = color(.7,.0, .2) )
{
    varying float magnitude = 0;

    public void displacement(output point P; output normal N)
    {
        /* put normalized displacement magnitude in a global variable
           so that the surface method below can use it. */
        point PP = transform ("shader", P);
        magnitude = pow (noise (PP*frequency), power);

        P += Km * magnitude * normalize (N);
        N = calculatenormal (P);
    }

    public void surface(output color Ci, Oi)
    {
        normal Nf =  faceforward( normalize(N), I );
        vector V  = - normalize( I );

        color specular_component =
            specularcolor * Ks * specular(Nf, V, roughness);
        color diffuse_component = Kd * diffuse(Nf);

        /* attenuate the specular component in displacement "dips" */
        specular_component *= (1-magnitude) * (1-magnitude);

        Oi = Os;
        Ci = ( Cs * (Ka * ambient() + diffuse_component) +
            specular_component );

        Ci = mix(dip_color*diffuse_component, Ci, 1-magnitude );

        Ci *= Oi;
    }
}
```

Listing 6.1: En example class shader computing both surface shading and displacement.

6.1.2 Variable Types

6.1.2.1 Scalars

All scalars in the RenderMan shading language, including integers, are declared using the *float* keyword. As an example, the following line declares two scalars and sets the value of one of them to 1.

```
float amplitude = 1.0, frequency;
```

Some notes about scalars:

- Since there is no boolean type in RSL (such as the `bool` keyword in C++), scalars are also used to hold truth values.
- Scalars can be *promoted* to almost any type. For example, a color can be initialized with a scalar:

```
Ci = 1;
```

6.1.2.2 Colors

The shading language defines colors as an abstract data type: it could be an RGB 3-tuple or an elaborate spectrum definition. But until now, all known implementations only allow a 3-tuple definition in various color spaces. A color definition has the following form (square brackets mean that enclosed expression is optional):

```
color  color_name = [color "color_space"](x, y, z);
```

As an example, the following line declares a color in HSV space:

```
color foo = color "hsv" (.5, .5, .5);
```

If the space is not provided it is assumed to be RGB:

```
color foo = 1; /* set the color to (1,1,1) in RGB space. */
```

'`rgb`'	Red, Gree and Blue.
'`hsv`'	Hue, Saturation and Value.
'`hsl`'	Hue, Saturation and Lightness.
'`xyz`'	CIE XYZ coordinates.
'`yiq`'	NTSC coordinates.

Table 6.2: Supported color spaces.

Operations on colors are: addition, multiplication and subtraction. All operations are performed channel per channel. Transformation of points between the various color spaces is performed using the `ctransform()` shadeop (see [ctransform], page 92)

6.1.2.3 Points, Vectors and Normals

Point-like variables are *3-tuples* used to store positions, directions and normals in the euclidean *3-space*. Such a 3-tuple definition is of the form:

```
{point|normal|vector} = [ {point|normal|vector} "coordsys"] (x, y, z);
```

Where '`coordsys`' can be one of standard coordinate systems (see Table 6.3) or any coordinate system defined by `RiCoordinateSystem`. As an example, all the following definitions are valid:

```
point p1 = (1, 1, 1); /* declare a point in current coordinate system */
vector v1 = vector "world" (0,0,0); /* a vector in world coordinates */
normal NN = normal 0; /* normal in current space. Note type promotion */
```

`current`	The default coordinate system in which the renderer performs its work. In *3Delight*, the '`current`' space is the same as the '`camera`' space
`object`	The coordinate system in which the primitive is declared
`shader`	The coordinate system in which the shader is declared
`world`	The coordinate system active at `RiWorldBegin`
`camera`	The coordinate system of the camera with positive z pointing forward, positive x pointing to the right and positive y pointing up
[cameraname:]`screen`	A perspective corrected coordinate system of the camera's image plane. The range is defined by `RiScreenWindow`. The optional *cameraname* can be used to specify any camera declared with `RiCamera` and will transform a point from *current* space to target camera screen space.
[cameraname:]`raster`	The 2D projected space of the image. The upper-left corner of the image is (0, 0) with x and y increasing towards their maximum values as specified by `RiFormat`. The optional *cameraname* can be used to specify any camera declared with `RiCamera` and will transform a point from *current* space to target camera raster space
[cameraname:]`NDC`	The Normalized Device Coordinates. Like '`raster`' space but x and y goes from 0 to 1 across the whole image. In other words: $NDC(x) = raster(x)/Xres$ and $NDC(y) = raster(y)/Yres$. The optional *cameraname* can be used to specify any camera declared with `RiCamera` and will transform a point from *current* space to target camera NDC space.

Table 6.3: Predefined Coordinate Systems

Transforming point-like variables between two coordinate systems is performed using the `transform()` shadeop (see [transform shadeop], page 92). It is important to note that that shadeop is *polymorphic*: it acts differently for points, vectors and normals. The following code snippet transforms a vector from '`world`' to '`current`' space:

```
vector dir = transform( "world", "current", vector(0,1,0) );
```

The following exampl transforsm a point from current space to *matte_paint*'s camera NDC space (the camera has to be declared in the RIB using `RiCamera`)

```
point mattepaint_NDC = transform( "matte_paint:NDC", P );
```

NOTE: *3Delight* keeps all points in the '`current`' coordinate system[2], which happens to be '`camera`'. This could sometimes lead to confusion:

```
point p1 = point "world" (0,1,0);
printf( "%p\n", p1 );
```

One could expect to see $(0, 1, 0)$ output by `printf()` but this is generally not the case: since *p1* is stored in the '`current`' coordinate system, the printed value will be the point resulting from the transformation of *p1* from '`world`' to '`current`'. So if the camera space is offset by $(0, -1, 0)$ the output of `printf()` would be $(0, 0, 0)$.

A different implementation would have been possible (where points are kept in their respective coordinate systems as late as possible) but that would necessitate run-time tracking coordinate systems along with point-like variables, which is not handy and offer no real advantage.

Operations on Points

`point + vector`
: The results is a point.

`point - point`
: The result is a vector.

`point * point`
: The result is point.

`point * vector;`
: The result is point.

Operations on Vectors and Normals

Any vector type in the following table can be interchanged with a normal.

`vector . vector`
: The result is a scalar (dot product);

`vector + vector`
`vector - vector;`
: The result is a vector;

On Correctness

The shader compiler will not, by default, enforce mathematical correctness of certain operations. For example, adding two points together makes no real sense geometrically (one should add a vector to a point) but this is still accepted by the shader compiler (although with a warning). Historically, in the very first definitions of the RenderMan Shading Language, there were no vectors nor normals so everything had to be performed on points.

6.1.2.4 Matrices

A matrix in the shading language is a 4x4 homogeneous transformation stored in row-major order[3]. A matrix initialization can have two forms:

```
matrix m = [matrix "space"] scalar;
matrix m2 = [matrix "space"] ( m00, m01, m02, m03, m10, m11, m12, m13,
                               m20, m21, m22, m23, m30, m31, m32, m33 );
```

[2] Similarly to all RenderMan-compliant renderers.

[3] Matrices declared using `Ri` calls or in a RIB file are column major.

The first form sets the *diagonal* of the matrix to the specified scalar with all the other positions being set to zero. So to declare an identity matrix one would write:

```
matrix identity = 1;
```

A zero matrix is declared similarly:

```
matrix zero = 0;
```

Exactly as for point-like variables (see Section 6.1.2.3 [Points Vectors and Normals], page 65, matrices can be declared in any standard or user-defined coordinate system (see Table 6.3):

```
matrix obj_reference_frame = matrix "object" 1;
```

Point-like variables can be transformed with matrices using the `transform()` shadoep (see [transform shadeop], page 92).

6.1.2.5 Strings

Strings are most commonly used in shaders to identify different resources: textures, coordinate systems, other shaders, ...etc. A string declaration is of the form:

```
string name = "value";
```

One can also declare strings like this:

```
string name = "s1" "s2" ... "sN";
```

All operations on strings are explained in Section 6.4.7 [string shadeops], page 112.

6.1.2.6 Co-Shaders

Co-shaders are both a variable type and an abstract concept defining a *modular component*. Co-shaders are declared in a shader using the `shader` keyword and can be loaded using special co-shader access functions as detailed in Section 6.4.9 [Co-Shader Access], page 119. For example,

```
shader specular_component = getshader( "specular_instance" );
if( specular_component != null )
    Ci += specular_component->Specular( roughness ) * Ks;
```

Declares a *specular_component* shader variable and uses it to load and execute the '`specular_instance`' co-shader (which could be a Blinn or a Phong specular model depending on the co-shader). The co-shader has to be declared in the scene description using the `RiShader()` call. Member variables in a co-shader can be accessed using `getvar()` method:

```
float Ks;
/* "has_Ks" will be set to 0 if no Ks variable is declared in the co-
   shader. Also note that the last output parameter "Ks" can be omit-
   ted in which case getvar serves to check the existence of the
   target member variable.  */
float has_Ks = getvar( specular_component, "Ks", Ks );
```

The arrow operator can be used instead of `getvar()` to access a member variable in a more compact way:

```
Ks = specular_component->Ks;
```

The difference with `getvar()` is that *3Delight* will print a warning if the variable is not declared in the target co-shader. Additionally, the arrow operator *will only access variables in a read-only mode and cannot be used to write into variables*, this means that to modify a member variable in a target co-shader one has to use a "setter" method.

6.1.2.7 The Void Type

This type is only used as a return type for a shading language function as explained in Section 6.1.7.4 [Functions], page 82.

6.1.3 Structures

6.1.3.1 Definition

Structures, or *structs* in short, are defined in a manner very similar to the C language. The main difference is that each structure member must have a default initializer.

```
struct LightDescription
{
   point from = (0, 0, 0);
   vector dir = (0, 0, 1);
   varying float intensity = 1;
}
```

Some observations:

- Note that there is no semicolon at the closing bracket as it is usual in C.
- A structure has no class modifier so each member has to be declared with the appropriate `uniform` or `varying` keyword. If there is no variable class specifier, `uniform` is assumed.
- Embedded structures do no need initializers, since each structure will have its own member initialization.
- Structures can be declared anywhere in the file. More generally, a structure can be declared anywhere a function can be delcared and same scoping rules apply.

6.1.3.2 Declaration and Member Selection

Defining a structure variable is done as with standard language variables and structure members are accessed using the arrow (`->`) operator as with C pointers. Using the `LightDescription` structure declared in Section 6.1.3.1 [Structure Definition], page 69, one can write:

```
LightDescription light_description;
light_description->intensity = 10;
```

Note that the arrow operator to select structure members is a no-cost operation, compared to the same operator used to access co-shader variables (see Section 6.1.2.6 [Co-shaders], page 68. It is possible, at declaration time, to override the members of a structure using a *constructor*:

```
LightDescription light_description = LightDescription(
    "from", point(1,0,0), "dir", vector(1,1,1) );
```

Note that the constructor runs at *compile time* so the member names must be constant strings. Declaring arrays of structures is also trivial:

```
LightDescription light_descriptions[10] =
     { LightDescription("intensity", 5) };
```

6.1.3.3 Structures as Function Parameters

Structures are passed to function per reference (no data is copied when calling the function). To modify any member of the structure inside the function it is necessary to use the `output` keyword. For example:

```
void TransformLight( output LightDescription ld; matrix trs )
{
    ld->from = transform( trs, ld->from );
    ld->dir = transform( trs, ld->dir );
}
```

6.1.3.4 Limitations

Currently, the following features are not supported:

- Structure inheritance and structure member functions.
- Passing structures to RSL plug-ins.
- Functions cannot return structures.

6.1.4 Arrays

6.1.4.1 Static Arrays

Any shading language type can be used to declare a one-dimensional array (excluding the `void` type). The general syntax for a static array is:

```
type array_name[array_size] [= {x, y, z, ...}];
type array_name[array_size] [= scalar];
```

array_size must be a constant expression and the total number of initializers must be equal or less than *array_size*[4]. When initializing an array with a scalar all array's elements are set to that particular value. For example:

```
float t[4] = { 0, 1, 2, 3 };
color  n[10] = 1; /* set first element to (1,1,1), the rest to 0 */
```

Note that the length of the array in the initializer can be omitted, in which case the size is deduced from the right-hand expression:

```
float t[] = { 1,2,3 }; /* create a static array of size 3. */
```

The length of arrays can be fetched using the `arraylength()` shadeop (see [arraylength shadeop], page 117:

```
float array_length = arraylength( t );
```

A particular element in an array can be accessed using squared brackets:

```
float t[4] = 1;
float elem_2 = t[2];
```

6.1.4.2 Resizable Arrays

Resizable arrays are declared using a non-constant length or by not providing the array length at all:

```
color components[]; /* empty */
shader lights[] = getlights();
color light_intensities[ arraylength(lights) ]; /* one per light */
```

Note that arrays specified without a length as shader parameters are *not* resizable arrays: their length is fixed when the parameter is initialized. Thus, only locally declared arrays can be resizable. Additionally, initializing an array will define a static array:

```
float t[] = { 1,2,3 }; /* WARNING: create a *static* array of size 3. */
```

Resizing Arrays

Resizable arrays can be resized using the `resize()` shadeop as in:

```
float A[];
resize(A, 3);
```

Note that the three elements of A in the example above are not initialized. Arrays can also be resized by assignment:

```
float A[];
float B[];
resize(A, 3);
```

[4] If less initializer are specified, the remaining elements are set to zero.

```
B = A; /* B is now of length 3 */
```

Arrays can be resized by adding or removing elements from their "tail":

```
float A[];
push( A, 1 );
push( A, 2 ); /* length = 2 */
pop( A ); /* length = 1 */
```

`resize()`, `push` and `pop()` are further described in Section 6.4.10 [Operations on Arrays], page 119.

Array's Capacity

Additionally to its length, a dynamic array has a *capacity*. Declaring a capacity for a dynamic array is solely a performance operation: it allows a more efficient memory allocation strategy. `capacity()` is described in Section 6.4.10 [Operations on Arrays], page 119.

6.1.5 Variable Classes

Additionally to variable types, the RenderMan shading language defines variables *classes*[5]. Since the beginning, RSL shaders were meant to run on a multitude of shading samples at a time, a technique commonly called *single instruction multiple data* or in short: SIMD. SIMD execution was a natural specification for the shading language mainly because the first production oriented RenderMan-compliant renderer implemented a REYES algorithm - a type of algorithm well suited for such grouped execution. An inherent benefit of an SIMD execution pipeline is that the cost of interpretation is amortized: one instruction interpretation is followed by an execution on many samples, this makes interpretation in RSL almost as efficient as compiled shaders.

6.1.5.1 Uniform Variables

Uniform variables are declared by adding the `uniform` keyword in front of a variable declaration. For example,

```
uniform float i;
```

Defines a uniform scalar *i*. Uniform variables are constant across the entire *evaluation sample set*. In slightly more practical terms, a uniform variables is initialized *once per grid*. All variables that do not vary across the surface are good uniform candidates (a good example is a `for` loop counter). It is important to declare such variables as uniforms since this can accelerate shader execution and lower memory usage[6].

NOTE: The *evaluation sample set* is a grid in the primary REYES render but is not necessary a grid in the ray-tracer.

6.1.5.2 Varying Variables

Varying variables are variables that can change across the surface and is initialized once per micro-polygon. A perfect example is the *u* and *v* predefined shader variables (see Section 6.2 [Predefined Shader Variables], page 84) or the result of the `texture()` call (see [texture shadeop], page 107). To declare a varying variable one has to use the `varying` keyword:

```
varying color t;
```

Declaring varying arrays follows the same logic but one has to make sure a varying is really needed in such a case: a varying array can consume a large amount of memory.

---

[5] Nothing to do with C++ classes.

[6] Although the shader compiler is good at detecting variables that have not been declared properly and turn them into uniforms.

NOTE: Assigning varying variables to uniforms is not possible and the shader compiler will issue a compile-time error. Assigning uniforms to varyings is perfectly legal.

6.1.5.3 Constant Variables

Constant variables were introduced in RSL 2.0 and are meant to declare variables that are initialized *only once* during the render. This type of variables is further explained in Section 6.1.1.2 [Shader Objects], page 62.

6.1.5.4 Default Class Behaviour

If no `uniform` or `varying` keyword is used, a default course of action is dictated by the RenderMan shading language. This default behaviour is context depended:

1. Member variables, declared in shader objects (see Section 6.1.1.2 [Shader Objects], page 62) are `varying` by default.
2. Variables declared in shader and shader class parameter lists are `uniform` by default.
3. Variables declared inside the body of shaders are `varying` by default.
4. Variables declared in function parameter lists inherit the class of the parameter and this behaviour also propagates inside the body of the function. This is possible since RSL functions are inlined, more on this in Section 6.1.7.4 [Functions], page 82.

It is recommended not to abuse the `varying` and `uniform` keywords where the default behaviour is clear, this makes code more readable.

6.1.6 Parameter Lists

Parameters list can be declared in three different contexts: shader parameters, shader class parameters and function parameters. Shader and shader class parameters have exactly the same form and will be described in the same section whereas function parameters has some specifics and will be described in their own section.

6.1.6.1 Shader and Shader Class Parameters

This kind of parameters has three particularities:

1. Each parameter *must* have a default initializer. This is to be expected since all parameters must have a valid state in case they are not specified in the RenderMan scene description (through `RiSurface` or similar).
2. By default, each parameter is uniform (see Section 6.1.5 [Variable Classes], page 71).
3. They can be declared as *output*. Meaning that they can be passed to a display driver (as declared by `RiDisplay`).

Follows an example of such a parameter list:

```
surface dummy(
    float intensity = 1, amplitude = 0.5;
    output color specular = 0; )
{
    shader body
}
```

6.1.6.2 Function Parameters

Function parameters have the same form as shader parameters but have no initializer. Another notable difference is that, by default, parameters class is inherited from the argument. For example:

```
/* note that 'x' has no class specification. */
float sqr( float x; ) { return x*x; }

float uniform_sqr( uniform float x; )  { return x*x; }

surface dummy( float t = 0.5; )
{
    /* function will be inlined in uniform form. */
    Ci = sqr( t );

    /* funciton will be inlined in varying form. */
    Ci += sqr( u );

    /* this will not compile! ('v' is varying but uniform_sqr
       expects a uniform argument) */
    Ci += uniform_sqr( v );
}
```

More about functions in Section 6.1.7.4 [Functions], page 82.

6.1.7 Constructs

6.1.7.1 Conditional Execution

Similarly to many other languages, the conditional block is built using the `if`/`else` keyword pair:

```
if( boolean expression )
    statement
[else
    statement ]
```

The bracketed `else` part is optional.

6.1.7.2 Classical Looping Constructs

There are the two *classical*[7] looping constructs in the shading languag: the `for` loop and the `while` loop. Both work almost exactly as in many other languages (C, Pascal, etc...). The general form of a `for` loop is as follows:

```
if( expression ; boolean expression ; expression )
    statement
```

For example:

```
uniform float i;
for( i=0; i<count; i += 1 )
{
    ... loop body ...
}
```

Note that *i* is uniform. This means that the loop counter is the same for all shading samples during SIMD execution. Declaring loop counters as varying makes little sense, usually. Also note that it is not possible to *declare* a variable in the loop initializing expression.
The `while` loop has the following structure:

```
while( boolean expression )
```

[7] We call them "classical" since they are present, in one form or another, in almost all programming languages.

```
    statement
```

For example,

```
uniform float i=0;
while( i<10 )
{
    i = i + 1;
}
```

The normal execution of the `for`/`while` loop can be altered using two special keywords:

`continue [l]` This skips the remaining code in the loop and evaluates loop's boolean expression again. Exactly as in the C language. The major difference with C is the optional parameter *l* that specifies how many loop levels to exit. The default value of *l* is 1, meaning the currently executing loop.

`break [l]` This keyword exits all loops until level *l*. This is used to immediately stop the execution of the loop(s).

NOTE**:** Having the ability to exit *l* loops seems like a nice feature but usually leads to obfuscated code and flow. It is recommended not to use the optional loop level *l*, if possible.

6.1.7.3 Special Looping Constructs

This section describes constructs that are very particular to the RenderMan shading language. These constructs are meant to describe, in a clear and general way, a fundamental problem in rendering: the interaction between light and surfaces.

The `illuminance` Construct

This construct is an abstraction to describe an integral over incoming light. By nature, this construct is only meaningful in surface or volume shaders since only in these two cases that one is interested to know about incoming light (an example of a different usage is shown in Section 7.6.4 [Baking Using Lights], page 143). There are two ways to use `illuminance`:

1. Non-oriented. Such a statement is meant to integrate light coming from *all directions*. In other words, integrate over the entire sphere centered at *position*.

   ```
   illuminance ( [string category], point position, ... )
       statement
   ```

2. Oriented. Such a statement will only consider light that is coming from inside the open cone formed by *position*, *axis* and *angle*.

   ```
   illuminance ( [string category,]
                 point position, vector axis, float angle, ... )
       statement
   ```

Light Categories

The optional *category* variable specifies a subset of lights to include or exclude. This feature is commonly named *Light categories*. As explained in Section 6.3 [Predefined Shader Parameters], page 89, each light source can have a special `__category` parameter which lights the categories to which it belongs. Categories can be specified using simple expressions: a series of terms separated by the `&` or `|` symbols (logical and, logical or). Valid category terms are described in Table 6.4. If a category is not specified `illuminance` will proceed with all active light sources.

`""`	Matches all lights regardless of their category.
`"name"`	Matches lights that belong to category '`name`'.
`"-name`	matches lights that do not belong to category '`name`', including lights that do not belong to any category.
`"*"`	Matches every light that belongs to at least one category.
`"-"`	Matches nothing.

Table 6.4: Valid category terms.

Follows a simple example.

```
illuminance( "specular&-crowd", P )
{
   /* Execute all lights in the "specular" category but omit
      the ones that are also in the "crowd" category. */
}
```

Message Passing

`illuminance` can take additional parameters, as shown in the general form above. These optional parameters are used for *message* passing and *forward message passing*. Forward message passing is performed using the special `send:light:` parameter and is used to set some given light parameter to some given value *prior* to light evaluation. For example,

```
uniform float intensity = 2;
illuminance( P, ..., "send:light:intensity", intensity )
{
     ... statements ...
}
```

Will set the *intensity* parameter of the light source to 2 and execute the light (overriding the value in light's parameter's list). The parameter must have the same type and same class (see Section 6.1.5 [Variable Classes], page 71) in order for the forward message passing to work.

Getting values back from a light source is done through the same mechanism by using the `light:` parameter prefix. For example,

```
/* Some default value in case light source has no "cone_angle" parameter */
float cone_angle = -1;
illuminance( P, ..., "light:cone_angle", cone_angle )
{
    /* do something with the "cone_angle" parameter ... */
}
```

Will retrieve the *cone_angle* parameter from the light source.

Both forward and backward message passing can be combined in the same `illuminance` loop:

```
/* Some default value in case light source has no "cone_angle" parameter */
uniform float intensity = 2;
float cone_angle = -1;
illuminance( P, ..., "send:light:intensity", intensity,
                     "light:cone_angle", cone_angle )
{
    /* do something with the "cone_angle" parameter ... */
```

```
}
```

NOTE: Message passing is a powerful feature in the shading language but one has to be aware that improper use could complicate the rendering pipeline *fundamentally* since it introduces a bi-directional flow of data between light sources and surface shaders.

Working Principles

The mechanics of `illuminance` are straightforward: loop over all active lights[8] withing the specified category and evaluate them to compute *Cl*, *Ol* and *L* (see Section 6.2 [Predefined Shader Variables], page 84). These variables are automatically available in the scope of all `illuminance` loops. Note that the result of light evaluation is cached so that calling `illuminance` again within the same evaluation context doesn't trigger re-evaluation of light sources. This optimization feature can be disabled using a special '`lightcache`' argument to `illuminate`. In the example below, the evaluation light cache will be flushed and the light sources will be re-evaluated.

```
illuminance( ..., "lightcache", "refresh" )
{
    ... statements ...
}
```

Flushing the light cache can have a sever performance impact, it is advised not to touch it unless necessary.

NOTE: Some shadeops contain *implicit* `illuminance` loops. These shadeops are: `specular`, `specularstd`, `diffuse` and `phong`. These are all described in Section 6.4.4 [Lighting], page 94.

The `illuminate` Construct

The `illuminate` construct is only defined in light source shaders and serves as an abstraction for positional light casters. In a way, it could be seen as the inverse of an `illuminance` construct. There are two ways to use `illuminate`:

1. Non-oriented. Such as a statement is meant to cast light in all directions.

   ```
   illuminate( point position )
       statement
   ```

2. Oriented. Such a statement will only cast light from a given *position* and along a given *axis* and *angle*. Surface points that are outside the cone formed by $< position, axis, angle >$ will not be lit (*Cl* will be set to zero, see below).

   ```
   illuminate( point position, vector axis, float angle )
       statement
   ```

Inside the `illuminate` construct, three standard variables are of interest:

1. The `L` variable. This is set by the renderer to $Ps - P$. This means that L is a vector pointing from light source's position to the point on the surface being shaded. The length of L is thus the distance between the light and the point on the surface.
2. The `Cl` variable. This variable is the actual color of the light and should be set inside the construct to the desired intensity.
3. The `Ol` variable. This is the equivalent of `Oi` and describes *light opacity*. It is almost never used and has been deprecated in the context of shader objects (see Section 6.1.1.2 [Shader Objects], page 62).

Listing Listing 6.2 shows how to write a standard point light. Note that the position given to `illuminate` is the origin of the shader space (see Table 6.3) and not some parameter passed to the

[8] Lights are listed in the same order as issued during scene description.

light; this is desirable since the point light can be placed anywhere in the scene using `RiTranslate` (or similar) instead of specifying a parameter.

```
light pointlight( float intensity = 1 ; color lightcolor = 1 )
{
    illuminate( point "shader" 0 )
    {
        Cl = intensity * lightcolor / (L.L);
    }
}
```

Listing 6.2: An example of a standard point light.

The `solar` Construct

This construct is similar to `illuminate` but describes light cast by *distant* light sources. It also has two forms albeit only one is partially supported by *3Delight*:

1. Non-directional. This form decribes light coming from all points at infinity (such as a light dome).

   ```
   solar( )
       statement
   ```

   Correctly implementing the `solar` constructs implies integration over all light direction inside shadeops such as `specular()` and `diffuse()`[9] . This is not yet implemented in *3Delight* and the statement above is replaced by:

   ```
   solar( I, 0 )
       statement
   ```

2. Directional. Describes light coming from a particular direction and covering some given solid angle.

   ```
   solar( vector axis, float angle )
       statement
   ```

   For the same reason as in the case above, *3Delight* doesn't consider the *angle* variable and considers this form as:

   ```
   solar( vector axis, 0 )
       statement
   ```

An example directional light is listed in Listing 6.3. This light source cast lights towards the positive z direction and has to be properly placed using scene description commands to light in any desired direction.

```
light directionallight( float intensity = 1 ; color lightcolor = 1 )
{
    solar( vector "shader" (0,0,1), 0 )
    {
        Cl = intensity * lightcolor;
    }
}
```

Listing 6.3: An example directional light using `solar`.

[9] And some sampling for custom made `illuminance` loops.

The `gather` Construct

This construct explicitly relies on ray-tracing[10] to collect illumination and other data from the surrounding scene elements. In other words, this construct collects surrounding illumination through *sampling*. Incidentally, `gather` is well suited to integrate arbitrary reflectance models over incoming *indirect* light (light that is reflected from other surfaces). The general form of a `gather` loop is as follow:

```
gather( string category, point P, dir, angle, samples, ... )
    statement
[else
    statement]
```

The $< P, dir, angle >$ triplet specifies the sampling cone and *samples* specifies the number of samples to use (more samples will give more accurate results). The angle should be specified in radians and is measured from the axis specified by *dir*: an angle of zero implies that all rays are shot in direction *dir* and an angle of $PI/2$ casts rays over the entire hemisphere. `gather` stores its result in the specified optional variables which depend on the specified *category*. The table below explains all supported categories and related output variables.

`samplepattern`
: This category is meant to run the loop without ray-tracing and without taking any further action but to provide ray's information to the user. This is useful to get the sampling pattern of `gather` to perform some particular operation. In this mode, one have access to the following output variables :

  'ray:origin'
  : Returns ray's origin.

  'ray:direction'
  : Returns ray's direction. Not necessarily normalized.

  'sample:randompair'
  : Returns a vector which contains two *stratified* random variables that are well suited for sampling. Only the x and y component of the returned vector are relelevant and the z component is set to zero.

  'effectivesamples'
  : Returns a varying float which is the number of samples effectively used by *3Delight*. This may be different from the requested number of samples.

  Listing Listing 6.4 combines '`samplepattern`' and simple transmission queries (see [transmission shadeop], page 98) to compute the ambient occlusion.

[10] Interestingly, this is the only construct in the RenderMan shading language that explicitly relies, in its standard form, on a specific rendering algorithm.

```
color trans = 0;
uniform float num_samples = 20;
uniform float max_dist = 10;
point ray_origin = 0;
vector ray_direction = 0;
float effectivesamples = 0;
gather( "samplepattern", P, N, PI/2, num_samples,
    "effectivesamples", effectivesamples,
    "ray:direction", ray_direction,
    "ray:origin", ray_origin )
{
}
else
{
    vector normalized_ray_dir = normalize( ray_direction );
    trans += transmission(
        ray_origin, ray_origin + normalized_ray_dir*max_dist );
}
trans /= effectivesamples;
```

Listing 6.4: An example usage of the '`samplepattern`' category in `gather`.

Note that in this mode, it is the `else` branch of the construct which is executed.

`illuminance`

In this case, `gather` uses ray tracing to perform the sampling. Additionally to variables available to the *samplepattern* category, any variable from surface, displacement or volume shaders can be collected:

'`surface:`*varname*'
: Returns the desired variable from the context of the surface shader that has been hit by the gather ray. Variables' values are taken *after* the execution of the shader. The typical variable is `surface:Ci` but other variables, such as `surface:N` or `surface:P`, are perfectly valid[11]. Additionally, '`varname`' can be any *output* variable declared in the shader.

'`displacement:`*varname*'
'`volume:`*varname*'
: Returns the desired variable from the context of the displacement or volume shader that has been evaluated for the gather ray. All comments for the surface case above also apply here. All returned variables are those encountered at the *closest surface to be hit*.

'`ray:length`'
: Returns distance to closest hit.

[11] All variable names are case sensitive.

The following example demonstrates how to compute a simple ambient occlusion effect.

```
float occ = 0;
uniform numSamples = 20;
float smpDiv = 0;
gather( "illuminance", P, Nf, PI/2, numSamples, "effectivesamples", smpDiv )
    ;/* do nothing ... */
else
    occ += 1;

occ /= smpDiv;
```

`environment:environmentname`

The '`environment`' category is used to perform *importance sampling* on a specified environment map. Importance sampling can be used to implement image based lighting as shown in the `envlight2.sl` shader provided with the *3Delight* package. Note that no ray-tracing is performed: importance-sampled rays are returned and it is up to the user to perform ray-tracing if needed. This category unveils three optional parameters (further explained in Section 7.4.4 [Image Based Lighting], page 133):

'`environment:color`'
: Return the color from the environment map.

'`environment:solidangle`'
: Returns the sampled solid angle in the environment map. This is the size of the sampled region and is usually inversely proportional to the number of sampled rays - more samples will find finer details in the environment map.

'`environment:direction`'
: The direction pointing towards the center of the sampled region in the environment map. Note that '`ray:direction`' holds a randomized direction inside the region and this direction is suitable for sampling.

An example on how to use these variables is shown in Listing 7.5.

`pointcloud:pointcloudfile`

Another special form of the `gather()` construct can be used to gather points in a previously generated point cloud. This is better illustrated in Example 6.1.

```
/* Average 32 samples from a point cloud surrounding the current
   point using gaussian filtering.
   Assume point cloud has been saved in "object" space.
*/
point object_P = transform( "object", P );
normal object_N = ntransform( "object", N );

string category = concat( "pointcloud", ":", texturename );

color surface_color = 1, sum_color = 0;
float total_atten = 0;

point position = 0;;
float point_radius;

uniform float radius = 0.1;

gather( category, object_P, object_N, PI, 32,
    "point:surface_color", surface_color,
    "radius", radius,
    "point:position", position )
{
    float dist = distance(object_P, position);

    dist = 2*dist / radius;
    float gaussian_atten = exp( -2. * dist * dist );
    total_atten += gaussian_atten;
    sum_color += gaussian_atten * surface_color;
}
/* normalize final color */
sum_color *= total_atten;
```

Example 6.1: Using gather() to access point cloud data.

As shown in the above example, specifying '`pointcloud:`*`filename`*' as the category to `gather()` will give access to the following variables:

'`point:position`'
: Position of the currently processed point

'`point:normal`'
: Normal of the currently processed point

'`point:radius`'
: Radius of the currently processed point

'`point:`*`varname`*'
: Any variable stored in the point cloud

Note:

- Points from the point cloud will be listed in `gather()` from the closest to the furthest, relatively to the provided position *P*.
- There is no guarantee that the actual number of points listed will be equal to the number of points requested. This can happen for example if there are not enough points in point cloud (still unlikely) or the provided *radius* limits the total number of points available in a certain position.

- Points outside the radius can still be listed if their extent is part of the gather sphere (points in a point cloud are defined by a point *and a radius*).

- The *N* and *angle* parameter provided to the point cloud version of `gather()` are not considered.

`gather` accepts many optional input parameters and these are all explained in Table 6.10. Additionally, `gather` accepts the '`distribution`' parameter to sample arbitrary distributions[12], this parameter is described in Table 6.11. `gather` also supports the '`samplebase`' parameter as described in Table 6.11. The next code snippet illustrates how to use an environment map as a sampling distribution to compute image based occlusion:

```
float occ = 0;
uniform numSamples = 20;
gather( "illuminance", P, Nf, PI/2, numSamples,
    "distribution", "lightprobe.tdl" )
{
    ;/* do nothing ... */
}
else
    occ += 1;

occ /= numSamples;
```

6.1.7.4 Functions

As in many other languages, functions in the RenderMan shading language are re-usable blocks of code that perform a certain task. The general form of an RSL function is as follows:

```
return_type function_name ( parameters list )
{
    ... function body ...

    return return_value;
}
```

In traditional shaders (see Section 6.1.1.1 [Traditional Structure], page 62) functions have to be declared prior to their calling points. In class shaders (see Section 6.1.1.2 [Shader Objects], page 62), functions can be declared anywhere outside the class or inside the class, with functions declared in the class scope being accessible to other shaders. An example function is shown Listing 6.5.

[12] This parameter is somewhat redundant to the '`environment`' category. In the future, '`environment`' category will be replaced by a '`samplepattern`' category combined with '`distribution`' parameter

```
void print_error( string tex; )
    { printf( "error loading texture %s.\n", tex ); }

color tiled_texture( string texture_name, float u, v;
                     float num_tiles_x, num_tiles_y;
                     output float new_u, newv; )
{
    color red() { return color(1,0,0); }

    if( texture_name == "" )
    {
        print_error( texture_name );
        return red();
    }

    /* assuming u & v are in the range [0..1]. */
    new_u = mod(u, 1/num_tiles_x) * num_tiles_x;
    new_v = mod(v, 1/num_tiles_y) * num_tiles_y;

    return texture( texture_name, new_u, new_v );
}
```

Listing 6.5: An example illustrating the syntax of a RSL function.

Some noteworthy points about the example above:

- Functions can have multiple exit points.
- Functions can return no value by using the `void` keyword.
- Functions can return their values using one or more `output` parameters.
- Similarly to the *Pascal* programming language, functions can be declared *inside* other functions or *in any new scope*.

One particularly useful concept in the RenderMan shading language is that functions are *polymorphic*[13], meaning that the same function can be delcared with different *signatures* to accept different parameter types. For example:

```
float a_plus_b( float a, b; ) { return a+b; }
color a_plus_b( color a, b; ) { return a+b; }
```

[13] More formally, the RenderMan shading languages imeplemtns *ad-hoc polymorphism* through method overrides.

6.1.7.5 Methods

Methods are declared as normal functions with the following differences:

- They are declared inside a shader object (see Section 6.1.1.2 [Shader Objects], page 62).
- They are preceded by the `public` keyword.
- Methods are *not inlined* as usual functions and are *callable from other shaders*.
- Methods cannot be declared inside other methods.

The general form for a RSL method is as follows:

```
public return_type function_name ( parameters list )
{
    ... function body ...

    return return_value;
}
```

For example,

```
public void surface( output color Ci, Oi; )
{
}
```

Declares the standard `surface()` entry point in a shader object.

6.2 Predefined Shader Variables

For each shader, some predefined variables are provided for input and output purposes. Here are the lists of variables for each shader type. Variables with a '`*`' at the end of their description are available only inside `illuminance` constructs (see [The Illuminance Construct], page 74) .

6.2.1 Predefined Surface Shader Variables

Name	Type	Storage	Description
Cs	`color`	`varying`	Surface color.
Os	`color`	`varying`	Surface opacity.
P	`point`	`varying`	Surface position.
dPdu	`vector`	`varying`	Derivative of surface position along *u*.
dPdv	`vector`	`varying`	Derivative of surface position along *v*.
N	`normal`	`varying`	Surface shading normal.
Ng	`normal`	`varying`	Surface geometric normal.
u, *v*	`float`	`varying`	Surface parameters.
du, *dv*	`float`	`varying`	Change of surface parameters.
s, *t*	`float`	`varying`	Surface texture coordinates.
E	`point`	`uniform`	Position of the eye.
I	`vector`	`varying`	Incident ray direction.
ncomps	`float`	`uniform`	Number of color components.
time	`float`	`uniform`	Current shutter time, as specified by `RiShutter`.
dtime	`float`	`uniform`	The amount of time covered by this shading sample.
dPdtime	`vector`	`varying`	How the surface position P is changing per unit time, as described by motion blur in the scene.
Ci	`color`	`varying`	Same as Cs.
Oi	`color`	`varying`	Same as Os.
L	`vector`	`varying`	Incoming light ray direction. *
Cl	`color`	`varying`	Incoming light ray color. *
Ol	`color`	`varying`	Incoming light ray opacity. *
Output			
Ci	`color`	`varying`	Incident ray color.
Oi	`color`	`varying`	Incident ray opacity.
P	`point`	`varying`	Displaced surface position.
N	`vector`	`varying`	Displaced surface shading normal.

Table 6.5: Predefined Surface Shader Variables.

6.2.2 Predefined Light Shader Variables

Name	Type	Storage	Description
P	`point`	`varying`	Surface position on the light.
dPdu	`vector`	`varying`	Derivative of surface position along u.
dPdv	`vector`	`varying`	Derivative of surface position along v.
N	`normal`	`varying`	Surface shading normal on the light.
Ng	`normal`	`varying`	Surface geometric normal on the light.
u, *v*	`float`	`varying`	Surface parameters.
du, *dv*	`float`	`varying`	Change of surface parameters.
s, *t*	`float`	`varying`	Surface texture coordinates.
L	`vector`	`varying`	Outgoing light ray direction. *
Ps	`point`	`varying`	Position being illuminated.
E	`point`	`uniform`	Position of the eye.
ncomps	`float`	`uniform`	Number of `color` components.
time	`float`	`uniform`	Current shutter time, as specified by `RiShutter`.
dtime	`float`	`uniform`	The amount of time covered by this shading sample.
I	`vector`	`varying`	Incident ray direction. *
Output			
Cl	`color`	`varying`	Outgoing light ray color.
Ol	`color`	`varying`	Outgoing light ray opacity.

Table 6.6: Predefined Light source Variables

6.2.3 Predefined Volume Shader Variables

Name	Type	Storage	Description
P	`point`	`varying`	Light ray endpoint.
I	`vector`	`varying`	Ray direction (pointing toward *P*).
E	`point`	`uniform`	Position of the eye.
Ci	`color`	`varying`	Ray color.
Oi	`color`	`varying`	Ray opacity.
ncomps	`float`	`uniform`	Number of color components.
time	`float`	`uniform`	Current shutter time, as specified by `RiShutter`.
dtime	`float`	`uniform`	The amount of time covered by this shading sample.
dPdu	`vector`	`varying`	Derivative of surface position along u.
dPdv	`vector`	`varying`	Derivative of surface position along v.
N	`normal`	`varying`	Surface shading normal.
Ng	`normal`	`varying`	Surface geometric normal.
u, *v*	`float`	`varying`	Surface parameters.
du, *dv*	`float`	`varying`	Change of surface parameters.
s, *t*	`float`	`varying`	Surface texture coordinates.
Cs	`color`	`varying`	Same as *Ci*.
Os	`color`	`varying`	Same as *Oi*.
Output			
Ci	`color`	`varying`	Attenuated ray color.
Oi	`color`	`varying`	Attenuated ray opacity.

Table 6.7: Predefined Volume Shader Variables

6.2.4 Predefined Displacement Shader Variables

Name	Type	Storage	Description
P	`point`	`varying`	Surface position.
dPdu	`vector`	`varying`	Derivative of surface position along u.
dPdv	`vector`	`varying`	Derivative of surface position along v.
N	`normal`	`varying`	Surface shading normal.
Ng	`normal`	`varying`	Surface geometric normal.
I	`vector`	`varying`	Incident ray direction.
E	`point`	`uniform`	Position of the eye.
u, *v*	`float`	`varying`	Surface parameters.
du, *dv*	`float`	`varying`	Change of surface parameters.
s, *t*	`float`	`varying`	Surface texture coordinates.
ncomps	`float`	`uniform`	Number of color components.
time	`float`	`uniform`	Current shutter time, as specified by `RiShutter`.
dtime	`float`	`uniform`	The amount of time covered by this shading sample.
dPdtime	`vector`	`varying`	How the surface position P is changing per unit time, as described by motionblur in the scene.
Output			
P	`point`	`varying`	Displaced surface position.
N	`normal`	`varying`	Displaced surface shading normal.

Table 6.8: Predefined Displacement Shader Variables

6.2.5 Predefined Imager Shader Variables

Name	Type	Storage	Description
P	point	varying	Pixel raster position.
Ci	color	varying	Pixel color.
Oi	color	varying	Pixel opacity.
u, *v*	float	varying	Surface parameters. Set to pixel's raster coordinates.
s, *t*	float	varying	Surface texture coordinate. Set to pixel's raster coordinates.
du, *dv*	float	varying	Change of surface parameters. (`du` = 1.0 / x resolution, `dv` = 1.0f / y resolution).
alpha	float	uniform	Fractional pixel coverage.
ncomps	float	uniform	Number of color components.
time	float	uniform	Shutter open time.
dtime	float	uniform	The amount of time the shutter was open.
Output			
Ci	color	varying	Output pixel color.
Oi	color	varying	Output pixel opacity.
alpha	float	uniform	Output pixel coverage.

Table 6.9: Predefined Imager Shader Variables

6.3 Predefined Shader Parameters

Some shader parameters have a special meaning and should not be used to for a different purpose. These are all listed in the table below.

`uniform string __category`
: Only meaningful as as light shader parameters, this string specifies the categories to which a light source belongs. Light categories are used to control what light sources are considered by constructs such as `illuminance()` and `solar()`. For example:

```
light donothing(
    string __category = "backlight,fog"; )
{
    shader body
}
```

More about light categories in [Light Categories], page 74 and a real-world example is shown in Listing 7.4.

`{uniform|varying} float __nonspecular`
: Only meaningful as a light shader parameter, this scalar specifies to what extent this light contributes in `specular()`, `specularstd()` and `phong()`. Valid range is [0..1], 1 disabling completely the light in the aforementioned shadeops.

`{uniform|varying} float __nondiffuse`
: Only meaningful as a light shader parameter, this scalar specifies to what extent this light contributes in the `diffuse()` shadeop. Valid range is [0..1], 1 disabling completely the light in the `diffuse()`.

Noteworthy details about `__nonspecular` and `__nondiffuse`:

1. These two parameters can be either `uniform` or `varying`. It is wasteful to declare them as `varying` if they do not change across the lit surface (which is almost always the case).
2. If these parameters are `uniform`, it might be wiser to use light categories instead (see [Light Categories], page 74) because light categories will exclude light sources *prior* to evaluation whereas these special parameters will be considered *after* light source evaluation, potentially leading to wasteful work.

6.4 Standard Functions

3Delight supports all the standard Shading Language (SL) built-in shadeops and constructs. The complete set of standard shadeops is listed below. Descriptions are kept brief, if any, since shadeops are already described in great details in the RenderMan specifications. Shadeops marked with (*) contain specific extensions and those marked with (**) are not part of the current standard. It is also possible to link shaders with C or C++ code to add new shadeops, see Section 10.1.2 [RSL Plug-ins], page 209.

3Delight supports all predefined shader variables. It doesn't follow strictly the 3.2 RenderMan Interface specification though: some shaders have access to more variables.

6.4.1 Mathematics

```
float radians ( float degrees )
float degrees ( float radians )
float sin ( float radians )
float asin ( float a )
float cos ( float radians )
```

`float acos ( float a )`
`float tan ( float radians )`
`float atan ( float a )`
`float atan ( float y, float x )`

`float sqrt ( float x )`
Returns the square root of x. The domain is [0..*infinity*]. If $x < 0$ this shadeop will return 0.

`float inversesqrt ( float x )`
Returns $1.0/sqrt(x)$. The domain of this function is [0..*infinity*[. If $x < 0$ this shadeop will return *infinity*[1].

`float pow ( float x, y )`
Returnx $x * *y$.

`float exp ( float x )`
Returns `pow`(e, x), the natural logarithm of x.

`float log ( float x [, base] )`
Returns the natural logarithm of x. If the optional *base* is specified, returns the logarithm of x in the specified base. The domain of this function is [0..*infinity*[. If $x < 0$ this shadeop will return $-infinity$.

`float sign ( float x )`
Returns:
- -1 if $x < 0$
- 1 if $x >= 0$

`float mod ( float x, y )`
Returns x *modulus* y. More formally, returns a value in the range [0..y], for which $mod(x, y) = x - n * y$, for some integer n.

`float abs ( float x )`
Returns the absolute value of x.

`float floor ( float x )`
`float ceil ( float x )`
`float round ( float x )`

`type min ( type x, y, ... )`
`type max ( type x, y, ... )`
`type clamp ( type x, min, max )`
`min()` and `max()` take two or more arguments of the same type and return the minimum and maximum values, respectively. `clamp()` will return *min* if x is smaller than *min*, *max* if x is greater than `max` and x otherwise. *type* can be a float or a 3-tuple. When running on 3-tuples, these shadeops will consider one component at a time (eg. `min(point(1,2,3), point(4,3,2))` will return `point(1,2,2)`);

`float step ( float min, value )`
`float smoothstep ( float min, max, value )`
`step()` returns 0 if *value* is less than *min*; otherwise it returns 1. `smoothstep()` returns 0 if *value* is less than *min*, 1 if *value* is greater than or equal to *max*, and returns a Hermite interpolation between 0 and 1 when *value* is in the range [*min*..*max*[.

[1] A very large floating point number is actually returned, it is equal to `FLT_MAX` (approximately 3.4e38)

`type mix ( type x, y, alpha )`
Returns x*(1-*alpha*) + y**alpha*. For multi-component types (color, point, ...), the operation is performed for each component.

`float filterstep ( float edge, value, ... )`
`float filterstep ( float edge, value1, value2, ... )`
Similar to `step()` but the return value is filtered over the area of the micropolygon being shaded. Useful for shader anti-aliasing. Filtering kernel is selected using the "*filter*" optional parameter. Recognized filters are '`gaussian`', '`box`', '`triangle`' and '`catmull-rom`'. Default is '`catmull-rom`'. If two values are provided, return value is filtered in the range [value1..value2].

`type spline ( [string basis ;] float value; type begin, p1, ..., pn, end)`
`type spline ( [string basis ;] float value; type p[])`
Interpolates a point on a curve defined by *begin*, *p1* to *pn*, and *end* control points. *value* should lie between 0 and 1, otherwise the return value will be clamped to either *p1* or *pn*. The default *basis* is the '`catmull-rom`' spline[2]. Other possible splines are '`bezier`', '`bspline`', '`hermite`' and '`linear`'. Any unknown spline is assumed to be '`linear`'. Any recognized spline may be prefixed by '`solve`', such as '`solvecatmull-rom`'. In such a case, the shadeop becomes a root solver and may be used as an invert function.

`type Du ( type x )`
`type Dv ( type x )`
`type Deriv ( type num; float denom )`
`Du()` and `Dv()` compute the parametric derivative of the given expressions with respect to the *u* and the *v* parameters of the underlying surface[3].

6.4.2 Noise and Random

`type noise ( float x )`
`type noise ( float x, y )`
`type noise ( point Pt )`
`type noise ( point Pt; float w )`
1D, 2D, 3D and 4D noise function. *type* can be `float`, `color`, `point` or `vector`.

`type pnoise ( float x, period )`
`type pnoise ( float x, y, xperiod, yperiod )`
`type pnoise ( point Pt, Ptperiod )`
`type pnoise ( point Pt; float w; point Ptperiod; float wperiod )`
Same as noise but has periodicity *period*. Maximum period is 256.

`type cellnoise ( float x )`
`type cellnoise ( float x, y )`
`type cellnoise ( point Pt )`
`type cellnoise ( point Pt, float w )`
Cellular noise functions (1D, 2D, 3D and 4D).

`type random ( )`
Returns a random `float`, `color` or `point`. Returned range is [0..1]. Can return `uniform` or `varying` values. Here is a trick to put a random color in each grid of micropolygons:

[2] This spline may also be selected with '`catrom`' or '`cr`'.
[3] `Deriv()` computes the following expression: $\frac{Du(num)}{Du(denum)} + \frac{Dv(num)}{Dv(denum)}$

```
uniform float red = random();
uniform float green = random();
uniform float blue = random();

Ci = color( red, green, blue );
```

6.4.3 Geometry, Matrices and Colors

`float xcomp` (*point* `Pt`)
`float ycomp` (*point* `Pt`)
`float zcomp` (*point* `Pt`)
`void setxcomp` (*output point* `Pt`; *float* `x`)
`void setycomp` (*output point* `Pt`; *float* `y`)
`void setzcomp` (*output point* `Pt`; *float* `z`)
Gets or sets the x, y, or z component of a point (or vector).

`float comp` (*matrix* `M`; *float* `row`, `col`)
Returns `M[row,col]`.

`void setcomp` (*output matrix* `M`; *float* `row`, `col`, `x`)
`M[row,col]` = *x*.

`float comp` (*color* `c`; *float* `i`)
`void setcomp` (*output color* `c`, *float* `i`, `x`)
Returns or sets the "*i*th" component of the given color.

`point transform` (*string* [`fromspace`,] `tospace`; *point* `Pt`)
`point transform` ([*string* `fromspace`;] *matrix* `M`; *point* `Pt`)
`point transform` (*string* [`fromspace`,] `tospace`; *vector* `V`)
`point transform` ([*string* `fromspace`;] *matrix* `M`; *vector* `V`)
`point transform` (*string* [`fromspace`,] `tospace`; *normal* `N`)
`point transform` ([*string* `fromspace`;] *matrix* `M`; *normal* `N`)
Transforms a `point`, `vector` or `normal` from a given space (*fromspace*) to another space (*tospace*). If the optional *fromspace* is not given, it is assumed to be the '`current`' space. Refer to Table 6.3 for the complete list of valid space names.

`vector vtransform` (*string* [`fromspace`,] `tospace`; *vector* `V`)
`vector vtransform` ([*string* `fromspace`;] *matrix* `M`; *vector* `V`)
`normal ntransform` (*string* [`fromspace`,] `tospace`; *normal* `Nr`)
`normal ntransform` ([*string* `fromspace`;] *matrix* `M`; *normal* `Nr`)
Same as `transform()` above but specifically selects a transform on normals or vectors (no polymorphism).

NOTE**:** One should use `transform()` instead since that shadeop will correctly select the appropriate transform depending on the parameter type.

`color ctransform` (*string* [`fromspace`,] `tospace`; *color* `src_color`)
Transforms color *src_color* from color space *fromspace* to color space *tospace*. If the optional *fromspace* is not specified, it is assumed to be '`rgb`'. *3Delight* recognizes the following color spaces: RGB, HSV, HSL, YIQ and XYZ[4]. If an unknown color space is given, *3Delight* returns *src_color*.

[4] When converting to and from XYZ color space, *3Delight* considers that RGB tristimulus values conform to Rec. 709 (with a white point of D_{65}).

`float distance` (*point* `Pt1`, `Pt2`)
`float length` (*vector* `V`)
`vector normalize` (*vector* `V`)
`distance()` returns the distance between two points. `length()` returns the length (norm) of the given vector. `normalize()` divides the given vector by its length (making it of unit length). All three operations involve a square root.

`float ptlined` (*point* `Pt1`, `Pt2`, `Q`)
Returns minimum distance between a point *Q* and a *segment* defined by *Pt1* and *Pt2*.

`point rotate` (*point* `Q`; *float* `angle`; *point* `Pt1`, `Pt2`)
Rotates a point *Q* around the line defined by *Pt1* and *Pt2*, by a given *angle*. New point position is returned. Note that *angle* is assumed to be in radians.

`float area` (*point* `Pt` [; *string strategy*])
Returns `length(Du(Pt)^Dv(Pt))`, which is approximately the area of one micro-polygon on the surface defined by *Pt*. The *strategy* variable can take two values:

'`shading`' Compute area of the micro-polygon based on surface derivatives. This will produce smoothly varying areas *if smooth derivatives are enabled* (see Section 5.2.7 [other attributes], page 51).

'`dicing`' Compute area of micro-polygons using their geometry, regardless of smooth derivatives.

If no *strategy* is supplied, '`shading`' will be assumed.

`vector faceforward` (*vector* `N`, `I`[, `Nref`])
Flips N, if needed, so it faces in the direction opposite to I. *Nref* gives the element surface normal; if not provided, *NRef* is set to *Ng*.

`vector reflect` (*vector* `I`, `N`)
Returns the vector which is the reflection of *I* around *N*. Note that *N* must be of unit length.

`vector refract` (*vector* `I`, `Nr`; *float* `eta`)
Returns the refracted vector for the incoming vector *I*, surface normal *Nr* and index of refraction ratio *eta*. *Nr* must be of unit length.

`float depth` (*point* `Pt`)
Returns the normalized *z* coordinate of *Pt* in camera space. Return value is in the range [0..1] (0=near clipping plane, 1=far clipping plane). *Pt* is assumed to be defined in '`current`' space.

`normal calculatenormal` (*point* `Pt`)
Computes the normal of a surface defined by *Pt*. Often used after a displacement operation on *Pt*. Equivalent to `Du(Pt)^Dv(Pt)`, but faster.

`float determinant` (*matrix* `M`)
`matrix inverse` (*matrix* `M`)
`matrix translate` (*matrix* `M`; *point* `Tr`)
`matrix rotate` (*matrix* `M`; *float* `angle`; *vector* `axis`)
`matrix scale` (*matrix* `M`; *point* `Sc`)
Basic matrix operations. The *angle* parameter passed to `rotate()` is assumed to be in radians.

6.4.4 Lighting

`color ambient ( )`
Returns the contribution from ambient lights. A light is considered ambient if it does not contain an `illuminate()` or `solar()` statement. It is not available in lightsource shaders.

`color diffuse` (*vector* ***Nr***)
Computes the diffuse light contribution. Lights placed behind the surface element being shaded are not considered. *Nr* is assumed to be of unit length. Light shaders that contain a parameter named `uniform float __nondiffuse` are evaluated only if the parameter is set to 0. Not available in lightsource shaders (see Section 6.3 [Predefined Shader Parameters], page 89).

`color specular` (*vector* ***Nr***, ***V***; *float* ***roughness***) *
Computes the specular light contribution. Lights placed behind the object are not considered. *Nr* and *V* are assumed to be of unit length. Light shaders that contain a parameter named `uniform float __nonspecular` are evaluated only if the parameter is set to 0. Not available in lightsource shaders (see Section 6.3 [Predefined Shader Parameters], page 89).

`color specularbrdf` (*vector* ***L***, ***Nr***, ***V***; *float* ***roughness***)
Computes the specular light contribution. Similar to `specular()` but receives a *L* variable (incoming light vector) enabling it to run in custom `illuminance()` loops.

`color specularstd` (*normal* ***N***; *vector* ***V***; *float* ***roughness***)
This is the standard specular model described in all graphic books since *3Delight* implements its own specular model in `specular`.

```
color specularstd( normal N; vector V; float roughness )
{
    extern point P;
    color C = 0;
    point Nn = normalize(N);
    point Vn = normalize(V);

    illuminance(P, Nn, PI/2)
    {
        extern vector L;
        extern color Cl;

        vector H = normalize(normalize(L)+Vn);
        C += Cl * pow(max(0.0, Nn.H), 1/roughness);
    }

    return C;
}
```

Listing 6.6: `specularstd()` implementation.

`color phong` (*vector* ***Nr***, ***V***; *float* ***size***)
Computes specular light contribution using the Phong illumination model. *Nr* and *V* are assumed to be of unit length. As in `specular()`, this function is also sensitive to the `__nonspecular` light shader parameter. Not available in lightsource shaders.

`color bsdf` (*vector* `L`*; normal* `N`*; ...*)
Evalutes one of the built-in *BSDF* available in *3Delight*. *L* is the incoming light direction and *N* the surface normal. The optional parameters are the distribution-related ones from Section 6.4.5.2 [The trace Shadeop], page 97.

`void fresnel` (*vector* `I`*,* `N`*; float* `eta`*; output float* `Kr`*,* `Kt` [*; output vector* `R`*,* `T`])
Uses the Fresnel formula to compute the reflection coefficient *Kr* and refraction (or transmission) coefficient *Kt* given and incident direction *I*, the surface normal *N* and the relative index of refraction *eta*. Note that *eta* is the ratio of the index of refraction in the volume containing the incident vector to that of the volume being entered: a ray entering a water volume from a void volume would need an *eta* of approximately 1.0/1.3. Here is a noteworthy quotation from "The RenderMan Companion":

> In most cases, a ray striking a refractive material is partly reflected and partly refracted. The function `fresnel()` calculates the respective fractions. It may also return the reflected and refracted direction vectors, so that is subsumes `refract()`.

If *R* and *T* are supplied, they are set to the direction vector of the reflected and the transmitted (refracted) ray, respectively.

6.4.5 Ray Tracing

6.4.5.1 Common Parameters

All the ray tracing shadeops can receive a number of common optional parameters, these are described in Table 6.10. Specific optional parameters are described with each shadeop individually.

Name	Type	Default	Description
"label"	`uniform string`	""	Specifies a label to attach to the ray. Refer to Section 7.3.3 [Ray Labels], page 127.
"subset"	`uniform string`	""	Specifies a subset of objects which are included in ray tracing computations. Refer to Section 7.3.4 [Trace Group Membership], page 128.
"bias"	`uniform float`	-1	Specifies a bias for ray's starting point to avoid potentially erroneous intersections with emitting surface. A value of '`-1`' forces *3Delight* to take the default value as specified by `Attribute "trace" "bias"`.
"hitmode"	`uniform string`	"default"	Can be used to override the shading mode set on the intersected object with [hitmode], page 46. Only applies when the object's visibility was set with the "diffuse" "specular" or "int transmission" visibility attributes. Refer to [Primitives Visibility and Ray Tracing Attributes], page 45.
"hitsides"	`uniform string`	"default"	This can be used to override which sides of surfaces a ray can hit. The default is to hit both sides of surfaces with `RiSides 2` and the front side of surfaces with `RiSides 1`. A value of '`reversed`' will hit both sides of surfaces with `RiSides 2` and the back side of surfaces with `RiSides 1`. Other possible values are '`front`', '`back`' and '`both`' which act the same regardless of `RiSides`.
"raytype"	`uniform string`	"default"	Can be either '`specular`', '`diffuse`' or '`transmission`'. This is accepted by `trace()`, `occlusion()`, `indirectdiffuse()` and `gather()` to override the default ray type. The ray type changes which visibility attributes are considered to know if a ray can hit a given primitive.
"raypruning"	`uniform float`	1	This option enables (enabled by default) or disables (not advised) automatic ray-pruning in the ray-tracer. **Disabling ray pruning may result in catastrophic performance if shaders are not written properly**.
"weight"	`varying color`	1	This parameter can be used to adjust the importance of the rays traced by a shadeop. This eventually allows fewer rays to be traced where their impact on the image is less. As a general rule, its value should be the factor which will multiply the result of the shadeop. **Proper use of this parameter can provide large gains of performance**.

Table 6.10: Common optional parameters to ray tracing functions.

6.4.5.2 The trace Shadeop

`color trace` (*point* `Pt`; *vector* `R` [; *output float* `dist`]; ...)

The trace shadeop is a general tool for collecting incoming light at point Pt on the surface. In its simplest form, it will collect light from direction R.

If the optional output parameter *dist* is specified, it will contain the distance to the nearest intersection point or a very large number ($> 1e30$) when no intersections are found. Note that all parameters must be in "current" space. `trace()` accepts several optional named parameters which are explained below.

`float samples`
: Specifies the number of samples to use. Higher sample counts will improve quality, at the cost of performance.

`float maxdist`
: Specifies a distance after which no intersections are checked. The default value is $1e38$ which in practice means that there is no maximum distance for intersections.

`string distribution`
: This parameter specifies which BRDF `trace()` will use to gather light from the scene. The possible values are `uniform`, `cosine`, `oren-nayar`, `blinn`, `ashikhmin-shirley`, `cook-torrance` and `ward`. The default value is `uniform`. Each distribution requires some specific additional parameters listed below. See also the [bsdf shadeop], page 95 which allows the same distributions to be used in an illuminance loop with direct light.

  `float samplecone`
  : Specifies an angle in radians which, together with Pt and R, describes a cone in which `trace()` will gather light. The angle specified is the cone's half angle, meaning $PI/2$ will sample a hemisphere. This applies to the `uniform` and `cosine` distributions. The default value is 0.

  `vector wo` This vector specifies the viewer direction for the `oren-nayar`, `blinn`, `ashikhmin-shirley`, `cook-torrance` and `ward` distributions. This is typically *-I*. Note that when this parameter is needed, the `R` parameter is expected to be the surface normal.

  `float roughness`
  : Specifies the roughness of the surface for the `oren-nayar`, `blinn`, `ashikhmin-shirley`, `cook-torrance` and `ward` distributions. The valid range is between 0 and 1, except for `oren-nayar` which has no upper limit (in practice, there is little difference above $PI/2$).

  `float roughnessv`
  : Specifies the roughness in the second axis for the anisotropic `ashikhmin-shirley` and `ward` distributions.

  `vector udir`
  : Specifies the orientation of the u direction for the anisotropic `ashikhmin-shirley` and `ward` distributions. This is the directon where `roughness` applies. `roughnessv` is applied in a perpendicular direction on the surface. The `udir` vector does not need to be strictly tangent to the surface as *3Delight* will project it on the surface itself.

  `float eta` Specifies an index of refraction to compute a fresnel effect for the `blinn`, `ashikhmin-shirley` and `cook-torrance` distributions. This is optional

and not specifying it is visually equivalent to using a very large index of refraction but faster to compute.

`uniform string environmentmap`
Specifies an environment map to use as incoming light in directions where there is no geometry.

`uniform string environmentspace`
Specifies a coordinate system to orient the environment map.

`color environmenttint`
Specifies a multiplicative factor for the content of the environment map.

`output varying color transmission`
Returns the transmission in the traced direction. This takes into account both coverage (if sampling a cone) and the opacity of the surfaces hit by rays. It is not influenced by the specified environment map. There is no speed penalty to get the transmission from `trace()`

IMPORTANT: The returned value is exactly the same as what would have been returned by the [transmission shadeop], page 98 with the important difference that `transmission()` defaults to intersecting objects visible to *transmission* rays while `trace()` uses *specular* rays.

`output varying color environmentcontribution`
Returns the contribution of the environment map to the output of trace.

EXAMPLE

```
/* Trace a ray, only considering intersections closer than 10 units.
   Intersection distance is stored in ``dist'' and intersection
   color in ``c''. If no intersection, dist will be very large. */
float dist;
color c = trace( P, Nf, dist, "maxdist", 10 );
```

6.4.5.3 Other Ray Tracing Shadeops

`float trace` (*point* `Pt`; *vector* `R`)
This is an obsolete form which returns the distance to the nearest intersection, when looking from *Pt* in the direction specified by the unit vector *R*. *Pt* and *R* must lie in '`current`' space. This function intersects objects that are visible to *transmission* rays and is strictly equivalent to:

```
float distance;
trace( P, dir, distance, "type", "tranmission" );
```

`color transmission` (*point* `Pt1`, `Pt2`, ...)
Determines the visibility between *Pt1* and *Pt2* using ray tracing. Returns `color 1` if unoccluded and `color 0` if totally occluded. In-between values indicate the presence of a translucent surface between *Pt1* and *Pt2*. Only objects tagged as visible to *transmission* rays are considered during the operation (using `Attribute "visibility" "transmission"`, Section 5.2 [attributes], page 44). *Pt1* and *Pt2* must lie in '`current`' space. When tracing area light shadows, it is usually better for performance to trace from the surface to the light. In addition to standard ray tracing optional parameters (see Table 6.10), this shadeop also accepts:

`varying float samplecone`
: Same as for the [trace shadeop], page 97.

`uniform float samples`
: Same as for the [trace shadeop], page 97.

`float occlusion` (*point* `Pt`; *vector* `R`; [*float* `samples`;] ...) *

Computes the amount of occlusion, using ray tracing, as seen from *Pt* in direction *R* and solid angle 2*PI (hemisphere). Returns 1.0 if all rays hit some geometry (totally occluded) and 0.0 if there are no hits (totally un-occluded).

- *Pt* and *R* must lie in '`current`' space and *R* must be of unit length.
- The optional *samples* parameter specifies the number of rays to trace to compute occlusion; if absent or set to 0, the value is taken from `Attribute "irradiance" "nsamples"` (see Section 5.2 [attributes], page 44).
- This shadeop accepts many optional token/value pairs. These are explained in Table 6.11.
- More on ambient occlusion in Section 7.4.1 [Ambient Occlusion], page 130 and Section 7.7 [Point-Based Occlusion and Color Bleeding], page 145.

EXAMPLE

```
/* Returns the amount of occlusion using default number of
   samples */
float hemi_occ = occlusion( P, Ng );

/* Returns the amount of occlusion for the hemisphere surrounding P,
   uses a rough approximation with 8 samples */
hemi_occ = occlusion( P, Ng, 8 );

/* Same as above, but only consider objects closer than 10 units and
   in a solid angle of Pi/4 */
hemi_occ = occlusion( P, Ng, 8, "maxdist", 10, "coneangle", PI/4 );

/* Same as above, but only consider light coming from a hemisphere
   oriented toward (0,1,0) */
uniform vector sky = vector (0, 1, 0);
hemi_occ =
    occlusion( P, Ng, 8, "maxdist", 10, "coneangle", PI/2, "axis", sky );
```

`color indirectdiffuse` (*point* `Pt`; *vector* `R`; [*float* `samples`;] ...) *

Computes diffuse illumination due to diffuse-to-diffuse indirect light transport by sampling a hemisphere around a point *Pt* and direction *R*. Use this shadeop to render *color bleeding* effects.

- *Pt* and *R* must lie in '`current`' space and *R* must be of unit length.
- This function makes it possible to lookup into an HDR image when sampled rays do not hit any geometry; the map is specified using the `"environmentmap"` parameter as shown in Table 6.11.
- Computing the occlusion while calling this function is also possible (through the `occlusion` parameter), with the following restriction: `indirectdiffuse()` only sees geometry tagged as visible to reflections, as opposed to `occlusion()` which sees geometry visible to shadows. For more informations about visibility attributes refer to Section 5.2 [attributes], page 44.

Note: This shadeop will automatically perform "final gathering" if there is a photon map attached to geometry. *Using photon maps will speed up global illumination compu-*

tations tremendously using this shadeop, more about photons maps and final gathering in Section 7.4.2 [Photon Mapping], page 131 and Section 7.4.3 [Final Gathering], page 133.

`color indirectdiffuse` (*string* ***envmap****; vector* ***dir***) *

Returns the irradiance coming from an environment and a given direction (*dir* must be of unit length). The environment can be either an environment map or a light probe. Environment maps have to be generated by `tdlmake` with the `-envcube` or `-envlatl` parameter. All other textures are interpreted as light probes. Probe images generated by software such as `HDRShop` and stored in other formats (such as `Radiance` files with an `.hdr` extension) must be converted by `tdlmake` to a normal TIFF texture. It is perfectly correct (and recommended) to provide high dynamic range images to this shadeop. Refer to Section 7.4.4 [Image Based Lighting], page 133 for more information.

Name	Type	Description
"coneangle"	uniform float	Controls the solid angle considered. Default covers the entire hemisphere. Default is $PI/2$.
"axis"	uniform vector	If specified, and different from `vector 0`, indicates the direction of the light casting hemisphere. Rays that are not directed toward this axis are not considered. This is useful for specifying skylights.
"samplebase"	uniform float	Scales the amount of jittering of the start position of rays. The default is to jitter over the area of one micropolygon. Default is 1.0 .
"maxdist"	uniform float	Only consider intersections closer than this distance. Default is $1e38$.
"environmentmap"	uniform string	Specifies an environment map to probe when a ray doesn't hit any geometry.
"environmentspace"	uniform string	Specifies the coordinate system to use when accessing the provided environment map. Default is "world".
"distribution"	uniform string or vector[]	Specifies what distribution to use when tracing rays. '`uniform`' and '`cosine`' distributions are recognized. One can also specify an environment map file that describes the distribution to use (an HDRI of the environment is most appropriate for this). Finally, an array of vectors can be given to specify an arbitrary, user generated distribution. Default is "cosine". Not supported in point-based occlusion.
"environmentcolor"	output varying color	If specified, it is set to the color resulting from environment map lookups on unnoccluded samples. If no environment map is provided, this variable is set to black.
"environmentdir"	output varying vector	If specified, it is set to the average un-occluded direction, which is the average of all sampled directions that do not hit any geometry. Note that this vector is defined in '`current`' space, so it is necessary to transform it to '`world`' space if an `environment()` lookup is intended.
"occlusion"	output varying float	[indirectdiffuse() only] If specified, it is set to the fraction of the sampled cone which was occluded by geometry.
"adaptive"	uniform float	Enables or disables adaptive sampling. Default is 1 (enable). Doesn't affect point-based algorithm.
"minsamples"	uniform float	Specifies the minimum number of samples for adaptive sampling. Defaults to *samples* if it is less than 64 and *samples*/4 otherwise.
"falloffmode"	"uniform float"	Specifies the falloff curve to use. 0 is exponential (default) and 1 is polynomial.
"falloff"	"uniform float"	This shapes the falloff curve. In the exponential case the curve is $exp(-falloff * hitdist)$ and in the polynomial case it is $pow(1-hitdist/maxdist, falloff)$[5]. The default value is 0 for no falloff.

Table 6.11: `occlusion()` and `indirectdiffuse()` optional parameters.

[5] So setting *falloff* to 1 gives a linear falloff curve.

Additionally, for point-cloud based occlusion and color bleeding, the following parameters are recognized:

Name	Type	Description
"pointbased"	uniform float	This has to be set to 1 if point-based occlusion and color bleeding are to be used. Default is 0.
"filename"	uniform string	Specifies the name of a point cloud file to be used to compute the occlusion and color bleeding.
"filenames"	uniform string[]	Like "filename" but allows several point cloud files to be used together.
"hitsides"	uniform string	Specifies which side(s) of the point cloud's samples will produce occlusion. Can take values of "front", "back" or "both". Default is "front".
"maxsolidangle"	uniform float	This is a quality vs speed control when a point cloud is used. Default is 0.1 .
"clamp"	uniform float	If set to 1, attempts to reduce the excessive occlusion caused by the point-based algorithm, at the cost of speed. Default is 0.
"sortbleeding"	uniform float	If set to 1 and "clamp" is also set to 1, this forces the color bleeding computations to take the ordering of surfaces into account. It is recommanded to set this parameter to 1. Default is 0.
"coordsystem"	uniform string	The coordinate system where the point cloud data was stored. Default is "world".
"areachannel"	uniform string	Specifies the name of the channel to be used as area of the points. Defaults is "_area".
"radiositychannel"	uniform string	Specifies the name of the channel to be used as radiosity of the points. Default is "_radiosity" or the first color channel if the one specified is not found. The channel may be either a single float or a color.
"opacitychannel"	uniform string	Specifies the name of the channel to be used as opacity of the points. Default is "_opacity". The channel may be a single float or a color.
"distance"	output varying float	If specified, it is set to the average distance of surfaces which contribute to the effect.
"shadingrate"	uniform float	Controls the shading rate for this specific shadeop and overrides the corresponding irradiance attribute (see Section 5.2.3 [Global Illumination Attributes], page 47).
"maxerror"	uniform float	Controls the maximum error for this specific shadeop and overrides the corresponding irradiance attribute (see Section 5.2.3 [Global Illumination Attributes], page 47).

Table 6.12: Parameters controlling point-based occlusion and color bleeding.

`float gather` (*string* `category`; *point* `P`; *vector* `dir`; *float* `angle`; *float* `samples`, ...)
`statement` [**else** `statement`]

`gather` is considered as a language construct and is detailed in [The Gather Construct], page 77.

`color subsurface` (*point* `Pt; ...`)

Returns subsurface lighting at the given point *Pt*. Subsurface light transport is automatically computed in a separate pass. More about subsurface scattering in Section 7.4.5 [Subsurface Scattering], page 136.

Name	Description
"uniform string groupname"	Allows lookups of arbitrary subsurface groups in the scene.
"uniform string filename"	Specifies the name of the point-cloud file to use. This tells the renderer to operate in point-cloud mode. Refer to Section 7.4.5.2 [Two Pass Subsurface Scattering], page 138.
"uniform string filenames[]"	Like "filename" but allows several point-cloud files to be used together.
"uniform string coordsystem"	Specifies the coordinate system where the points in the point-cloud are to be interpreted. The default is "world" and this has no effect if "filename" is not used.
"color scattering"	Specifies the reduced scattering coefficient when operating in point-cloud mode.
"color absorption"	Specifies the absorption coefficient when operating in point-cloud mode.
"color diffusemeanfreepath"	Specifies the diffuse mean free path when operating in point-cloud mode.
"color albedo"	Specifies albedo (reflectance) when operating in point-cloud mode.
"float ior"	Specifies the index of refraction when operating in point-cloud mode.
"float unitlength"[1]	Specifies the object scale when operating in point-cloud mode.
"uniform float smooth"	May enable smoother results when the point density is too low for the amount of desired scattering. 0 disables the smoothing and 1 enables it. Values smaller or greater than 1 may also be used to control the amount of smoothing.
"uniform float followtopology"[2]	The subsurface approximation assumes that the surface topology is that of a flat surface. Very different geometry can give incorrect results. Setting this parameter to 1 will enable an attempt to correct for this
"uniform string areachannel"	Specifies which channel of the point cloud is used as the area of samples. Default is "_area".
"uniform string radiancechannel"	Specifies which channel of the point cloud is used as the radiance of samples. Default is "_radiance_t" or the first color channel if the one specified is not found.
"color irradiance"	Specifies the irradiance entering the surface at this point. This is an optional parameter that is used by the renderer to compute fast approximations in certain contexts. The renderer will issue warning `R5063` when this parameter is needed but not provided (usually when combining subsurface scattering and global illumination). A good approximation to this value is the diffuse illumination at the surface point.

Table 6.13: `subsurface()` optional parameters.

[1] "scale" is also.
[2] "normalize" is also accepted

`float photonmap` (*string* `mapname`; *point* `P`; *vector* `N`; ...)

This shadeop performs a lookup in the photonmap specified by *mapname* at the surface location described by (P, N). This shadeop accepts the following additional parameters:

`"float estimator"`
: An integer specifying the number of photons to use for the lookup. More photons will give smoother results. The default is 50.

`"string lookuptype"`
: Can take one of the following values:

  ‘`irradiance`’
  : Returns the irradiance at the specified location.

  ‘`radiance`’
  : Returns the radiance at the specified location. Note that 3DELIGHT stores a coarse estimation of the radiance which is not meant for direct visualization. It is mainly useful for the `indirectdiffuse()` shadeop when performing final gathering as explained in Section 7.4.3 [Final Gathering], page 133.

`"float mindepth"`
: When performing *irradiance* lookups, specifies the minimum number of bounces for a photon to be considered in irradiance computation. For example, setting a *mindepth* of ‘`1`’ will avoid photons that come directly from the light sources (meaning that the call will return only *indirect light* contribution).

EXAMPLE

```
/* Perform an irradiance lookup using 100 photons */
color res = photonmap( "gloabal.map", P, Nf, "estimator", 100 );
```

Refer to Section 7.4.2 [Photon Mapping], page 131 for more details about photon maps.

`float caustic` (*point* `P`; *vector* `N`)

This shadeop performs a lookup in the caustic photon map that belongs to the surface being shaded at the surface location described by (P, N). This shadeop can be written using the `photonmap()` shadeop:

```
color caustic( point P, normal N )
{
    uniform float estimator = 50;
    uniform string causticmap = "";

    attribute( "photon:causticmap", causticmap );
    attribute( "photon:estimator", estimator );

    color c = 0;

    if( causticmap!="" )
    {
        c = photonmap(
                causticmap, P, N,
                "lookuptype", "irradiance",
                "estimator", estimator );
    }

    return c;
}
```

Table 6.14: `caustic()` shadeop implementation using the `photonmap()` shadeop.

6.4.6 Texture Mapping

`type texture` (*string* ***texturename***[*float* ***channel***]; ...)
`type texture` (*string* ***texturename***[*float* ***channel***]; *float* ***s***, ***t***; ...)
`type texture` (*string* ***texturename***[*float* ***channel***]; *float* ***s1***, ***t1***, ***s2***, ***t2***, ***s3***, ***t3***, ***s4***, ***t4***; ...)

Returns a filtered texture value, at the specified texture coordinates. `type` can be either a `float` or a `color`. If no texture coordinates are provided, s and t are used. An optional *channel* can be specified to select a starting channel in the texture. This can be useful when a texture contains more than three channels. Use `tdlmake` to prepare textures for improved performance and memory usage (see Section 3.3 [Using the texture optimizer], page 15). `texture()` accepts a list of optional parameters as summarized in Table 6.16. EXAMPLE

```
/* Sharper result (width<1) */
color c = texture( "grid.tdl", s, t, "width", 0.8 );

/* Returns the green component */
float green = texture( "grid.tdl"[1] );

/* Returns the alpha channel, or 1.0 (opaque) if no
   alpha channel is present */
float alpha = texture( "grid.tdl"[3], "fill", 1.0 );

/* Returns an _unfiltered_ color from the texture */
color unfiltered = texture( "grid.tdl", s, t, s, t, s, t, s, t );
```

Note that this shadeop will return proper linear-space colors if the texture has a gamma specification. Refer to Section 7.5 [Texture Mapping in Linear Space], page 140 for details.
The texture() shadeop also includes support for MARI's UDIM tile mapping. Simply provide a file name with the string `UDIM` in it and that will be replaced by the 4 digit tile number before looking for the texture.

`type ptexture ( string texturename; uniform float channel; float faceindex; ... )`
`type ptexture ( string texturename; uniform float channel; float faceindex; float s, t; ... )`
`type ptexture ( string texturename; uniform float channel; float faceindex; float s1, t1, s2, t2, s3, t3, s4, t4; ... )`

Same as `texture` but acts on "ptextures". `ptexture()` accepts a list of optional parameters as summerized in Table 6.15.

Name	Type	Default	Description
"blur"	`float`	0	Refer to Table 6.16
"width"	`float`	1.0	Refer to Table 6.16
"lerp"	`uniform float`	0	If set to 1, lookups will be interpolated between the two closest mipmaps. This usually achieves higher quality.
"fill"	`uniform color`	0	Refer to Table 6.16
"filter"	`uniform string`	`"gaussian"`	Specifies the reconstruction filter to use when accessing the ptexture map. Supported filters are: 'gaussian', 'point', 'bilinear', 'bspline', 'catmull-rom', 'box' and 'mitchell'. The default is 'mitchell'.
"expand"	`uniform float`	0	Refer to Table 6.16

Table 6.15: `ptexture()` optional parameters.

`type environment ( string texturename[channel]; vector V; ... )`
`type environment ( string texturename[channel]; vector V1, V2, V3, V4; ... )`

Returns a filtered texture value from an environment map, for a specified direction. As in `texture()`, an optional *channel* can be specified to select a starting channel when performing texture lookups. Use `tdlmake` to prepare cubic and long-lat envmaps. If an unprepared TIFF is given to `environment()`, it is considered as a lat-long environment map. `environement()` recognizes the same list of optional parameters as `texture()`. If the given file name is "ray-trace", environment uses ray tracing instead of texture lookups, which of course is more expensive. Only geometry tagged as visible to "trace" rays is considered (`Attribute "visibility" "trace" 1`). Optional parameters accepted by this shadeop are listed in Table 6.16. When using ray tracing, this `environment()` also accepts the same optional parameters as `trace` (see [trace shadeop], page 97). EXAMPLE

```
/* Do an env lookup */
vector Nf = faceforward( N, I );
color c = environment( "env.tdl", vtransform("world", Nf) );

/* Only fetch the alpha channel, if no alpha present, returns 1 */
float red_comp = environment( "env.tdl"[3], Nf, "fill", 1 );
```

Note that this shadeop will return proper linear-space colors if the texture has a gamma specification. Refer to Section 7.5 [Texture Mapping in Linear Space], page 140 for details.

Name	Type	Default	Description
"blur"	`varying float`	0	Specifies an additional length to be added to texture lookup region in both s and t, expressed in units of texture coordinates (range = [0..1]). A value of 1.0 would request that the entire texture be blurred in the result. In ray tracing, specifies the half angle of the blur cone, in radians.
"sblur"	`varying float`	0.0	Specifies "blur" in s only.
"tblur"	`varying float`	0.0	Specifies "blur" in t only.
"width"	`uniform float`	1.0	Multiplies the width of the filtered area in both s and t.
"swidth"	`uniform float`	1.0	Specifies "width" in s only.
"twidth"	`uniform float`	1.0	Specifies "width" in t only.
"samples"	`uniform float`	16/4	Specifies the number of samples to use for `shadow()` and `environment()` lookups. This influences the antialias quality. A value of 16 is recommended for shadow maps and 4 is recommended for both DSM and `environment()` lookups. `texture()` only uses this value with the '`box`' filter. In ray tracing mode, specifies the number of rays to trace.
"fill"	`uniform color`	0	If a channel is not present in the texture, use this value.
"filter"	`uniform string`	`"gaussian"`	Specifies the reconstruction filter to use when accessing the texture map. Supported filters are: '`gaussian`', '`triangle`' and '`box`'.
"bias"	`uniform float`	0.225	Used to prevent self-shadowing in `shadow()`. If set to 0, the global bias is used, as specified by `Option "shadow" "bias"` (see Section 5.1 [options], page 29);
"usedmipmap"	`output varying float`	–	`texture()` only. Returns the mipmap level used for the lookup.
"expand"	`uniform float`	0	If this is set to 1, single channel textures have their values duplicated to all channels when doing a color lookup. 0 leaves the default behavior of setting the missing channels to the value provided by "fill".
"lerp"	`uniform float`	0	If set to 1, lookups will be interpolated between the two closest mipmaps (doesn't apply to shadow map lookups). This usually achieves higher quality.

Table 6.16: `texture()` `shadow()` and `environment()` optional parameters.

`type shadow` (*string* `shadowmap`[*float* `channel`]; *point Pt; ...*) *
`type shadow` (*string* `shadowmap`[*float* `channel`]; *point* `Pt1`, `Pt2`, `Pt3`, `Pt4`; *...*) *

Computes occlusion at a point in space using a shadow map or a deep shadow map. Shadow lookups are automatically anti-aliased. When using deep shadow maps, colored shadows and

motion blur are correctly computed. If the file name passed to `shadow()` is `"raytrace"` then ray tracing is used to compute shadows. Note that if ray tracing is used, only objects tagged as visible to shadows are considered (using `Attribute "visibility" "transmission"`). Optional parameters to `shadow()` are described in Table 6.16. Additional information for both shadow maps and deep shadow maps are found in Section 7.2 [Shadows], page 122.

`void bake` (*string* `bakefile`; *float* `s`, `t`; *type* `value`)

This shadeop writes the given *value* (of type float, point or color) to a file named *bakefile*. This file can then be converted to a texture map using `tdlmake` (see Section 3.3 [Using the texture optimizer], page 15) or a call to `RiMakeTexture`. We recommend using the `.bake` extension for files containing such "baked" data.

By defaul, data is saved in a human-readable ASCII format for easy inspection but saving in binary format is also possible. To do so one can concatenate the `"&binary"` string to the file name, as shown in Listing 6.7.

```
surface bake2d(
    string bakefile = "";
    float binary = 0; )
{
      Ci = noise(s*32, t*16);

      if( binary == 1 )
          bake( concat(bakefile,"&binary"), s, t, Ci );
      else
          bake( bakefile, s, t, Ci );
}
```

Listing 6.7: Baking into a binary file.

Note that baking in 2D space is only appropriate when the underlying surface has a well-behaved 2D parametrisation (which is often the case in most models that had undergone some UV unwrapping). More about baking in Section 7.6 [Baking], page 141.

`float bake3d` (*string* `bakefile`; *string* `channels`; *point* `P`, *normal* `N`, ...)

This shadeops records a set of values to the file *bakefile* for the given position and normal. Values to record are supplied as named parameter pairs. For example:

```
float occlusion = occlusion( P, N, samples, "maxdist", 3 );
bake3d( texturename, "", P, N, "surface_color", Ci, "occlusion", occlusion );
```

Note that the *channels* parameter is ignored in the current implementation. The files created by the `bake3d()` are unfiltered point clouds which can be used directly or converted to brick maps as explained in Section 7.6.3 [Brick Maps], page 143; in both cases data is directly accessible through `texture3d()` (see [texture3d shadeop], page 111). `bake3d()` returns 1 if the write is successful and 0 otherwise. It takes various optional parameters listed in Table 6.17.

Name	Type	Default	Description
"coordsystem"	`string`	"world"	Specifies the coordinate system into which the values are written.
"radius"	`varying float`	based on grid	Specifies the radius of the disk (or sphere if N is 0) covered by each baked sample.
"radiusscale"	`float`	1.0	Allows scaling of the radius value without overriding it. In point cloud files, the size of a disk affects the importance of its orientation for lookups with `texture3d`. Larger disks are less likely to be picked if the normals do not match. They are more strongly oriented.
"interpolate"	`float`	0.0	When set to 1.0, saves the centers of the micropolygons instead of its corners. This is primarily useful for point-based occlusion and color bleeding (see Section 7.7.3 [Point-Based Imaging Pipeline], page 145).

Table 6.17: `bake3d()` optional parameters.

Additionally, there is two channel names that are reserved for use with point-based occlusion and color bleeding (see Section 7.7 [Point-Based Occlusion and Color Bleeding], page 145):

'`float _area`'
: Specifies the area of the given point. It is not necessary to specify this channel if it is not different from the actual micro-polygon area.

'`color _radiosity`'
: Specifies the radiosity at the given point. This is used for point-based color bleeding.

`float texture3d` (*string* ***bakefile****; point* ***P****, normal* ***N****, ...*)
: This shadeop is used to perform lookups in files created by `bake3d()` (see [bake3d shadeop], page 110). Its syntax is indeed very similar to `bake3d()`:

```
float occlusion;
texture3d( texturename, P, N, "surface_color", Ci, "occlusion", occlusion );
```

`texture3d()` returns 1 if the lookup was successful and 0 otherwise. It takes various optional parameters listed in Table 6.18.

Name	Type	Default	Description
"coordsystem"	`string`	"world"	Specifies the coordinate system from which the values are written.
"filterradius"	`float`	–	Specifies the radius of the lookup filter. If none is given, the renderer computes one that is faithful to the shaded element derivatives.
"filterscale"	`float`	1	A multiplier on the radius of the lookup filter. It is not necessary to specify '`filterradius`' to use this parameter.
"maxdepth"	`float`	–	Sets a maximum depth for lookups. If not specified, maximum depth is used. Works for both point clouds and brick maps.

Table 6.18: `texture3d()` optional parameters.

6.4.7 String Manipulation

`string concat` (*string* ***str1***, ..., ***strn***)
Concatenates one or more strings into one string.

`string format` (*string* ***pattern***; ***val1***, ..., ***valn***) *
Similar to the C `sprintf` function. *pattern* is a string containing conversion characters. Recognized conversion characters are:

`%f` Formats a float using the style [-]ddd.ddd. Number of fractional digits depends on the precision used (see example).

`%e` Formats a float using the style [-]d.ddde dd (that is, exponential notation). This is the recommended conversion for floats when precision matters.

`%g` The floating point is converted to style `%f` or `%e`. If a precision specifier is used, `%f` is applied. Otherwise, the format which uses the least characters to represent the number is used.

`%d` Equivalent to `%.0f`, useful to format integers.

`%p` Formats a point-like type (`point`, `vector`, `normal`) using the style `[%f %f %f]`.

`%c` Same as `%p`, but for colors.

`%m` Formats a matrix using the style `[%f %f %f %f, %f %f %f %f, %f %f %f %f, %f %f %f %f]` if a precision specifier is used. Otherwise, each element is formatted to full precision as with `%g`.

`%s` Formats a string.

`%h` Formats a shader handle. Light shaders and co-shaders will use the handle of their declaration. The null shader object is output as `<null>`. Surface, Displacement, etc become `<surface>`, `<displacement>`, etc.

Note that all conversion characters recognize the precision specifier.

EXAMPLE

```
/* Formats a float using exponential notation */
string expo = format( "%e", sqrt(27) );

/* Formats a float, with 5 decimals in the fractional part */
point p = sqrt(5);
string precision5 = format( "p = %.5p", p );

/* Aligns text */
string aligned = format( "%20s", "align me please" );
```

`void printf` (*string* `pattern`; `val1`, ..., `valn`) *
Same as `format()` but prints the formatted string to `stdout` instead of returning a string.

`float match` (*string* `pattern`, `subject`)
Does a string pattern match on *subject*. Returns 1 if pattern exists anywhere within subject, 0 otherwise. The *pattern* can be any standard `regex`[3] expression.

6.4.8 Message Passing and Information

`float textureinfo` (*string texturename, fieldname; output uniform type* `variable`)
Returns information about a particular texture, environment or shadow map. *fieldname* specifies the name of the information as listed in Table 6.19. If *fieldname* is known, and *variable* is of the correct type, `textureinfo()` returns 1.0. Otherwise, 0.0 is returned.

EXAMPLE

```
/* mapres[0] gets map resolution in x,
   mapres[1] gets map resolution in y */

uniform float mapres[2];
textureinfo( "grid.tdl", "resolution", mapres );

/* Get current to camera matrix used to create the shadow map */

uniform matrix Nl;
if( textureinfo( "main-spot.tdl", "viewingmatrix", Nl )!= 1.0 )
{
    Nl = 1;
}
```

Additionally, a special call to `textureinfo()` is provided to test texture existence:

```
color C;
if( textureinfo(texturename, "exists", 0) )
    C = texture(texturename);
else
    C = errorcolor;
```

The existence check also checks for validity so if the file exists on disk but is not a valid texture, this function will return 0. An error message will only be output if the file is present but invalid.

[3] See '`man regex`' for details (`man` pages available on UNIX-like platforms only).

Field	Type	Description
"resolution"	`uniform float[2]`	Returns texture map resolution.
"type"	`uniform string`	Returns type of texture map. Can be one of the following: "texture", "shadow", "environment", "ptexture", "pointcloud", "brickmap", "photonmap".
"channels"	`uniform float`	Returns the total number of channels in the map.
"viewingmatrix"	`uniform matrix`	Returns a matrix representing the transform from "current" space to "camera" space in which the map was created.
"projectionmatrix"	`uniform matrix`	Returns a matrix representing the transform from "current" space to map's raster space[4].
"bitsperchannel"	`uniform float`	Returns the number of bits used to represent each channel of the texture. This is typically 8, 16 or 32.
"pixelaspectratio"	`uniform float`	Returns the aspect ratio of the pixels in the texture.
"min"	`uniform float[]`	Returns the minimum value of each channel over the entire texture.
"max"	`uniform float[]`	Returns the maximum value of each channel over the entire texture.
"fileformat"	`uniform string`	Returns the file format for textures. Can be one of: "tiff", "openexr" or "deepshadow".

Table 6.19: `textureinfo()` field names.

`float atmosphere` (*string* `paramname`*; output type* `variable`)
`float displacement` (*string* `paramname`*; output type* `variable`)
`float incident` (*string* `paramname`*; output type* `variable`)
`float opposite` (*string* `paramname`*; output type* `variable`)
`float lightsource` (*string* `paramname`*; output type* `variable`)
`float surface` (*string* `paramname`*; output type* `variable`)

Functions to access a parameter in one of the shaders attached to the geometric primitive being shaded. The operation succeeds if the shader exists, the parameter is present and the type is compatible, in which case 1.0 is returned. Otherwise, 0.0 is returned and *variable* is unchanged. Note that assigning a `varying` shader parameter to a `uniform` *variable* fails. Also, `lightsource()` is only available inside an `illuminance()` block and refers to the light source being examined.

`float attribute` (*string* `dataname`*; output type* `variable`)

Returns the value of the data that is part of the primitive's attribute state. The operation succeeds if *dataname* is known and the type is correct, in which case 1.0 is returned. Otherwise, 0.0 is returned and *variable* is unchanged. The supported data names are listed in Table 6.20. User defined attributes are also accessible as in:

```
uniform float self_illuminated = 0;
attribute( "user:self_illuminated", self_illuminated );
if( self_illuminated == 1 ) {
 .. do some magic here ...
}
```

[4] x varies from -1 to 1 (left to right) and y from 1 to -1 (top to bottom)

It is possible to query the attributes of a light source from a surface shader. To do so, use the `attribute()` shadeop inside an illuminance loop and prefix the attribute name with `"light:"`.

```
illuminance( P )
{
    string name;
    if( 1 == attribute( "light:identifier:name", name ) )
        printf( "light name: %s\n", name );
}
```

The list of recognized uniform attributes is are listed in Table 6.20.

"Ri:ShadingRate"	"Ri:Sides"
"Ri:Color"	"Ri:Opacity"
"Ri:Matte"	"Ri:DetailRange"
"Ri:TextureCoordinates"	"Ri:Orientation"
"GeometricApproximation:motionfactor"	"displacementbound:sphere"
"displacementbound:coordinatesystem"	"shaderdisplacementbound:sphere"
"shaderdisplacementbound:coordinatesystem"	"geometry:backfacing"
"geometry:frontfacing"	"geometry:geometricnormal"
"grouping:membership"	"identifier:name"
"trace:bias"	"trace:displacements"
"trace:maxspeculardepth"	"trace:maxdiffusedepth"
"photon:shadingmodel"	"photon:causticmap"
"photon:globalmap"	"photon:estimator"
"sides:doubleshaded"	"visibility:camera"
"visibility:diffuse"	"visibility:specular"
"visibility:trace"	"visibility:transmission"
"visibility:photon"	"dice:hair"
"dice:rasterorient"	"cull:backfacing"
"cull:hidden"	"derivatives:smooth"
"derivatives:centered"	"subsurface:groupname"
"subsurface:scale"	"subsurface:refractionindex"
"subsurface:shadingrate"	"subsurface:absorption"
"subsurface:scattering"	"subsurface:meanfreepath"
"subsurface:reflectance"	"attribute:light:samples"
"attribute:light:samplingstrategy"	"attribute:light:emitphotons"
"irradiance:shadingrate",	"irradiance:nsamples"
"hider:composite"	"user:*attributename*"

Table 6.20: `attribute()` field names.

The type of each attribute is directly related to its declaration when declating the corresponding attribute in the Ri stream (with the exception of integer attributes becoming floating point values in the shader since there is no integer type in SL).

Two entries in the table require slightly more information:

`shaderdisplacementbound:sphere`
: This returns the displacement bound as specified by the `__displacementbound_sphere` shader parameter. If both displacement and surface shader has such parameters, their will simply be added together *meaning that they should be declared in the same space*. Refer to [shader declared displacementbound], page 50.

`shaderdisplacementbound:coordinatesystem`
: Will return the __displacementbound_coordinatesystem. Surface shader parameter will be returned if displacement shader has no such parameter. Again, it is imperative to have the same space in both shaders.

`float option` (*string* ***dataname****; output type* ***variable***)

Returns the data that is part of the renderer's global option state. The operation succeeds if *dataname* is known and the type is correct, in which case 1.0 is returned. Otherwise, 0.0 is returned and *variable* is unchanged. The supported data names are listed in Table 6.21. User defined options (as specified by `RiOption`, see [User Options], page 41) are also accessible:

EXAMPLE

```
uniform float shadow_pass = 0;
option( "user:shadow_pass", shadow_pass );
if( shadow_pass == 1 ) {
 ...
}
```

Name	Type	Description
"Format"	`uniform float[3]`	Returns [x resolution, y resolution, aspect ratio].
"FrameAspectRatio"	`uniform float`	Frame aspect ratio.
"Hider"	`uniform string`	Current hider.
"CropWindow"	`uniform float[4]`	Crop window coordinates as specified by `RiCropWindow`.
"DepthOfField"	`uniform float[3]`	Returns [fstop, focal length, focal distance]; as specified by `RiDepthOfField`.
"Shutter"	`uniform float[2]`	Returns [shutter open, shutter close]; as specified by `RiShutter`.
"Clipping"	`uniform float[2]`	Returns [near, far]; as specified by `RiClipping`.
"Ri:FrameBegin"	`uniform float`	Returns the frame number given to `RiFrameBegin`.
"limits:gridsize"	`uniform float`	Option "limits" "gridsize"
"limits:texturememory"	`uniform float`	Option "limits" "texturememory"
"trace:maxdepth"	`uniform float`	Option "trace" "maxdepth"
"trace:specularthreshold"	`uniform float`	Option "trace" "specularthreshold"
"searchpath:shader"	`uniform string`	shader searchpath
"searchpath:texture"	`uniform string`	texture searchpath
"searchpath:display"	`uniform string`	display searchpath
"searchpath:resource"	`uniform string`	resource searchpath
"searchpath:archive"	`uniform string`	archive searchpath
"searchpath:procedural"	`uniform string`	procedural searchpath
"shutter:offset"	`uniform float`	Option "shutter" "offset"
"user:*useroption*"	`any type`	A user defined option.

Table 6.21: `option()` field names.

Name	Type	Description
"renderer"	`uniform string`	Returns "3Delight".
"version"	`uniform float[4]`	Returns [Major, Minor, release, 0] (e.g. [1,0,6,0]).
"versionstring"	`uniform string`	Version expressed as a string, (e.g., "1.0.6.0").

Table 6.22: `rendererinfo()` field names.

`uniform float` *rendererinfo* (*string* `dataname`*; output type* `result`)
Returns information about the renderer. The operation succeeds if *dataname* is known and the type is correct, in which case 1.0 is returned. Otherwise, 0.0 is returned and *result* is unchanged. The supported data names are listed in Table 6.22.

`float rayinfo` (*uniform string* `keyword`*; output type* `result`)
Provides informations about the current ray. Returns 1.0 if *keyword* is known and *result* is of the correct type. Returns 0.0 on failure.

EXAMPLE

```
/* Choose the number of rays to trace depending on current ray depth. */
uniform float ray_depth;
rayinfo( "depth", ray_depth )

samples = max(1, samples / pow(2, ray_depth) );

trace_color = trace( P, Nr, "samples", samples );
```

Name	Type	Description
"speculardepth"	`uniform float`	Returns the specular depth of the current ray.
"diffusedepth"	`uniform float`	Returns the diffuse depth of the current ray.
"shadowdepth"	`uniform float`	Returns 1.0 if the current ray is a transmission ray, 0 otherwise.
"depth"	`uniform float`	Returns the total depth of the current ray: $speculardepth + diffusedepth + shadowdepth$.
"type"	`uniform string`	Returns one of the following: '`camera`', '`light`' (for photons), '`specular`', '`diffuse`', '`subsurface`', or '`transmission`'.
"label"	`uniform string`	Returns the label attached to this ray, if any.

Table 6.23: `rayinfo()` field names.

`float raylevel` ( )
`float isindirectray` ( )
`float isshadowray` ( )
Deprecated functions. Their functionalities can be reproduced using the `rayinfo()` shadeop. See [rayinfo shadeop], page 117.

`float arraylength` (*var*)
Returns the number of elements in the array *var* or -1 if *var* is a scalar.

`float isoutput` (*var*)
Returns 1 if *var* is output to a display driver. This is true if:

1. The variable is declared as `output` in shaders' parameters.
2. The variable is specified in at least one display driver.

This function is useful to optimize shaders that have to calculate many AOVs: it is wasteful to compute an AOV if there is no display driver that uses it.

```
surface shader1( output varying float noisy = 0 )
{
    if( isoutput( noisy ) )
    {
        noisy = noise( P );
    }
}
```

`float isoutput (` *string* `varname )`

Returns 1 if the variable with the given name is output to a display driver. This is the same function as the other form above but by name instead of directly giving the variable to test. This form can also query individual array components:

```
surface shader1( output varying float outputs[3] = {} )
{
    if( isoutput( "outputs[1]" ) )
    {
        outputs[1] = foo();
    }
}
```

`void outputchannel (` *uniform string* `channelname;` *value* `)`

This function allows output variables to be dynamically added to a shader. For example, these two shaders would produce equivalent behavior:

```
surface shader1(
    output varying color myoutput = 0 )
{
    myoutput = Cs;
}

surface shader2();
{
    outputchannel( "myoutput", Cs );
}
```

If the variable *channelname* already exists, it is simply assigned to. If there is some incompatibility in the type of detail of the existing variable then nothing is done. *value* may be of any type except string. Output variables should still be declared whenever possible as it is more efficient.

Note that if `outputchannel()` is called from a light shader or a coshader, the output is added to the main shader by default (usually the surface) so it will be visible to the display system. To override this behavior, prefix the variable name with `"light:"` or `"local:"`.

`void gridmin (` *varying float* `value )`
`void gridmax (` *varying float* `value )`

These special functions take a varying floating point parameter as input and find out the minimum and maximum values over the entire grid (in REYES) or shading point set (ray-tracing). These functions are useful to take decisions for an entire set of shading samples. For example, if it is known that some parameter is less than X for the entire grid then one might skip some particular code branch. Not to be confused with min and max shadeops.

6.4.9 Co-Shader Access

`shader getshader` (*shader* ***shader_name***)
Get the shader with the specified name and active in the current attribute scope.

`shader[] getshaders` (["*category*"*; string category_name*])
Get all co-shaders declared in the current attribute scope, filtering them using an optional category name.

`shader getlight` (*string light_handle*)
Returns the co-shader for the light source with handle *light_handle*.

`shader[] getlights` (["*category*"*; string category_name*])
Returns all active lights in the current attribute scope, filtering them using an optional category name. This method can be used to build custom illuminance loops. For example,

```
vector L;
color Cl, C=0;
shader lights[] = getlights( "category", "diffuse" );
uniform float num_lights = arraylength( lights ), i;
for (i = 0; i < num_lights; i += 1)
{
    lights[i]->light( L, Cl );
    C += Cl * (Nn . normalize(-L));
}
```

Note that the L vector is automatically inverted by `illumiance()` but in this case one has the responsibility to invert it manually. Refer to [The Illuminance Construct], page 74 for a more explanation about the mechanics of `illuminance()` and refer to [Light Categories], page 74 and Section 6.3 [Predefined Shader Parameters], page 89 for more about light categories.

`float getvar` (*shader* ***coshader***, *string* ***name***, [*output* ***type*** ***variable***])
A function to access member variables in a specified co-shaders (see Section 6.1.2.6 [Co-shaders], page 68). Another use of this function is giving it the null shader handle to make it search for the variable directly on the geometric primitive, bypassing all shaders. This is likely to be slower to compute than getting a similar variable from another shader.

6.4.10 Operations on Arrays

`void resize` (*output type* ***A***[], *uniform float* ***length***)
Resizes the array to the specified *length*.

- Increase array length will leave the new elements uninitialized.
- *length* is always uniform.
- If *length* is set to 0, the memory held by the array is released

`void reserve` (*output type* ***A***[], *uniform float* ***length***)
Increase array's capacity if necessary without affecting it's length.

`void push` (*output type* ***A***[], *type* ***value***)
Increase the array length by one and initializes the last element to *value*.

`type pop` (*output type* ***A***[])
Returns the last element of A and decreases array's length by one. The capacity of the array is not decreased.

7 Rendering Guidelines

7.1 Multithreading and Multiprocessing

3Delight can render an image using both multithreading and multiprocessing. Additionally, *3Delight* is able to render an image using any number of machines that are reachable on the network (and potentially using many threads or processes on each machine). This very complete set of functionalties makes *3Delight* usable on any multiprocessor hardware configuration.

This section explains threading and multiprocessing in more detail and gives hints on when to use one or the other.

7.1.1 Multithreading

By default, *3Delight* starts the render using as many *threads* as there are processors. A thread is different from a process in that it runs in the same memory space as the parent process, meaning that using many threads on a single image won't affect memory use significantly (unlike multiprocessing as explained in Section 7.1.2 [Multiprocessing], page 120). One can override the number of threads used by passing the `-p` option to `renderdl`. For example,

```
% renderdl -p 3 frame1.rib
```

will use three threads to render an image. More about the `-p` option is explained in Section 3.1 [Using the RIB renderer], page 7.

The number of threads can also be specified inside the RIB using a dedicated `RiOption`. For example,

```
Option "render" "nthreads" 2
```

will use two threads to render the image.

3Delight will assign a small region of the image to each thread started. Typically, each region will have a size of 2x2 buckets.

7.1.2 Multiprocessing

A process is different from a thread in that it runs in its own memory space; this means that using multiprocessing will generally use more memory resources than a multithreaded render. The advantage of using processes instead of threads is that there is very little *synchronization* overhead between processes and this might lead to faster renders compared to multithreaded renders.

Multiple processes can be launched using `-P` command line option. For example,

```
renderdl -P 2 image.rib
```

will start a render using two processes.

The way `renderdl` splits the image in a multiprocess render is different from the multithreading case: the image is split in large areas (or *tiles*), each tile is assigned to one process. *3Delight* doesn't know in advance how to efficiently cut the image into such tiles, for example, a process might be assigned to an empty region thereby leaving other processes to do more work and thereby wasting time. This is why *3Delight* uses a load balancing strategy that records rendering times for each process and tries to better tile the image for the next render. This means that two consecutive renders of a frame will use different tiling strategies and the second render should use less time than the first one (refer to Section 7.1.4 [Performance Notes], page 122 for more details).

The tiling strategy can be manually forced using the `-tiling` command line option (Section 3.1 [Using the RIB renderer], page 7). Three different tiling strategies are supported: vertical, horizontal and mixed. For example, to use the mixed tiling mode, one would write:

```
% renderdl -P 4 -tiling m frame1.rib
```

Thes best splitting strategy depends on the image being rendered: choose the tiling so that the complexity is well distributed among the processes. Most outdoor scenes should be rendered using vertical tiling so that every process gets its share of the "sky" region where complexity is usually very low. If the complexity is uniformly distributed inside the scene, tiling orientation won't make much of a difference.

7.1.3 Network Rendering

The `-P` and `-p` options start processes on the local machine only; if rendering on many hosts is desired then `renderdl` should be provided with a list of usable machines through the `-hosts` option[1]. But before using the remote rendering feature, it is essential that the following points are checked:

Properly configured `rsh` or `ssh`
: By default, *3Delight* uses `rsh` to start a rendering process on remote machines, this means that the machine that starts the render should have the required permissions on all used remote hosts[2]. If `rsh` is judged insecure, one could use SSH by providing `renderdl` with the `-ssh` option. But this requires passwordless SSH authentication and requires more setup than the `rsh` method.

3Delight must be executable from each rendering machine
: This should be the case if the package is installed on a NFS drive. Note that it is important that the `.cshrc` or `.tcshrc` file contain all the environment variables necessary to run *3Delight*. An easy way to verify that everything is setup properly on a remote machine is to invoke `rsh` (or `ssh`) manually:

```
% rsh hostname renderdl somefile.rib
```

The RIB file should be accessible from each remote machine.
: This means that the RIB should reside on a shared drive, such as a NFS drive.

Once everything is setup properly, one can perform network rendering using the `-hosts` option. For example, to compute an image using four machines (the current machine, origin1,leto and harkonnen), one would issue the following command (this will also open a framebuffer window):

```
% renderdl -hosts origin1,leto,harkonnen,localhost -id frame1.rib
```

Note that by default, every machine will use as many threads as possible; so if '`origin1`' has two CPUs, both will be used to render the part of the image that has been assigned to the machine. Additionally, a machine could be specified more than once, this will launch that many processes on the specific machine. For convenience, `renderdl` also accepts a text file describing the list of hosts:

```
% renderdl -hosts workers.txt,leto -id frame1.rib
```

This will read all the machines listed in the file `workers.txt` (each host name on a single line) and use those for rendering as well as the '`leto`' machine.

It is not required that the current machine is part of the hosts set: one could launch a single processor render on a different machine (eg. if the current machine is too loaded):

```
% renderdl -hosts worker -id frame1.rib
```

Note that different machine architectures can be used to compute a single image: it is possible to use both MacOS X and Linux machines to cooperate on a single image.

---

[1] The `-p` and the `-hosts` options should not be used together.

[2] This usually means modifying the `/etc/hosts.equiv` file, see `rsh` documentation for details.

7.1.4 Performance and Implementation Notes

It is recommanded to use multithreading when rendering medium or complex scenes on a local machine. The number of threads should be set to the number of *physical* processors available. Some "hyperthreaded" machines show two processors even if there is only one physical CPU installed on the motherboard; using two processes on those architectures won't necessarily improve performance (and certainly won't double it). It is sometimes desirable to set one more thread than available when the rendering generates a lot of IO (such as texture or network access): having one more thread will give the operating system the oportunity to switch to an alternate rendering thread while another one is waiting for an IO operation to complete.

Using multiprocessing is recommanded for scenes where shading time is not important and most processing is spent in *visibility* computations. This usually happens in scenes with very simple shaders. Normal production scenes force the renderer to spend much more time in the shading stage (especially when using ray tracing) and multithreading is the right choice for these.

When rendering over a network, *3Delight* uses the TCP/IP protocol to gather data from the remote hosts. This could put a heavy load on the network and on the machine that starts the render. When using a large pool of machines, one should make sure that the master machine is fast enough to handle all connections. This is particularly important when rendering deep shadow maps over the network.

There is a fundamental design difference between multiprocessing and multithreading: multiprocessing is implemented in the `renderdl` executable while multithreading is implemented in the *3Delight* library. While this doesn't make any difference for users that render RIBs using the `renderdl` command, users that link with the *3Delight* library can only use multithreading.

7.1.5 Licensing Behavior

In the multithreading case, the behavior is as follows:

- If enough licenses are available, the render will proceed normally.
- If there are fewer licenses available than requested threads, *3Delight* will start the render with as many threads as possible and will dynamically add threads as more licenses are made available.
- If there are no licenses available, *3Delight* will wait for at least one license and will automatically start the render. Threads will be added dynamically as more licenses are available.

In the multiprocessing case, each launched process waits for a license to be available.

7.2 Shadows

3Delight has an extensive support of shadow rendering methods. It is up to you to choose the algorithm that suits you best.

7.2.1 Raytraced Shadows

Tracing shadow rays is an easy way for rendering shadows that requires very little setup. This technique has the advantage of being convenient and efficient in the case of complex scenes. Ray traced shadows are further described in Section 7.3.2 [Ray Traced Shadows], page 126.

7.2.2 Standard Shadow Maps

These are normal shadow maps that are widely used in the industry. Generating such shadow maps implies placing the camera at the position of the light source and rendering the scene from that view point (`zfile` or `shadowmap` display driver has to be selected, see Section 8.6 [dspyzfile], page 170

and Section 8.7 [dspy_shadowmap], page 171 for information about file formats used and options). The shadow map is then used from inside a light source shader to cast shadows on objects. Shadow maps have a number of advantages:

- *They are fast to generate.* Indeed, shadow maps can be generated much faster than a normal "color" image, mainly because only depth information is needed. When rendering a shadow map, one could remove all surface and light shaders, even displacement shaders can be removed if they do not affect the geometry too much. Also, filtering (using `PixelFilter` command) can be lowered (even to 1x1) and `ShadingRate` increased (up to 10 or more).
- *They can be reused in more than one render.* If the scene is static and only the camera moves, a generated shadow map can be used for all subsequent renders. This often happens in the lighting stage of a production pipeline.
- *They can be used to generate low cost penumbra.* Specifying an appropriate blur to the `shadow()` call, one can simulate penumbra effects (see [shadow shadeop], page 109).
- *They provide fairly good results when used carefully.* Many of the recent CG productions use normal shadow maps and obtain excellent results.

Now, the drawbacks:

- *Self shadowing.* This is the most common problem encountered when using shadow maps. It appears as dark artifacts in areas that should appear completely lit. This problem can be resolved by using an appropriate shadow bias, either via an option (see [Rendering Options], page 34) or through the '`bias`' parameter when calling `shadow()` (see [shadow shadeop], page 109).
- *Nearly impossible to generate high quality area shadows.* Even if tweaking with shadow blur can give a nice penumbra effect, it is impossible to generate a true area shadow.
- *Expensive to generate really sharp shadows.* This is because high resolution shadow maps are needed which often leads to longer render times and memory/disk usage.
- *No motion blur in shadows.* Moving geometry still casts shadows. It is wise to remove motion blur when rendering shadow maps.
- *No colored shadows.* Translucent surfaces cast opaque shadows.
- *Only objects that are in the shadow map cast shadows.* That is why shadow maps work so well with spot lights: they light only a limited field of view. Point lights are more tricky to handle (need six shadow maps) and distant lights are difficult to setup with shadow maps.

When creating shadow maps, make sure that shadow casting objects are framed correctly (and tightly) in the camera view that is used to render the shadow map. If objects are too far (small in shadow map view), precision problems may arise and high resolution shadow maps are needed. If objects are too close and parts of them are clipped, shadows might be missed. *3Delight* supports the '`midpoint`' (Section 5.1 [options], page 29) algorithm for normal shadow maps. This should help to get rid of shadow bias problems in most cases.

7.2.3 Deep Shadow Maps

Deep shadow maps retain many of the features found in normal shadow maps and provide some more goodies:

- *Translucent shadows.* Translucent surfaces cast correct shadows.
- *Shadows in participating media.* It is possible to render volumetric shadows using DSMs (the display driver has a flag to account for this feature, see Section 8.8 [dspy_dsm], page 171).
- *Supports mipmapping.* DSMs are mipmapped, which means that they exhibit much less aliasing artifacts than normal shadow maps.
- *Needs lower resolution.* Instead of generating large shadow maps to boost quality, one can generate a smaller DSM by using higher `PixelSample` values when creating the DSM.

- *Copes well with fine geometric details.* Fine details such as hair and particles can be handled without increasing the shadow maps resolution too much.
- *Render time shadow lookups are faster.* Since DSMs are prefiltered, render time lookups are faster (in general) than with normal shadow maps since the signal reconstruction step is more efficient. In addition, DSMs are more texture cache friendly than shadow maps.
- *Easily included in a rendering pipeline that already uses shadow maps.* No modifications are necessary in shaders when changing shadow map format (normal or deep), the `shadow()` shadeop works with both formats.

Before throwing DSMs into the production pipeline, consider the following facts:

- *More expensive to compute.* Generating good DSMs is generally slower than shadow maps. This is often the case when many `PixelSample`s are used during the generation step.
- *Needs more disk space.* DSMs can grow quite large, mainly for two reasons:
  1. DSMs are mipmapped by default.
  2. More data is stored per texel.

Using the "midpoint" option to produce deep shadow maps can lead to unpredictable results!

For deep shadow map creation please refer to Section 8.8 [dspy_dsm], page 171.

7.2.4 Aggregated Shadow Maps

Aggregated shadow maps are a collection of *standard* shadow maps concatenated into one single file. A good example of the usefullness of this feature is to generate shadow maps for point light sources, in which case six shadow maps are needed per light source (one for each cordinal direction). In this case, having six shadow maps to deal with creates at least two "problems":

1. Pipeline wise, managing six shadow maps is more complicated than managing just one.
2. Shaders (most commonly light source shaders) need to accept six shadow map string parameters, which is tedious.

The aggregation operation is performed by passing the `aggregate` parameter to the shadowmap display driver, as explained in Section 8.7 [dspy_shadowmap], page 171. For example, to aggregate a shadow map to the already existing `shadow_point.shd` file, one would issue:

```
Display "shadow_point.shd" "shadowmap" "z" "integer aggregate" [1]
```

There is no limit on the number of shadow maps that can be aggregated together and the resulting shadow map can be queried from inside shaders using a single `shadow()` call. The resulting operation of this single call is the accumulation of the opacities of all shadow maps present in the agregated file. Parameters passed to `shadow()`, such as '`blur`' and '`samples`', will be used on all the shadow maps in the aggregated file.

Note: Aggregated shadow maps do not need to be rendered from the same point of view, as in the point light example.

7.3 Ray Tracing

3Delight implements a number of ray tracing capabilities to render effects such as reflections, refractions, ray probes, transmission probes and soft shadows. Many of the standard SL shadeops have been extended to fully take advantage of *3Delight*'s ray tracing features. In a summary:

`trace()` A function to perform multiple-importance sampling operation in order to sample BRDFs and environments. This is the function to use to render physically-plausible materials. The calling syntax is described in [trace shadeop], page 97.

`transmission()`
A function to render ray traced shadows. The calling syntax is described in [transmission shadeop], page 98.

`occlusion()`
: A shadeop to render *ambient occlusion*[3] effects. This will intersect with objects visible to *transmission* rays. The calling syntax is described in [occlusion shadeop], page 99.

`indirectdiffuse()`
: A shadeop to render *color bleeding* effects. This will intersect objects visible to *reflection* rays. Can be used to implement the *final gathering* algorithm when combined with photon maps, as explained in Section 7.4.3 [Final Gathering], page 133. The calling syntax is described in [indirectdiffuse shadeop], page 99. In practice, this function is deprecated in favor of the much more advanced `trace()`.

`gather()`
: A general construct to sample the scene using ray tracing and to sample HDR images. This will interesect objects visible to *reflection* rays. The calling syntax is described in [gather shadeop], page 103.

7.3.1 Reflections and Refractions

Using the `trace()` shadeop is the simplest way to compute reflections and refractions although existing shaders need to be modified. Shaders that use `environment()` can perform ray tracing without any code modifications by providing "raytrace" as the environment map name. A shader that can be enhanced in this way is the standard `shinymetal`. Note that geometry does not appear in ray tracing unless it is visible to the right type of rays. For environment(), this is accomplished using:

```
Attribute "visibility" "int specular" [1]
```

When computing reflection or refractions on certain kind of surfaces (such as glass), it is important to take surface orientation into account to compute the correct index of refraction (*eta*): rays exiting a surface should invert the index of refraction as shown in the following code snippet:

```
normal Nn = normalize( N );
vector In = normalize( I );
normal Nf = faceforward( Nn, In );
float eta = 1.3; /* glass */

if( In.Nn < 0 )
{
    /* The ray is exiting from the surface, inverse the relative index of
       refraction */
    eta = 1.0 / eta;
}

/* Compute refracted and reflected ray */
float kr, kt;
vector reflDir, refrDir;
fresnel(In, Nf, eta, kr, kt, reflDir, refrDir);
...
```

Listing 7.1: Computing the correct index of refraction for surfaces such as glass.

It is often useful to account for an environment map when reflected or refracted rays do not hit anything and this can be achieved using the optional '`transmission`' parameter available to `trace()`, as shown in Listing 7.2.

[3] Also known as *geometric exposure*.

```
color SampleEnvironment(
    point P;  vector R;  float blur, raysamples;  string envname;)
{
   /* First trace a ray and then composite the environment on top if
      the transmission != 0 */
    color transmission;
    color ray = trace( P, R,
        "samples", raysamples,
        "samplecone", blur,
        "transmission", transmission );

    if( transmission == 0 || envname == "" )
        return ray;

    /* NOTE: should apply correct blur here */
    color env = environment(envname, P, vtransform("world",R) );
    return ray + transmission * env;
}
```

Listing 7.2: How to combine ray tracing and environment mapping.

7.3.2 Ray Traced Shadows

Before rendering ray traced shadows, one should make sure that shadow casting objects (those that can obstruct light rays) are visible to '`transmission`' rays. This is accomplished using:

```
Attribute "visibility" "int transmission" [1]
```
[4]

Once the visibility flags are correctly set, one can compute shadows in many ways:

1. By using `transmission()`, usually from a light source shader.
2. By calling `shadow()` and passing '`raytrace`' as a shadow map name.
3. By using the automatic shadows functionality.

Using "automatic" shadows is the simplest way to render shadows since it requires the addition of only one statement in front of a light source declaration (provided that objects are visible to '`transmission`' rays):

```
Attribute "light" "shadows" "on"
LightSource "spotlight" 1 ...
```

Using `shadow()` with "raytrace" as a shadow map name is another simple way to trace shadows without any shader programming[5]; and existing shaders that use shadow maps are directly usable. In this case, user has more control on shadows' shape using the "blur", "bias" and "maxdist" options passed to `shadow()` (see [shadow shadeop], page 109). Here is a RIB snippet that uses the standard `spotlight` shader to compute "soft shadows":

```
LightSource "spotlight" 1 "string shadowmap" ["raytrace"] "float blur" 0.05
...
```

The specialized `transmission()` can be used to test the visibility between two given points; it can also be used to render soft shadows. Here is an example light source that uses this shadeop.

[4] Refer to [Primitives Visibility and Ray Tracing Attributes], page 45 for more about visibility.

[5] Note that `shadow()` can ray trace shadows only from *light source* shaders.

```
light shadowspot (
    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 beamdistrib = 2;
    float samples = 1;
    float blur = 0;
    float bias = -1;
)
{
    uniform vector A = (to - from) / length(to - from);
    float cos_coneangle = cos( coneangle );
    float cos_delta  = cos( coneangle - conedeltaangle );

    illuminate( from, A, coneangle )
    {
        float cosangle = L.A / length(L);

        color atten = pow(cosangle, beamdistrib) / (L.L);
        atten *= smoothstep(cos_coneangle, cos_delta, cosangle);
        atten *=
            transmission( Ps, from,
              "samples", samples,
              "samplecone", 0,
              "bias", bias );

        Cl = atten * intensity * lightcolor;
    }
}
```

Listing 7.3: Ray traced shadows using `transmission()`

7.3.3 Ray Labels

Ray labels are identifiers attached to traced rays. Labels are specified using the '`label`' optional parameters that is supported by all ray tracing shadeops. A shader can then obtain the label of the current ray (the one that caused the shader evaluation) using the `rayinfo()` shadeop (see [rayinfo shadeop], page 117).

```
    ...

    uniform string ray_label = "";
    rayinfo( "label", ray_label );

    if( ray_label == "approx" )
    {
      ... do something ...
    }
    else
    {
      .. do something more accurate
    }
    ...
```

7.3.4 Trace Group Membership

All ray tracing capable shadeops accept a "subset" parameter which tells *3Delight* that traced rays intersect only with geometry which is part of the given group(s). For example:

```
color env = trace( P, Nf, "subset", "nearby" );
```

Intersects only objects that belong to the "nearby" group. The following table lists the meaning of the various strings which can be specified as subset:

`""`	Objects which don't belong to any group.
`"group1"`	Objects which belong to *group1*.
`"group1,group2"`	Objects which belong to *group1* or *group2*.
`"+group1"`	Objects which belong to *group1* or which don't belong to any group.
`"-group1"`	Objects which don't belong to *group1*.

Additionally, it is possible to specify a default subset in case ray tracing functions do not specify one. This is achieved using `Attribute "grouping" "tracesubset"`.

7.3.5 Ray Tracing Displacements

By default, *3Delight* renders ray traced displaced geometry using bump mapping, meaning that only surface normals are affected by displacement shaders, not geometry. It is possible to correctly render such geometry in ray tracing by using:

```
Attribute "trace" "displacements" [1]
```
[6]

It is important to note that there is an additional cost when using true geometric displacements in the context of ray tracing, for the following reasons:

1. *3Delight* needs more memory to hold displaced geometry (even though it uses a caching mechanism to reduce memory allocations).
2. Displacement bounds expand primitives' bounding boxes, this has a bad effect on ray tracer's space partitioner, meaning less efficient ray / primitive intersections. It is clear that badly set displacement bounds (or extreme displacement) can make ray tracing of true displacements *very* costly.
3. For many primitives, geometrical intersections are less efficient than the analytical algorithm *3Delight* uses for non-displaced ray tracing.

[6] Refer to Section 5.2.5 [displacement], page 49 for more information.

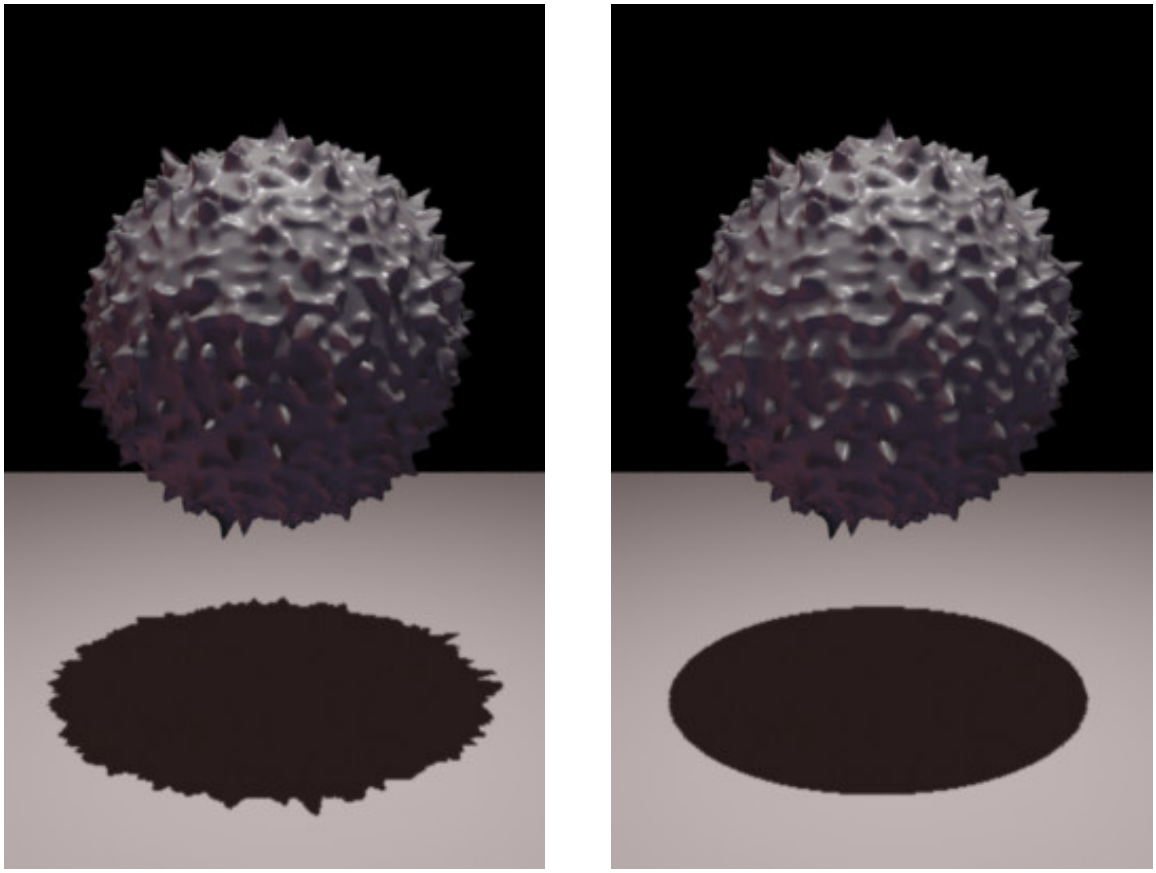


Figure 7.1: Raytraced displacements become necessary when silhouette details and self shadowing are important.

7.3.6 Oversampling and Derivatives

All ray tracing functions accept a parameter to specify the number of rays to trace. Tracing many rays is the only way to perform spacial and temporal (motion-blur) anti-aliasing when using ray tracing[7]. Follows a table that shows, for every ray tracing shadeop, the default number of rays traced and how to modify the default behavior.

`trace()`
`shadow()`
`environment()`
`transmisson()`

Traces only one ray if '`samples`' is not specified. The following examples traces 16 rays for better anti-aliasing:

```
color ref = trace( P, R, "samples", 16 );
```

`occlusion()`
`indirectdiffuse()`

Traces the number of rays specified using the following attribute (see Section 5.2.3 [Global Illumination Attributes], page 47):

[7] Increasing `RiPixelSamples` only affects the anti-aliasing in the primary REYES render.

Attribute "irradiance" "integer nsamples" [n]

The default is 64 samples. If the optional `float samples` parameter is specified to the shadeop, it overrides the '`nsamples`' attributes (see [occlusion shadeop], page 99).

`gather()` Traces the number of rays specified to the function.

Automatic shadows

Traces the number of rays as specified by the following attribute:

Attribute "light" "integer samples" [n]

The default value is 1.

3Delight automatically computes ray differentials on every traced ray, this means that first order derivatives are available in shaders called in a ray tracing context. This ensures that derivative functions such as `Du()`, `Dv()` and `Deriv()` (see Section 6.4.1 [math shadeops], page 89) behave properly even after many reflections and refractions.

7.3.7 The Ray Tracing Hider

3DELIGHT uses an efficient REYES algorithm to render objects visible from the eye and uses a ray tracer for effects such as reflections and refractions. By selecting the '`raytrace`' hider however, 3DELIGHT can use its ray tracer to render primary visibility. The following example sets the ray tracer as the primary renderer and switches off *jittering*:

```
Hider "raytrace"
  "int jitter" [0]
```

The ray tracer is slower than the default REYES renderer but it could prove useful in some cases:

- No "eye split" problems. Primitives that are difficult to render using the standard algorithm can be rendered in the ray tracer without problems.
- Motion blurred mirror-like objects will have good reflections. This is a classical problem in REYES renderers where shading is done only once for a mirror-like objects, meaning that motion-blurring the mirror will blur the reflection, which is not the correct behavior.
- Can produce nicer refractions and reflections since there is one shading point per *subsample* and not per micro-polygon.

The ray tracing hider produces the same quality images as the REYES hider and supports all the features you expect from a high end renderer: motion blur, depth of field and correct derivatives in shaders.

7.4 Global Illumination

7.4.1 Ambient Occlusion

Ambient occlusion (also known as "geometric exposure") is trivially computed using the `occlusion()` shadeop (see [occlusion shadeop], page 99). Objects contributing to occlusion have to be tagged as visible to transmission rays using `Attribute "visibility" "transmission" "opaque"`[8]. `occlusion()` works by tracing a specified number of rays into the scene or can be provided with a point cloud to compute occlusion using a point-based approximation. The rest of this chapter deals with the ray tracing version of `occlusion()`, the point-based approach is detailed in Section 7.7 [Point-Based Occlusion and Color Bleeding], page 145.

[8] '`Os`' and '`shader`' are also valid but slower and add very little.

The number of samples specified as a parameter to `occlusion()` is a quality *vs.* speed knob: less samples give fast renders and more noise, more samples yield to longer render times but smoother appearance. 16 to 32 samples is a recommended setting for low quality renders, 128 and more samples are needed for high quality results. It is important to turn `RiShadingInterpolation` to '`smooth`'; this increases `occlusion()` quality, especially on the edges where two different surfaces meet and form a concave feature.

Occlusion can also be returned using the `indirectdiffuse()` shadeop (see [indirectdiffuse shadeop], page 99). This is useful when both indirect illumination and ambient occlusion are needed. Note that using `indirectdiffuse()` to compute occlusion can be much more costly than using `occlusion()` since it often triggers execution of shader code.

Here is a simple light source shader illustrating the use of `occlusion()`:

```
light occ_light(
  float samples = 64;
  string __category = "occlusion";
  float __nonspecular = 1; )
{
  normal shading_normal = normalize( faceforward(Ns, I) );

  illuminate( Ps + shading_normal )
  {
    Cl = 1 - occlusion( Ps, shading_normal, samples );
  }
}
```

Listing 7.4: Ambient occlusion using a lightsource shader

7.4.2 Photon Mapping

3DELIGHT supports photon mapping for both caustics and indirect lighting. Photon maps can either be generated in a different pass and stored on disk for re-use, or they can be automatically generated by 3DELIGHT in the main rendering pass and stored in memory only. The next two sections describe the two approaches in detail. Examples illustrating both methods can be found in `$DELIGHT/examples/photons`.

7.4.2.1 Photon Mapping Concepts

Photon mapping is an elegant technique to render effects such as caustics and multi-bounce global illumination efficiently. The algorithm proceeds by tracing a specified number of "photons" from light sources that are then "bounced" from surface to surface and stored on diffuse surfaces. The stored photons can then be used to compute light irradiance during the main (beauty) pass. The main applications for photon maps are:

Render caustics
: To do this, one needs to explicitly call a shadeop to compute irradiance – either by calling `photonmap()` (see [photonmap shadeop], page 105) or `caustic()` (see [caustic shadeop], page 106).

Render color bleeding using "final gathering"
: This is trivially achieved by rendering a photonmap and calling `indirectdiffuse()` which will automatically query photon maps for secondary bounces. More about final gathering in Section 7.4.3 [Final Gathering], page 133 and [indirectdiffuse shadeop], page 99.

Render fast global illumination previews
: This is done by directly using the photon maps to compute irradiance at each sample, without calling `indirectdiffuse()`. In some cases, this could lead to acceptable results even for final renders.

7.4.2.2 Single Pass Photon Mapping

Here is the list of steps required to enable single pass photon map generation:

1. Set `Option "photon" "emit"` to the number of photons to trace.
   This indicates to the renderer how many photons are to be traced into the scene. This specifies the total number of photons to trace and not a number for each light: 3DELIGHT will analyze each light such as to trace more photons from brighter lights. The number of photons depends on scene complexity and light placement and can vary between thousands and millions of photons.
2. Mark primitives visible to photon rays using `Attribute "visibility" "photon" [1]`.
   Primitive that are not marked as visible to photons will not be included in the photon tracing process.
3. Name a global photon map and a caustic photon map using `Attribute "photon" "globalmap"` and `Attribute "photon" "causticmap"`. These photon map names can then be used to access the photon maps as explained in [photonmap shadeop], page 105. It is common to set, for the entire scene, only one global map for final gathering and one for caustics but it is perfectly legal to specify one per object or per group of objects.
4. Select material type for surfaces using `Attribute "photon" "shadingmodel"`.
   This will indicate how each surface react to photons. Valid shading models are: '`matte`', '`chrome`', '`water`', '`glass`' and '`transparent`'.
5. Enable photon map generation for desired lights using `Attribute "light" "emitphotons" ["on"]`. Normally, only spot lights, point lights and directional lights are turned on for photon generation. Light sources that do global illumination work (such as calling `occlusion()` or `indirectdiffuse()`) *should not cast photons*.

In this single pass approach, the total number of bounces -both diffuse and specular- are set using `RiOption`. For example, to set the total number of bounces to 5, one would write:

```
Option "photon" "maxdiffusedepth" [5] "maxspeculardepth" [5]
```

Using the single appoach to photon mapping has some interesting advantages:

- Only uses core memory: no data is written to disk if the user doesn't want it.
- Simple to use with already established pipelines.

The main drawback being that generated photon maps are not re-usable.

7.4.2.3 Two Pass Photon Mapping

If re-using photon maps is a prerequisite, then the two pass approach can be used to generate the photon maps and save them into files to be used in subsequent renders. This is done much like for the one pass approach but instead of using the `Option "photon" "emit"` command a special '`photon`' hider is used:

```
Hider "photon" "int emit" [100000]
```

When using this hider, the renderer will not render the scene but instead store all the computed photon maps into files specified by `Attribute "photon" "globalmap"` and `Attribute "photon" "causticmap"`.Those photon maps can then be read by specifying the same attributes in a normal render (using the '`hidden`' hider). An example of this approch is found in the `$DELIGHT/examples/photons` directory.

The total number of bounces a photon can travel can be specified on a per-prmitive basis, for example:

```
Attribute "trace" "maxdiffusedepth" [10] "maxspeculardepth" [5]
```

7.4.3 Final Gathering

Final gathering is a technique to render global illumination effects (color bleeding) efficiently. In summary, for global illumination rendering using n bounces, the algorithm first proceeds by emitting photons from light sources for the $n-1$ bounces. Then, the last bounce is simulated by tracing a specified number of rays from surfaces; at each ray-surface intersection the photon map is used to compute the radiance. The main benefit here is that instead of ray tracing the entire n bounces, we only ray trace one bounce and the rest is available through photon maps.
This functionality is readily available using the `indirectdiffuse()` shadeop (see [indirectdiffuse shadeop], page 99) which traces a specified number of rays in the scene and collects incoming light. If `indirectdiffuse()` encounters photon maps on surfaces it will automatically use them and avoid ray tracing, effectively achieving the so called final gathering algorithm.

7.4.4 Image Based Lighting

Integrating synthetic elements into real footage became an important aspect of modern computer imagery. To make the integration of such elements as realistic as possible, it is of prime importance to respect the lighting characteristics of the real environment when rendering the synthetic elements. Lighting characteristics include both the color of the elements as well as the shadows cast by those elements into the real world.

Both imperatives can be achieved by placing lights in the virtual environment that mimic the placement of lights in the real environment. This is usually a tedious task that is also error prone. A more elegant technique relies on photographs of real environments to compute lighting on synthetic elements. Those photographs are usually high dynamic range images taken on set.

Preparing HDR Images

High dynamic ranges images come mostly in two format: light probes or two fisheye images. `tdlmake` can read both formats and convert them into a cubic environment map usable from shaders. The following example converts a "twofish" environment in EXR format into a cubic environment map usable in 3DELIGHT:

```
% tdlmake -twofish north.exr south.exr env.tdl
```

Once the environment is converted into a cubic environment map it can easily be accessed from the shading language.

Using HDR Images

Most people are familiar with the `occlusion()` shadeop that simulates shadows as cast by a hemispherical light[9]. The shadeop proceeds by casting a specified number of rays in the environment and probing for intersections with surrounding geometry. Rays are usually cast uniformly on the hemisphere defined by surface's point and normal. While very useful, this technique is not convincing for non uniformly lit environments. To overcome this problem, *3Delight* offers a special `gather()` construct to extract illumination from an environment map: passing the environment map as a special *category* to `gather()` will perform an importance sampling over the environment and provide useful information in the body of the construct:

[9] Although one can specify a '`distribution`' parameter to `occlusion()` to specify *arbitrary ray distributions*, as explained in [occlusion shadeop], page 99.

‘`environment:direction`’
: The current direction from the environment map. More precisely, the current direction describes the center of a *luminous cluster* in the environment map. This direction can be used to trace transmission rays to obtain hard shadows.

‘`environment:solidangle`’
: The solid angle for the current direction. This can be considered as the extent, in space, of the luminous cluster currently considered.

‘`ray:direction`’
: A random direction inside the luminous cluster described by ‘`environment:direction`’ and ‘`environment:solidangle`’. This can be used to trace transmission rays for to obtain blurry shadows.

‘`environment:color`’
: The color “coming” from the current environment direction. This is the average color of the luminous cluster.

This is better illustrated in Listing 7.5.

```
...
/* Assuming that:
   'envmap' contains the name of the environment map.
   'kocc' containts the shadowing strength.
   'envspace' is the space in which the environment is. */
string gather_cat = concat("environment:", envmap);
color envcolor = 0;
vector envdir = 0, raydir = 0;
float solidangle = 0;

gather( gather_cat, 0, 0, 0, samples,
    "environment:color", envcolor, "ray:direction", raydir,
    "environment:direction", envdir, "environment:solidangle", solidangle )
{
    raydir = vtransform( envspace, "current", raydir );
    envdir = vtransform( envspace, "current", envdir );

    color trs = transmission( Ps, Ps + raydir * maxdist, "bias", bias );
    trs = 1 - kocc * (1 - trs);
    Cl += envcolor * trs;
}
...
```

Listing 7.5: Using `gather()` to extract lighting information from environment maps.

Thanks to the special `gather()` construct shown in Listing 7.5, it becomes easy to compute *image based lighting*: transmission rays are traced into the scene to compute the shadowing and the color given by ‘`environment:color`’ is used directly to compute the environment color. It is recommanded to combined the environment light with surface's BRDF to obtain more realistic results.

If one wants the *diffuse* environment lighting with no occlusion, it is always possible to call the `indirectdiffuse()` shadeop (see [image based indirectdiffuse], page 100). This shadeop provides a simple way to do diffuse image based lighting without texture blurring and a guaranteed smoothly varying irradiance function. `indirectdiffuse()` efficiently integrates the irradiance coming from distant objects, as described by the provided environment map. No special operations are needed on environment maps: the same environment used for standard reflection mapping can be directly

passed as a parameter. One could simply call `indirectdiffuse` with an environment map and the surface normal to get the diffuse lighting as seen by the [P, N] surface element:

```
color diffuse_component = indirectdiffuse( envmap, N );
```

IMPORTANT: Prior *3Delight* 7.0, image based lighting was accomplished using `occlusion()` and `indirectdiffuse()`. This combination still works as usual but one must be aware that:

- `occlusion()` uses a less than perfect algorithm to detect the luminous clusters in an environment map.
- Combining `occlusion()` and `indirectdiffuse()` does not yield to "correct" results since the diffuse lighting doesn't take the occlusion into account.

So we strongly recommand using the `gather()` construct instead of the previous method.

The `envlight2` light shader provided with *3Delight* combines both the image based occlusion and indirect diffuse lighting and has been used to produce figure Figure 7.2.

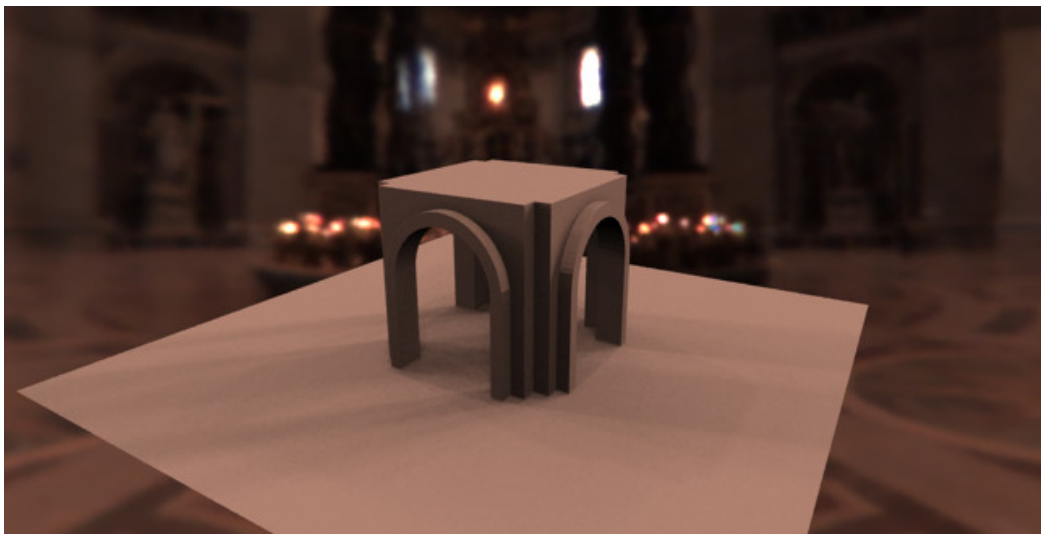


Figure 7.2: An image rendered using an HDRI map and the `envlight2` shader. There is no "light sources" in the scene: the shadows are cast from bright regions in the environment.

7.4.5 Subsurface Scattering

7.4.5.1 One Pass Subsurface Scattering

3Delight features an integrated subsurface light transport algorithm, meaning that subsurface scattering simulations do not need additional passes.

Here is an image created with the example included in the *3Delight* distribution (it can be found in `$DELIGHT/examples/subsurface/`):

Figure 7.3: Teapot rendered with subsurface scattering enabled.

Running a subsurface simulation is a three step process:

1. Surfaces defining a closed object are marked as belonging to the same subsurface group.
2. Subsurface light transport parameters are specified using `Attribute "subsurface"` on one of the surfaces that is part of the closed object.
3. Object's surface shader is modified to correctly account for subsurface lighting.

Each operation is further explained below. A functional RIB file can also be found in `$DELIGHT/examples/subsurface`.

Defining a Closed Object

This is done using `Attribute "visibility" "subsurface" "`*`groupname`*`"`; an example RIB snippet:

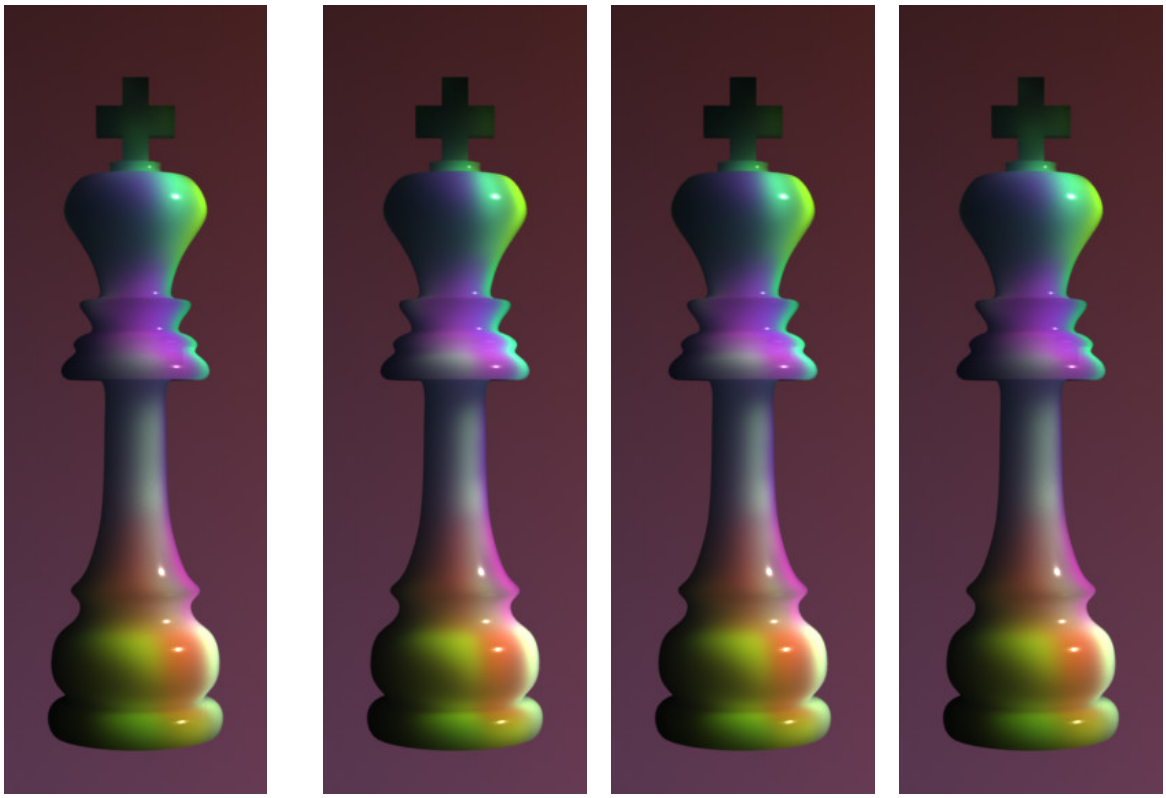
```
AttributeBegin
  Attribute "visibility" "subsurface" "marble"

  ... Object's subsurface light transport parameters go here ...
  ... Object's geometry goes here ...
AttributeEnd
```

Parts of geometry belonging to the same closed object can be specified *anywhere* in the RIB, as long as they all belong to the same subsurface group.

Specifying Subsurface Scattering Parameters

Subsurface light transport parameters can be specified in two different ways (see [Subsurface Scattering Attributes], page 48): the first one is by specifying the reduced scattering coefficient, reduced absorption coefficient and surface's relative index of refraction; the second way is by specifying material's mean free path, diffuse reflectance and index of refraction. All the provided measures should be given in millimeters.

The '`shadingrate`' attribute has the conventional interpretation and determines how finely irradiance is sampled in the precomputation phase. The default value is 1.0 but larger values should be used for highly scattering materials, greatly improving precomputation speed and memory use as shown in Figure 7.4.



'`shadingrate`' = 1.0	= 4.0	= 10.0	= 50.0
Timing = 48*secs*	= 40*secs*	= 33*secs*	= 23*secs*
Memory = 43*Megs*	= 20*Megs*	= 18*Megs*	= 16*Megs*

Figure 7.4: Highly scattering materials can benefit from higher shading rates. The chess piece is translucent enough to use very high shading rates without affecting quality significantly.

Since geometry is rarely specified in millimeters, a scaling factor, which is applied to the given parameters, can also be provided to *3Delight*. For example, specifying a scale of '`0.1`' means that geometry data is specified in centimeters (objects are ten times bigger). Continuing with the previous RIB snippet:

```
AttributeBegin
  Attribute "visibility" "subsurface" "marble"

  # marble light scattering parameters
  Attribute "subsurface"
    "color scattering" [2.19 2.62 3.00]
    "color absorption" [0.0021 0.0041 0.0071]
    "refractionindex" 1.5
    "scale" 0.1
    "shadingrate" 1.0

  ... Object's geometry goes here ...
AttributeEnd
```

Surface properties can be obtained from specialized literature. A reference paper that contains many parameters (such as skin) directly usable in *3Delight*:

- Henrik Wann Jensen, Stephen R. Marschner, Marc Levoy and Pat Hanrahan. *A Practical Model for Subsurface Light Transport*. ACM Transactions on Graphics, pp 511-518, August 2001 (Proceedings of SIGGRAPH 2001).

Modifying surface Shaders

3Delight renders subsurface scattering in two passes: it firsts starts a precomputation process on all surfaces belonging to some subsurface group and then starts the normal render. When shaders get executed in the preprocessing stage, `rayinfo("type", ...)` returns "subsurface". Using this information, a shader can perform conditional execution depending on the context. Here is the standard plastic shader changed as to work with subsurface scattering.

```
surface simple( float Ks = .7, Kd = .6, Ka = .1, roughness = .04 )
{
  normal Nf = faceforward( normalize(N), I);
  vector V = normalize(-I);

  uniform string raytype = "unknown";
  rayinfo( "type", raytype );

  if( raytype == "subsurface" )
  {
    /* no specular is included in subsurface lighting ... */
    Ci = Ka*amient() + Kd*diffuse(Nf);
  }
  else
  {
    Ci = subsurface(P) + Ks * specular(Nf, V, roughness);
  }
}
```

Listing 7.6: Simple subsurface shader

7.4.5.2 Two Pass Subsurface Scattering

One disadvantage of using the one-pass method for subsurface scattering is that the pre-computation performed by the renderer is not re-usable from frame to frame. This can be fixed by using a two-pass technique that relies on a point-cloud file to store the intermediate color values.

- In the first pass the frame is rendered normally, as a beauty pass, and the result is saved in the '`_radiosity`' channel of a point-cloud file (if needed, the area of the micro-polygon can be altered by using the '`_area`' float channel). It is important to correctly set the culling attributes so that hidden surfaces are rendering, as explained in Section 7.6 [Baking], page 141.
- In the second pass, the `subsurface()` shadeop (see [subsurface shadeop], page 104) is called by

specifying the point-cloud file along with the subsurface parameters and is normally blended with the direct illumination as in the one-pass method.

The two pass method is better illustrated using a simple example.

```
Projection  "perspective" "fov" [ 35 ]

ArchiveBegin "geo"
  AttributeBegin
    Translate  0 0 3 Scale  0.25 0.25 0.25 Translate  0 -0.5 0
Rotate -120 1 0 0
Geometry "teapot"
 AttributeEnd
ArchiveEnd

LightSource  "spotlight" 1
  "intensity" [ 120 ] "from" [ 5 3.5 2.5 ]
  "uniform string shadowmap" "raytrace"
  "coneangle" [ 2 ] "lightcolor" [ 1 1 1 ]

Attribute  "visibility" "transmission" "Os"

Display  "none" "null" "rgb"
WorldBegin
  Surface  "simple_bake"
    "Ks" [ 0.7 ] "roughness" [ 0.02 ] "string bakefile" ["teapot.ptc"]
  Attribute "cull" "hidden" [0] "backfacing" [0]
  ReadArchive "geo"
WorldEnd

Display  "teapot_using_ptc.tif" "tiff" "rgb"
WorldBegin
  Surface "simple_using_ptc"
    "Ks" [ 0.7 ] "roughness" [ 0.02 ] "string bakefile" ["teapot.ptc"] "float scale" [0.05]
  ReadArchive "geo"
WorldEnd
```

Listing 7.7: Example RIB illustrating two-pass subsurface scattering.

The `simple_bake` and `simple_using_ptc` are listed below.

```
surface simple_bake( float Ks = .7, Kd = .6, Ka = .1, roughness = .04;
  string bakefile = "" )
{
  vector V = normalize(-I);
  Ci = Cs * (Ka*ambient() + Kd*diffuse(normalize(N))) + subsurface(P);
  float A = area(P);
  bake3d( bakefile, "", P, N, "_radiosity", Ci, "_area", A, "interpolate", 1 );
}

surface simple_using_ptc(
  float Ks = .7, Kd = .6, Ka = .1, roughness = .04;
  string bakefile = "";
  color albedo = color(0.830,0.791, 0.753);
  color dmfp = color(8.51, 5.57, 3.95);
  float ior = 1.5, scale = 1.0;  )
{
  normal Nf = faceforward( normalize(N), I);
  vector V = normalize(-I);

  Ci = subsurface(P, Nf,
      "filename", bakefile,
      "diffusemeanfreepath", dmfp, "albedo", albedo,
      "scale", scale, "ior", ior )
    + Ks * specular(Nf, V, roughness);
}
```

Here is a summary of the major differences with the one-pass techniques:

1. The subsurface parameters (albedo, diffuse mean free path) are not set by attribute anymore, they are specified to the `subsurface()` shadeop. The shading rate of the subsurface pass is simply specified using `RiShadingRate()` in the baking pass.
2. There is no more subsurface groups: groups are implicitly declared by the geometry rendered and baked in the point-cloud file.
3. User has the ability to use two different shaders: one for the baking pass and one for the final pass. This is not possible in the one-pass method. In normal situations this isn't desirable though, most of the code line will be shared between the shaders. It is of course necessary not to include the specular component of the illumination model in the pre-pass (this includes reflections and refractions): this is not only inaccurate but also slows down the rendering of the pre-pass.

7.5 Texture Mapping in Linear Space

A good rendering and imagine pipeline correctly "tracks" the color space of all image assets. Indeed, serious problems can arise if color space is not properly considered, from texture acquisition to the actual rendering. One can imagine the following problematic case:

1. An artist paints a texture on his calibrated monitor using a screen gamma of 2.0 .
2. A lighting artist puts that texture on an object with his screen gamma set to 1.7 .
3. Since the texture in step 1 has not been properly *linearized*, the lighting artist will find it difficult to properly match the texture with surrounding environment because it has been created in a different color space.

The problem above can be avoided if all the artists work in linear color space and with well calibrated monitors. Still, it is sometimes difficult to get the original source image in linear color space (for example, an image could be provided by a third party) and a conversion is necessary to bring the image into linear space. Another part of the problem is that many algorithms in computer graphics are only correct in linear space; two such algorithms are texture blurring and texture filtering. A somewhat unexpected implication of this is that performing the mip-mapping

process using the `tdlmake` utility produces wrong mipmaps if the input image is not in linear space[1]. Another unexpected result is that the texture filtering performed by `texture()` and the likes (see [texture shadeop], page 107) on non linear images also results in slightly incorrect images.

To elegantly solve the aforementioned problems, `tdlmake` acknowledges two options (see [tdlmake options], page 15) to specify the color space of the source image: `-rgbagamma` or `-gamma` to specify the gamma and `-colorspace` to specify the name of some particular color space. For example,

```
tdlmake -rgbagamma 1.7 1.8 1.7 1.0 trees.tif trees.tdl
```

Will process the input image as if the gamma was 1.7 for red, 1.8 for green 1.7 for blue and 1.0 for alpha (the `-gamma` similarly specifies one single gamma value for all channels for the input image). The following sets the color space to *sRGB*:

```
tdlmake -colorspace sRGB intput.tif output.tdl
```

The actual implications of specifying a gamma (or color space) to `tdlmake` are the following:

- `tdlmake` will create correct mip-maps by performing a proper linearization prior to filtering. The resulting image will still be stored with its original gamma so it is identical in look to the input.
- `tdlmake` will store the gamma of the image using a special TIFF tag.
- The `texture()` and `environment()` shadeops will properly linearize values for filtering and *will return linear-space colors* so that the lighting is correctly performed in linear space.

7.6 Baking

"Baking" is the pre-computation of a *costly shading* operator into a texture map or a point cloud so that it can be reused in more than one render using efficient texture lookups. Baking is generally desirable for costly ray tracing effects such as ambient occlusion and indirect illumination (see Section 7.4 [Global Illumination], page 130). Before considering the baking approach one should be aware that:

- Baking makes the rendering pipeline more complicated, because of the introduction of an extra step;
- Not all costly operations are "bake-able" because they change abruptly with scene animation (not reusable);
- Baking might require, for the entire render, important amounts of storage;
- Baking requires RIB and shader code modifications and slightly more technical skill.

That being said, baking is widely used in production and could result in important rendering speedups.
Before baking, one must make sure to disable *3Delight*'s hidden surface removal in order to include, in the bake file, informations that are invisible in a normal render. Visibility culling, as well as backface culling (when `RiSides` is set to '`1`'), can be disabled using the following RIB snippet:

```
Attribute "cull" "hidden" [0]
Attribute "cull" "backfacing" [0]
```

These two attributes should be set for all primitives for which baking is desired (see [Primitives Visibility and Ray Tracing Attributes], page 45). Additionally, it is good to disable the view dependant dicing so that micro-polygons on object's silouettes are properly formed:

```
Attribute "dice" "rasterorient" [0]
```

NOTE: Culling and dicing attributes could have a severe impact on performance, they should not be used in beauty (main) renders.

[1] Although the error is usually unnoticeable.

7.6.1 2D Baking

2D baking is the operation of storing some function into a standard *texture map*. This is possible when the underlying model has a suitable parametrisation, which is often the case since one needs a proper parametrisation to use texture mapping. The following example code saves the ambient occlusion into a 2D texture map using the `bake()` shadeop (see [bake shadeop], page 110):

```
float occ = occlusion( P, N );

if( is_baking == 1.0 )
{
    /* Bake occlusion into a file */
    bake( "occ.bake", s, t, occ );
}
```

Listing 7.8: Occlusion baking using the `bake()` shadeop.

The created `.bake` is a collection of 2D points and their associated data and is not suitable for later texture lookups. To convert it into a mip-mapped texture that can be read back using the `texture()` shadeop one could use `tdlmake` (see Section 3.3 [Using the texture optimizer], page 15):

```
tdlmake -bakeres 2048  occ.bake occ.tdl
```

Another way to achieve the same effect is to call `RiMakeTexture` from inside the RIB file (or programmatically from inside a C program):

```
MakeTexture "occ.bake" "occ.tdl" "clamp" "clamp"
    "sinc" 8 8 "int bakeres" 2048
```

An example of the entire process is illustrated in the setup found in the `$DELIGHT/examples/baking` directory.

7.6.2 3D Baking

When no suitable and continuous two dimensional parametrisation exists for some piece of geometry, 2D baking becomes tedious or non-feasible. One parametrisation that is always available is the 3D space in which primitives are defined: this gives us the option to save the data in a 3D point cloud file using the `bake3d()` shadeop (see [bake3d shadeop], page 110):

```
...
bake3d( "occ.ptc", "", P, N, "surface_color", Ci );
...
```

The saved values can be read back using the `texture3d()` call (see [texture3d shadeop], page 111):

```
...
color res =0;
texture3d( "occ.ptc", P, N, "surface_color", res );
...
```

The interface to the `bake3d()` shadeop is richer than the one for `bake()` shadeop as it is possible to specify more than one data channel per 3D point, among other things.

NOTE: Point cloud files created with the `bake3d()` shadeop can be viewed using the `ptcview` utility.

Implementation Details

In contrast with other implementations, *3Delight*'s point cloud files can be loaded partially in memory and efficiently accessed using a caching mechanism, similar to renderer's 2D texture caching system.

This is important since point clouds can become quite large. Additionally, point clouds can be read anytime during their creation: a shader that issues a `bake3d()` call can also call `texture3d()` on the same file. In other words, a point cloud file is always in a readable state.

7.6.3 Brick Maps

Point cloud files are stored in a sparse data structure that is not suited for filtered lookups. This means that accessing point cloud files could result in aliasing. The solution is to convert the point cloud file into a mip-mapped 3D texture, similarly to the 2D textures produced by `tdlmake`. Such 3D textures are commonly called "brick maps" and have nice properties such as filtering and automatic level of detail. There is two ways to convert a point cloud file into such a texture:

a. By using the `ptc2brick` utility (see Section 3.9 [Using ptc2brick], page 26) as in:

```
ptc2brick scene.ptc scene.bkm
```

b. By calling `RiMakeBrickMap` from inside a RIB file (or programmatically from inside a C source file):

```
...
MakeBrickMap [ "scene.ptc" ] "scene.bkm"
...
```

NOTE**:** The first parameter to `RiMakeBrickMap` in the example above is an *array* of strings and not a single string. This means that the command can receive more than one point cloud file at a time. The same applies to `ptc2brick`: more than one point cloud file can be specified as input.

A brickmap file can then be accessed using the `texture3d()` shadeop (see [texture3d shadeop], page 111), exactly as for point cloud files.

7.6.4 Baking Using Lights

One potential downside to baking is the necessary modifications to all contributing shaders. The basic structure of a shader that has been modifying for baking is similar to this:

```
surface foo(
    ... common parameters ...
    ... baking related parameters )
{
    ... main shader's code ...

    ... baking code ...
}
```

The most annoying part of this approach is the addition of the *baking related parameters*. Not only one has to add them to all shaders but also one needs to specify them for each shader instance. One solution to this problem is to specify the baking parameters as *user attributes*:

```
Attribute "user" "string bakefile" "color.ptc"
```

The shader can then access these parameters using the `attribute()` shadeop (see [attribute shadeop], page 114). Another solution is to use a light source to perform the baking. The main advantage of using light sources instead of user attributes is that most 3D packages have an easy way to perform and visualize *light linking*; so one can easily attach a light source to parts of a scene that needs to be baked.

```
light bakelight(
    bool do_bake = 0.0;
    string bake_file = "";
    float bake_color = 0.0;
    string spacename = "world";
    string __category = "bakelight";)
{
    if( do_bake==1.0 && bake_file != "" )
    {
        if( bake_color == 1 )
        {
            color C = 0;
            surface( "Ci", C );
            bake3d(
                bake_file, "", Ps, Ns,
                "interpolate", 1, "coordsystem", spacename,
                "Ci", C );
        }
        else
        {
            bake3d(
                bake_file, "", Ps, Ns,
                "interpolate", 1, "coordsystem", spacename );
        }
    }
}
```

Listing 7.9: Baking using a light source.

The light source in Listing 7.9 above can bake both the color (for color bleeding) or the area (for occlusion). It uses the `surface()` shadeop to get `Ci` from the surface shader that is executing the light source. This means that the light source has to be executed *after* `Ci` has been computed. The execution of a light source can be triggered explicitly using the `illuminance()` construct but one should avoid executing light sources that are not meant for baking[2] and that is why a `__category` string is specified. Here is an example shader that will trigger the execution of the baking light:

```
surface plastic( ... )
{
    Ci = ... ;

    /* execute light sources in the "bakelight" category */
    illuminance( "bakelight", P );
    { /* nothing to do here */
    }
}
```

In this case, the light source should be attached to the scene during the backing pass and detached in the beauty pass. This process can be made somewhat easier by using a *user option* to enable the light during the backing pass and disabling it in the beauty pass so to avoid light linking operations between passes. The `illuminance` call then becomes:

```
...
float enable = 0;
option( "user:bakepass", enable );
illuminance( "bakelight", P, "send:light:do_bake", enable );
{
```

[2] Executing all light sources can have an adverse impact on performance.

```
}
```

In this case the user has to properly set the `bakepass` boolean to '`1`' during the bake pass and to '`0`' during the beauty pass. The `send:` command will set the `do_bake` parameter of the light source using *forward message passing*.

7.7 Point-Based Occlusion and Color Bleeding

Point based graphics reply on a dense point set approximation of scene geometry to compute effects such as occlusion and color bleeding. It is a two pass process that consists of a rapid "geometry conversion" step followed by the main render. The following sections explain the particularities of those algorithms and show how to use *3Delight*'s point-based functionalities.

7.7.1 Difference with Occlusion Baking

Firstly, it is important to remember that point-based occlusion (and color bleeding) is *not* occlusion baking (as explained in Section 7.6 [Baking], page 141): point-based occlusion computes the occlusion during the *beauty* (second) pass, while the baking approach computes the occlusion during the first pass. The "baking" in the point-based method is nothing more than a simple conversion of scene's geometry into a point cloud file. That is why we will refer to the first pass in the point-based approach as the "pre-processing" or "conversion".

7.7.2 Benefits Over Ray-Traced Occlusion

Main benefits are summarized below:

- Due to the nature of the point-based approach, the resulting ambient occlusion effect is smooth.
- Displacements, a costly operation in the ray-tracing case, come at no additional cost in the point-based method.
- No need to declare objects visible to the ray-tracer, this can save a lot of memory.

The drawbacks of point-based occlusion can be summarized in the following points:

- It is an *approximation* to the real occlusion. In general, the obtained occlusion is darker than the ray tracing case and it is necessary to adjust quite a few parameters to obtain the same look. More about parameter adjustments in Section 7.7.4 [Algorithm Control Parameters], page 147.
- The pipeline is obviously more complicated since one need a special pass to create the point cloud(s).

7.7.3 Point-Based Imaging Pipeline

Since point-based algorithms work on point clouds, one need to convert scene's geometry to such representation. This implies a first pass where polygons, surfaces, and other geometrical primitives are converted to a point cloud, and a second pass where the point cloud is read back and used to compute the occlusion or color bleeding effects.

Point Cloud Creation

Converting a scene into a point cloud representation is a straightforward task but care must be taken to ensure the correctness and the optimality of the operation. One should start by disabling all the culling operations to ensure that backfacing and hidden surfaces are not rejected by the renderer prior to shading:

```
Attribute "cull" "hidden" [0]
Attribute "cull" "backfacing [0]
```

Additionally, one should use a dicing method that is independent from the camera view to have correct sampling of points on objects' silhouettes:

```
Attribute "dice" "rasterorient" [0]
```

Since the quality of the image is not important during this first pass, one can lower the filter size for fast rendering:

```
PixelSamples 1 1
PixelFilter "box" 1 1
```

Since no shading is performed during this pass, all surface shaders can be removed from the scene[3] and replaced by a shader that converts micro-polygons to points. The general structure of such a shader is given below:

```
/* A simple surface shader to write out the micro-polygons into a
   point cloud file. */

surface ptc_write(
    string ptc_file = "default.ptc";
    string ptc_coordsys = "world"; )
{
    bake3d( ptc_file, "", P, N,
        "coordsystem", ptc_coordsys,
        "interpolate", 1 );

    Ci = Cs;
}
```

Listing 7.10: An example shader to convert micro-polygons into a point cloud.

There is many points of interest in Listing 7.10:

1. `bake3d()` saves the points as disks, and not dimensionless points. This is performed by taking the area of each micro-polygon using surface derivatives. If needed, it is possible to manually specify the area of a micro-polygon to `bake3d()`:

   ```
   ...
   float A = area( P, "dicing" );
   bake3d( ptc_file, "", P, N,
       "coordsystem", ptc_coordsys,
       "_area", A,
       "interpolate", 1 );
   ...
   ```

   It is suggested not to specify the '`_area`' channel if it is equal to `area(P, "dicing")` (see [area shadeop], page 93) since this will increase the size of the point cloud file unnecessarily (*3Delight* already stores that area in the point cloud file).

2. The '`interpolate`' parameter passed to `bake3d()` is important: since we wish to convert micro-polygons to points, we are interested in micro-polygons' centers and not corners (see [bake3d shadeop], page 110). *Omitting this parameter may lead to unexpected results in the point-based occlusion algorithm.*

[3] Surface shaders that contain displacement should be treated with care or replaced by displacement shaders

3. The normal passed to `bake3d()` is not a `faceforward()` version of the normal. This is important since we need the original surface normal and not an altered one. This also implies that scene geometry should be consistantly oriented in order to get correct results.

If one is to compute point-based color bleeding effects, an additional '`_radiosity`' channel should be stored in the point cloud file:

```
...
color rad = compute_surface_radiance();
bake3d( ptc_file, "", P, N,
    "coordsystem", ptc_coordsys,
    "_radiosity", rad,
    "interpolate", 1 );
...
```

Note that point-cloud creation should be very rapid, so there is no real problem when rendering animated sequences since a point-cloud can be created for each frame (as apposed to baking where it is a good thing to re-use baked data for many frames).

Point Cloud Use

During the main pass, only minor modifications are necessary to the `occlusion()` call to use previously saved point cloud files. An example call looks like this:

```
float occ = 1 - occlusion( P, N, 0,
    "filename", ptc_file, "pointbased", 1 );
```

Color bleeding is computed using the `indirectdiffuse()` shadeop:

```
color bleeding = indirectdiffuse( P, N, 0,
    "filename", ptc_file, "pointbased", 1 );
```

An example for point-based occlusion can be found in `$DELIGHT/ptc_occlusion`.

7.7.4 Algorithm Control Parameters

During the conversion (baking) pass, the important parameters are:

1. Camera Placement. Camera must "see" al the elements for which the point-based occlusion is to be computed. `ptcview` can be used to verify that the save point cloud file contains all the necessary points.
2. Shading Rate. This determines the total number of points in the point cloud. This is an important quality vs speed control knob: higher shading rates will lead to faster processing times and smaller point cloud files at the expense of a lower quality render.
3. Resolution. This has the same impact as the shading rate.

During the main (beauty) pass, the important control parameters are all part of `occlusion()` parameter list (see [occlusion shadeop], page 99):

`maxdist` This parameter is an important look control knob in point-based occlusion. As explained below, point-based occlusion is not good when rendering very dense environments since and this parameter artificially lowers the complexity of the scene for the algorithm.

`maxsolidangle`
This is a quality vs speed control knob. A good range of values is 0.01 to 0.5.

`clamp` Setting this parameter to 1 will force *3Delight* to account for occlusion in dense environments. The results obtained with this parameter on should look similar to what a ray-traced rendering would give. Enabling this parameter will slow down the point-based algorithm by a factor of 2.

7.8 Multi-Camera Rendering

3Delight is able to render many camera views in a single run. This feature is commonly called *multi-camera rendering* and is particularly useful for *stereo rendering*: both the left and right eye are rendered at once. Rendering many views at once can save a substantial amount of time. The main performance gains come from:

1. Scene loading is performed only once for all camera views. The gains in this case are significant for large scenes.
2. Shading is performed only once (in the case of the REYES hider).

7.8.1 Semantics

Multi-camera rendering is enabled by specifying a camera name, previously declared by `RiCamera`, to the `RiDisplay` command. Follows a RIB snippet to illustrate this:

```
FrameBegin
  Projection "perspective" "fov" [60]

  TransformBegin
    # Translate to the left.
    Translate -3 0 0
    Camera "right_eye"
  TransformEnd

  Display "left_eye.tif" "tiff" "rgba"
  Display "+right_eye.tif" "tiff" "rgba" "string camera" ["right_eye"]

  WorldBegin
    ...
  WorldEnd
FrameEnd
```

Listing 7.11: Example of multi-camera usage.

7.8.2 Limitations

The main limitation of the multi-camera rendering feature is the fast that shading only occurs once[4]. This makes the render faster but since the shader is invoked only once all *view dependent* computations such as specular highlights will be incorrect. To invoque the shader once per camera it is necessary to use a special attribute:

```
Attribute "shade" "viewdependent" [1]
```

This will fix the view dependent shading problem mentioned above but will be slower because shading is performed more often. When rendering effects such as occlusion (which is not view dependent) this option should be set to 0.

7.9 Display Subsets

3Delight can renderer an arbitrary number of the so called "secondary displays". Each secondary display can be fed with certain variables from shaders attached to the geometry. This is handy to output many different variables for later compositing or number crunching. But what if one wants to have different geometry, not only output variables, in each display? This functionality is

[4] Not the case when using the '`raytrace`' hider.

achieved using the *grouping membership* attribute and the special '`string subset`' parameter that is accepted by `RiDisplay`. The operation is summerized in two steps:

a. Put geometry in groups that reflect the wanted separation. This can be done using:

```
Attribute "grouping" "membership" ["groupname"]
```

This attribute is further described in Section 5.2.1 [primitives identification], page 44.

b. To render a particular group in a particular display, use the '`subset`' special parameters:

```
Display "layer.tif" "tiff" "rgb" "string subset" ["groupname"]
```

Note that it is exactly the same attribute as used for *trace group memberships*, as explained in Section 7.3.4 [Trace Group Membership], page 128. One can combine many groups together, as in,

```
Display "layer.tif" "tiff" "rgb" "string subset" ["group1,group2"]
```

Or exclude particular groups, as in,

```
Display "layer.tif" "tiff" "rgb" "string subset" ["-group3"]
```

The display subset feature would be incomplete if it was not possible to have a *different* alpha channel for each display: when rendering a foreground and a background one expects to be able to composite the two layers together. This is achieved by adding the *alpha* channel to the display:

```
Display "foreground.tif" "tiff" "rgba" "string subset" ["foreground"]
Display "foreground_ambient.tif" "tiff" "color ambient,alpha"
"string subset" ["foreground"]
```

7.10 Exclusive Output Variables

IMPORTANT: Although this functionality is still implemented and useful in some occasions, we consider that the *display subsets* feature (see Section 7.9 [Display Subsets], page 148) supersedes *exclusive output variables* and offer even more flexibility.

By Moritz Moeller

3Delight possesses the ability of rendering data to an arbitrary number of so called "secondary displays", usually image files on disk. These displays are fed with certain variables from shaders attached to the geometry. Since any variable in a shader can be used to output data this way, they are also called "Arbitrary Output Variables" or AOVs for short[5].

The main use of AOVs is decomposing the inner workings of a shader for later recombination in a compositing package, where each AOV can be altered rather quick (compared to the time it would require to re-render a "beauty" image containing all of them combined into a final and more or less fixed result).

However, AOVs come with two problems. Firstly, every shader that doesn't write to an AOV explicitly is treated as implicitly rendering black to it. Thus, if you had, say, a furry creature and you wanted to render a diffuse pass for the creature's skin and its fur each, respectively, the skin geometry would correctly occlude the fur on the back of the creature and the fur geometry would correctly occlude any skin of which it was in front of. When you add those two AOVs together in a compositing package, not using any alpha channel at this stage yet, you get the exact same image as if you had rendered them into one image.

[5] In other renderers that have added this functionality only in recent years, this is most commonly referred to as "rendering in passes".

Commonly, one stage in compositing is all about adjusting black and white points, gain, gamma and color correction of image data. In the described example, this might give edge issues with geometry like fur. Particularly with many AOVs that make up a single element this requires a lot of precautions and even then, artifacts will often show up (think of several specular-, diffuse- and translucency AOVs making up the fur with each element requiring individual adjustments in the compositing package). To circumvent this problem altogether one can ask *3Delight* not to matte certain AOVs. This is done with the '`exclusive`' flag (see Table 5.1). This flag is accepted by the `RiDisplay` and `RiDisplayChannel` calls. When set to '`1`', the respective display (channel) will not be matted by any other AOV. Still, certain objects might matte such an "exclusive" AOV, if they have the `RiMatte` attribute switched on and output the AOV. To suppress even those objects from matting an AOV, *3Delight* supports a '`matte`' flag for `RiDisplay` and `RiDisplayChannel`. A totally unmatted AOV can be produced by setting '`matte`' to '`0`' and '`exclusive`' to '`1`'.

The second problem that arises is that certain AOVs contain geometry that requires an alpha channel different from the primary display's alpha. One could of course just output another single channel AOV with the coverage information of each AOV that might need an individual alpha in compositing. For your convenience, *3Delight* provides a shortcut for this: the *calculatealpha* flag, supported by the `Display`, when set to '`1`', will add a correctly identified[6] alpha channel to the display[7].

To complete the set of features that make a compositor's life a happier one, *3Delight* provides a way to produce images with unassociated alpha. It is called `"associatealpha"` and has to be turned off by setting it to 0, to get unmultiplied output. The AOVs that the renderer then produces are ready to be adjusted in the target compositing application, with the requirement removed, of dividing them by their alpha first. For example, in Apple's *Shake*, this means you can skip the *MDiv* node behind each *FileIn* node that loads AOVs produced by *3Delight*.

7.11 Rendering Outlines for Illustrations

3Delight has advanced "inking" features for "toon rendering" and other illustration purposes. This is implemented using edge detection algorithms in *image space*. All edge detection features are controlled through parameters to the `RiDisplay` call. Since *3Delight* can output many such displays in one render, it becomes possible to output different edge detection results into different files or framebuffers. For example, one display can output edge detection based on depth variation and another one on surface normals variation.

[6] If supported by the display driver.
[7] Note that this will increase the number of actual channels in the Display or DisplayChannel by one.

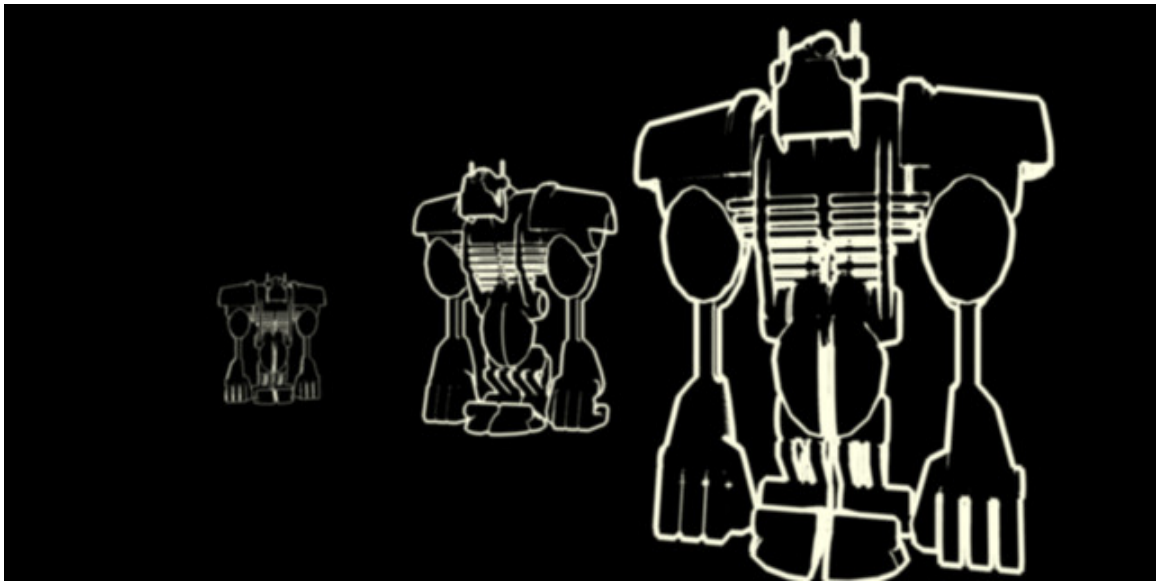


Figure 7.5: Outlines on `Oi` (opacity) and `z` (depth). Note how outlines get thinner with distance. Example scene available in `$DELIGHT/examples/toon`.

Additionally, it is possible to output, in the same display, edges detected on many variables at once (such as depth, normals *and any other surface variable*). Table 7.1 contains the list of all edge detection parameters accepted by `RiDisplay`.

Parameter	**Default Value**	**Description**
"string edgevariables"	""	Comma separated list of variables to be used for edge detection.
"float edgethreshold"	1e30	Controls edge detector sensitivity.
"float edgefilterwidth"	2	Controls edge width. This can be a fractional number and can be set to smaller than 1.0 to get very thin outlines. Note that to have proper anti-aliasing on thin outlines, one should use higher `PixelSamples`.
"float edgewidth"	0	Controls edge width. This can be specified instead of edgefilterwidth. It is relative to the frame width (eg. 0.001 on a 1k frame will give 1 pixel edges).
"color edgecolor"	[1 1 1]	Color of the generated edges.
"float edgefadeout"	[near far minwidth]	Controls edge thickness fadeout with depth. *near* and *far* specify the depth range in which the fadeout will be performed. *minwidth* specifies the minimum outline width.

Table 7.1: RiDisplay parameters for edge detection.

At least `edgevariables` and `edgethreshold` must be specified to enable edge rendering. It works by looking for abrupt variations of one or more output variables over the image. For example, rendering outlines based on changes in depth would be done with:

```
Declare "outline" "color"
Quantize "outline" 255 0 255 0.5
Display "depth_outline.tif" "tiff" "outline"
    "string edgevariables" "z"
    "float edgethreshold" 0.1
    "float edgefilterwidth" 1.5
```

The `Declare` command specifies an aribitrary output variable which is used as a placeholder for edge data. A `Quantize` command is used to specify that the output is to be 8-bit. Variables specified with `edgevariables` (in this case only 'z') are the ones processed to detect edges.

It is possible to use any predefined or user-defined output variable to control the presence of outlines. It is also possible to process more than one variable in the same image. For example, to emphasize the contour of the object in the previous example we would use:

```
Declare "outline" "color"
Quantize "outline" 255 0 255 0.5

DisplayChannel "N"
    "float edgefilterwidth" 1
    "float edgethreshold" 0.3

DisplayChannel "Oi"
    "float edgefilterwidth" 2.5
    "float edgethreshold" 0.1
    "color edgecolor" [1 0.3 0.3]

Display "mixed_outline.tif" "tiff" "outline"
    "string edgevariables" "N,Oi"
```

Notice that this time we base most of the edge detection on normals instead of depth. Which is better depends on the scene involved and the goals of the artist. Finally, *3Delight* allows basic compositing of detected edges over any output variable. For example, to see what the edges would look like over the normal image:

```
DisplayChannel "N"
    "float edgefilterwidth" 2
    "float edgethreshold" 0.3
    "color edgecolor" [0 0 0]

Display "cheesy.tif" "tiff" "rgb"
    "string edgevariables" "N"
```

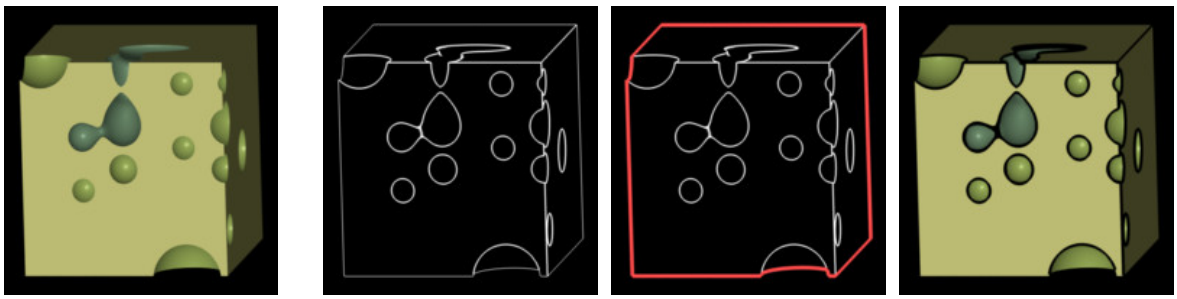


Figure 7.6: Example of edge detection using normals. The first image is a standard render. The next one is the result of finding normal outlines. The third image is a combination of normal outlines and opacity outlines. The final image is of black outlines composited over the rgb display.

7.12 Dicing Cameras

As we already know, *3Delight* renders geometry by tessellating it into micro-polygons. This way of doing things makes running displacement shaders straightforward and efficient. One potential problem might arise though: when rendering sequences with a moving camera, the renderer might tessellate the geometry slightly differently from frame to frame, leading to displacement shaders to react differently between frames and this in turn may lead to *pops* in static geometry. This usually happens in displacement shaders that have a high perturbation frequency. Although this problem can be solved using lower shading rates and proper anti-aliasing it is often easier to use a *dicing camera*.

A dicing camera is a camera that is declared along with the main camera and is usually static in the animation sequence, so that objects are always tessellated the same way from frame to frame. One can declare such a camera using the `RiCamera` command:

```
Display "example.tif" "tiff" "rgb"

Format 640 480 1
Projection "perspective" "fov" 60
Translate 0 0 5

TransformBegin
  Translate 0 0 100
  Camera "faraway"
TransformEnd

WorldBegin
  Translate 0 -1.5 0
  Rotate -90 1 0 0
  Attribute "dice" "referencecamera" "faraway"
  Geometry "teapot"
WorldEnd
```

Listing 7.12: Declaring and using a dicing camera.

In this example, a teapot is diced using a camera named "faraway" that has been placed very far (see [dice-referencecamera], page 52). So rendering the image will produce a teapot with very rough tessellation. A few notes:

- It is possible to declare multiple cameras and use them on different objects, if desired.
- A camera will take into account the entire graphical state when declared: screen window, resolution, field of view, etc... This means that one can control the shading rate of a camera by changing the resolution prior to its declaration (lower resolution means higher shading rate).
- A list of cameras can be specified instead of just one camera. In that case *3Delight* will tesselate using the camera that is the "closest" to the rendered objects. An example RIB snippet"

  ```
  Attribute "dice" "referencecamera" "faraway,closer"
  ```

IMPORTANT**:** Before integrating dicing cameras into your pipeline, make sure the problem has been correctly identified. Dicing cameras will make your pipeline slightly more complicated and many problematic displacement shaders can be healed by proper programming.

7.13 Level of Detail

Level of detail gives the ability to specify several definitions of the same model, letting the renderer select an appropriate definition considering the size of the model in the rendered image and some user guidance. LOD can be thought of as being a mipmapping algorithm that works on geometry instead of textures.

As an example, to render a forest of trees, one might specify two tree models: a complex one with detailed structures and leaves and a simpler one with only the basic structures and fewer (and most probably bigger) leaves. The renderer is able to "cross-fade" between the provided models so that the transition between two different levels of detail remains smooth. As with other features (such as motion blur and depth of field), level of detail quality depends on the *oversampling* used (specified using `RiPixelSamples`): the higher the oversampling the smoother the transtion between detail levels will be.

When used correctly, level of detail can provide signficant savings in terms of memory use and rendering time; it is important however to get things right:

1. Only use level of detail for complex models that can be visualized at different sizes. Crowds, hair patches and trees are good candidates for level of detail rendering.
2. Specify a correct bounding box with `RiDetail`. This enables the renderer to estimate correctly which model is needed for a particular screen size[8].
3. Use procedurals (such as `RiDelayedReadArchive`) to specify the models. This avoids loading models that are not needed for some particular detail level:

```
AttributeBegin
  Detail -1 1 -1 1 -1 1

  DetailRange 0 0 10 20
  Procedural "DelayedReadArchive" ["tree-simple.rib"] [-1 1 -1 1 -1 1]

  DetailRange 10 20 120 130
  Procedural "DelayedReadArchive" ["tree-medium.rib"] [-1 1 -1 1 -1 1]

  DetailRange 120 130 1e38 1e38
  Procedural "DelayedReadArchive" ["tree-complexe.rib"] [-1 1 -1 1 -1 1]
AttributeEnd
```

4. Transitions between levels of detail must be as fast as possible. This means that values provided by different `RiDetailRange` calls should have small overlapping windows. Large overlapping windows forces the renderer to keep **two** model definitions in memory in order to "cross-fade" between them. Of course, abrupt transitions may cause undesirable geometry "pops" in animation so a fine balance is required.

Note: Different levels of detail can have different rendering *attributes*. This means that costly resources (such as displacement maps) can be stripped off the lower detail models.

7.14 Network Cache

3Delight offers a special extension for more efficient rendering in networked environments: a local file cache for textures and archives to minimize network traffic and file server load. This proves particularly powerful when using a large quantity of rendering servers.

The working principle of *3Delight*'s network cache is to copy files to a local file system and reuse them when needed. If either a texture or a RIB archive is cacheable and the file cache is full, one or more files are removed from the cache to make space for the new one; a LRU (Least Recently Used) strategy is used to choose which texture(s) or archive(s) to remove from the cache.

Eligibility to caching is different for textures and archives, in summary:

[8] It is also possible to use `RiRelativeDetail` to modify renderer's detail estimate.

'`Textures`'
'`Shadow maps`'
'`Environment maps`'
: Cached when accessed through the `texture()`, `shadow()` and `environment()` shadeops, meaning that textures that are never accessed do not waste cache space.

'`Delayed Archives`'
: Cached only if occlusion culling determines that the "archive is visible", meaning that invisible models do not waste cache space.

'`Archives`' Cached immediately.

The design of the network cache was justified by the following observations:

- Rendering a sequence on a render farm puts a great amount of pressure on the network and file server(s): starting a heavy job simultaneously on many machines can make the network unusable and the file server non responsive, especially on slow networks or weakly configured environment.
- Accessing a file locally is faster than accessing it on a network mounted system such as NFS. Clearly, texture and archive access can have a significant impact on rendering speed.
- File server(s) load would be greatly minimized since file access requests happen less often.
- The nature of the rendering process makes it very probable that a texture or archive used during a rendering of a given image is used again in forthcoming renderings of more images, so keeping it handy is a good investment.
- Storage is cheaper, and simpler, than bandwidth!

Activating the Network Cache

To enable the network cache, use the following `RiOption`:

```
Option "netcache" "string cachedir" ["/tmp/3delight_cache/"]
```

This informs *3Delight* to use a directory named `/tmp/3delight_cache/` to cache textures files. If the directory is already created, *3Delight* uses whatever cached files it contains. It is important to use a *locally mounted* directory since caching network files on a network volume is useless. The chosen directory should be dedicated to *3Delight*: *do not put any files in that directory since they can disappear without notice.* It is possible to disable the cache again by calling the same `RiOption` with a null file name. This can be useful in multi-frame RIBs.

```
Option "netcache" "string cachedir" [""]
```

Cache size is controlled using another `RiOption`. For example, to specify 1000 megabytes of network cache, use:

```
Option "netcache" "integer cachesize" 1000
```

Specifying a size which is smaller than the actual cache size causes files to be removed from the cache until the specified size is reached.

There is no need to specify which files to cache, *3Delight* detects automatically slow access files and caches them. Slow access files are files mounted on a NFS disk or a CD-ROM.

It is also possible to instruct *3Delight* to cache some output files by using:

```
Option "netcache" "integer writecache" 1
```

This can save a great deal of bandwidth when the output file is going to be reused as the input to another render on the same machine. Shadowmaps are a common example of this. Even for files which are not going to be reused, it is usually more efficient for the file server to receive them in one large chunk. Note that because the size of output files is not known in advance, they do not count towards the maximum cache size until they are finished writing. This means the cache can temporarily grow larger than the set size so ensure there is some extra space available on the drive used.

It is possible to override the network cache's choices about which files are eligible for caching. Refer to [Network Cache Options], page 41 for more information.

Purging the Network Cache

The network cache can be purged manually at any moment *if no renderings are running on the machine*. Simply erase the directory and all its contained files:

```
rm -rf /tmp/3delight_cache
```

Using Directory Textures For Large Textures

The network caching algorithm copies the entire content of the texture from the server to the local machine, which is desirable in most cases. In some situations however, when the textures are very large but the quality settings are not high enough to see all the details, copying the entire texture might be a waste of resources. For such cases, *3Delight* provides "directory textures": textures saved in a format that permits the caching of data with a *mipmap granularity level*. The format is quite simple: the texture becomes a directory with all the mipmap levels saved as individual TIFF files that area cacheable independently. Creating such textures is straightforward:

```
tdlmake -dirtex source.tif destination.tdl
```

This command will create a directory named "destination.tdl" that contains all the mipmap levels.

To know if directory textures are beneficial in a particular situation, one should analyze texture access statistics to see if the most detailed mipmaps are accessed or not.

Safety

Many precautions have been taken to ensure the proper operation of the network cache:

1. *3Delight* does **not** access the original textures or archives in any dangerous way, only **reading** is performed on those files.
2. The cache is kept synchronized with the files it mirrors: if an original texture is newer than the cached one, the cache is updated.
3. A file is identified by its *full path*: files that have the same name in different directories do not collide.
4. UNIX file links are resolved prior to caching, this ensures that a given file is cached only once even if many links point to it.
5. The network cache is multiprocess safe. Even if many renders are running on the same machine, the cache is kept in a consistent state: one *3Delight* instance does not remove a texture or archive used by another instance!
6. Cache directory is created with full access permissions to '`user`' and '`group`', but only read access to '`other`'[9].
7. If, for any reason, *3Delight* is unable to cache a file, it reverts to use the original, and this is *the worst case scenario*.

7.15 RIB Output

It is sometimes desirable to redirect the RIB stream into a file instead of sending it to the renderer. This is particularly useful for applications that link directly with the Ri client library (see Section 10.3 [linking with 3delight], page 244). Some renderers come with a specialized library that is intended

[9] Permissions mask : `0775`.

for such usage[10] while *3Delight* provides the same functionality through the library that is used for rendering in three different ways:

1. By specifying an output file name to `RiBegin` (specifying an empty file name as in `RiBegin("")` outputs the RIB stream to `stdout`). This is the recommended way for exporting geometry from 3D packages. It is possible to concatenate a RIB stream to an existing file by using `RiBegin(">>filename.rib")`. It is also possible to pass the RIB stream to custom filters by using a command such as `RiBegin("| filteringscript | gzip > final.rib.gz")`.
2. By specifying a file descriptor using the `Option "rib" "pipe"` option.
3. By setting the `RISERVER` environment variable to a valid file name before rendering (more precisely, before the first `RiBegin` call). This environment variable can be of great help when debugging since one can redirect the RIB stream to a file for analysis, without prior code modification. Binary RIB output is possible by setting the `RIFORMAT` environment variable to '`binary`'. Compressed output is possible by setting the `RICOMPRESSION` environment variable to '`gzip`'.

Additional controls on RIB output are provided through `RiOption`. One is for binary RIB output and the other is for procedurals expansion. When procedurals expansion is enabled, the output is a flat RIB, including the content of all accessed archives (read through `ReadArchive`) and procedurals (called by `RiProcedural`). The actual options are better illustrated through an example:

```
/* This one must be called before RiBegin or it will have no effect.
   With this option the output stream will be gzipped. */
RtString compression[1] = {"gzip"};
RiOption( "rib", "compression", (RtPointer) compression, RI_NULL );

RiBegin( "test.rib" );
    /* Set output format to binary */
    RtString format[1] = {"binary"};
    RiOption( "rib", "format", (RtPointer)format, RI_NULL );

    /* Enable procedurals expansion during output. */
    RtString expansion[1] = {"on"};
    RiOption( "rib", "callprocedurals", (RtPointer)expansion, RI_NULL );

    ... Scene description follows here ...
RiEnd( );
```

Listing 7.13: RIB output using `RiBegin`.

Some important remarks about procedurals expansion:

- It is disabled by default, so all standard procedural calls (such as `RiProcDynamicLoad`) are replaced by their RIB counterpart (e.g. `Procedural "DelayedReadArchive"`). User defined procedurals are always expanded.
- When calling a procedural (such as `RiProcDynamicLoad`) the level of detail is set to `RI_INFINITY`.
- Contrary to normal rendering, RIB output does not perform any kind of frustum culling or visibility checks; this means that procedurals are opened as soon as they are issued. This ensures that the entire scene is included.
- All used archives/executables must be present in *default* search paths.
- It is possible to cause procedurals to be turned into archives read with `"DelayedReadArchive"` by using the `"rib" "callprocedurals"` option (see [RIB Output Options], page 42).

[10] `librib.a` or `libribout.a`

```
/* Set output format to binary */
int fd = open( "test.rib", O_WRONLY );
RiOption( "rib", "pipe", &fd, RI_NULL );

/* Set output format to binary. Can also be done _inside_ the Begin/End
   block. */
RtString format[1] = {"binary"};
RiOption( "rib", "format", (RtPointer)format, RI_NULL );

/* Enable procedurals expansion during output. Can also be done _inside_
   the Begin/End block. */
RtString expansion[1] = {"on"};
RiOption( "rib", "callprocedurals", (RtPointer)expansion, RI_NULL );

RiBegin( RI_NULL );
... Scene description follows here ...
RiEnd( );

close( fd ); /* don't forget to close that file! :> */
```

Listing 7.14: RIB output using pipes

`renderdl` can also be used to redirect the RIB stream to a file by using the `catrib` option (see [renderdl options], page 8). Redirecting a RIB into another RIB may sound useless but is in fact useful in many regards:

- `renderdl` can output binary encoded RIBs (using `-binary`) which are much smaller in size than their ascii counterpart.
- renderdl can read binary RIBs so it can be used to convert those into a humanly readable ASCII format.
- It is sometimes desirable to "flatten" a RIB which includes many archives or procedurals, this can be done using the `-callprocedurals` option.
- The resulting RIB is nicely formatted!

EXAMPLES

Output the content of a RIB in "gzipped" binary format:

```
> renderdl -catrib -binary -gzip teapot.rib > teapot.binary.rib
```

Do the same using environment variables (working in a tcsh shell):

```
> setenv RIFORMAT binary
> setenv RISERVER teapot.binary.rib
> renderdl teapot.rib
```

Append the content of a RIB to an existing file, expanding all procedurals and including all archives:

```
> renderdl -catrib -callprocedurals frame12.rib >> sequence.rib
```

7.16 Optimizing Renderings

3Delight is heavily optimized: only "state of the art" algorithms are used in every area and continuous re-evaluation of every functionality against the most recent techniques ensures that every release is a step forward in terms of speed and memory consumption. This doesn't mean that no user guidance is required, to the contrary, carefully chosen options and scene layouts can speed up renderings enormously. The following sections give some important advices to that matter.

Rendering Options Tuning

Be careful to select the right options the render: the fastest option settings without a quality tradeoff.

Format (image resolution)
: *Image size is directly proportional to rendering time*, that is why it is very important to select the right resolution for the render: lower resolution means less used memory and faster renders.

PixelSamples and PixelFilter
: More pixel samples means slower renders. A "`PixelSamples 6 6`" is more than enough for final renders, even with motion blur. Using the high quality "sinc" filter requires a wider support ("`PixelFilter "sinc" 4 4`" or more) that is why it is generally slower.

ShadingRate
: Test renders should specify "`ShadingRate 4`" or more. Final renders should use "`ShadingRate 1`". Smaller shading rates are rarely needed and are often used by shaders that produce high frequencies[1].

Bucket and Grid size
: In our experience, a bucket size of 16x16 and a grid size of 256 is a very good general setup for *3Delight*. This can be bettered by doing experiments on a particular scene. Also, a given grid should cover approximately one bucket in raster space: so for a shading rate of 1 and a bucket size of 32x32 the grid size should be 1024.

Horizontal or Vertical ?
: By default, the renderer issues buckets row per row. This is not the optimal bucket order if the image contains objects that are thin and horizontal. Use "Option "render" "string bucketorder" "vertical"" to specify a vertical bucket issuing strategy (column by column). This trick is likely to save memory only, impact on speed is negligible in general.

Multithreading

Use multithreading or multiprocessing when possible. Depending on the nature of the rendered scene, performance improvements could be linear with the number of processes used. Refer to Section 7.1.4 [Performance Notes], page 122 for details.

Ray Tracing

Avoid ray tracing, if possible! Ray tracing is inheritantly slow since it implies some intensive number crunching. Most importantly, it forces *3Delight* to retain geometry for the entire render, meaning higher memory usage. An exception to the above is ray traced shadows in small to medium sized scenes, which in our experience render very fast. Here is a few advices concerning ray tracing:

- Use `Attribute "visibility"` to specify which objects are visible to traced rays, minimizing the number of "ray-traceable" objects is beneficial for both speed and memory usage. Additionally,

[1] Those shaders benefit from properly anti-aliased code.

using simpler objects ("stand-ins") for ray tracing is a good idea, those should be made invisible to camera rays.

- Limit maximum ray depth using `Option "trace" "integer maxdepth"`. A value of '2' is enough in most cases. Remember that increasing this limit may slow down the renderer *exponentially*.
- Use grouping memberships to ray trace only a subset of the scene; this helps only for large scenes.
- Use shadow maps instead of ray traced shadows. Those are faster to generate and are reusable. Colored shadows can be rendered efficiently using deep shadow maps. See Section 7.2 [Shadows], page 122.
- Use environment maps or reflection maps to render reflections. In most cases, the viewer won't see the difference.
- Use image based lighting instead of brute force global illumination. See Section 7.4.4 [Image Based Lighting], page 133.
- If using ray traced shadows, use the '`opaque`' or '`0s`' transmission modes, this avoids costly shader interrogations.
- Shaders triggered by a secondary (ray traced) rays should use a less expensive code. Secondary rays can be detected using the `rayinfo()` shadeop. See [rayinfo shadeop], page 117.
- Use true displacements and motion blur only when necessary (in the context of ray tracing).

Textures

It is well known that texture access eats up a significant amount of rendering time. Textures are also known to be great resource hogs.

- Do not use oversized textures or shadow maps. Appropriate texture or shadow map dimensions depends on final image resolution and should be adjusted by performing experiments.
- Use compressed textures, it pays off in most cases. Two benefits are less disk usage and less network traffic. `tdlmake` has several options for texture compression, see [tdlmake options], page 15.
- Use "box" filtering if possible (see [texture shadeop], page 107). The "gaussian" filter gives nicer results in general (since it is a high-quality anisotropic filter), but in many cases (such as on smooth textures) the "box" filter can be used without any visible quality penalty.
- Increase texture cache memory. This is particularly important when using deep shadow maps since one tile of DSM data takes much more space than a tile from a simple texture.
- Use the network cache, as described in Section 7.14 [Network Cache], page 154. When rendering over a network, caching textures locally can save a substantial amount of time. We timed speed gains of 15% and more with heavy production scenes on very large networks (more than a hundred of machines).

Geometry

- **Use higher order surfaces**. *3Delight* runs much more efficiently when provided with high level geometry such as NURBS and Subdivisions. Feeding finely tessellated polygons to the renderer is not recommended since it is non optimal in many ways. In addition, using curved surfaces gives most certainly nicer results (no polygonal artifacts).
- **Use procedural geometry** whenever possible; this can have a considerable impact on rendering speed, especially in complex scenes. *3Delight* loads only a procedural object if it is visible and disposes it after use, this is beneficial for both speed and memory usage. More on procedurals in Section 5.3.8 [procedural primitives], page 57.

- **Use level of detail** in complexe scenes. More on level of detail at Section 7.13 [Level of Detail], page 153.

Shaders

The "trend" is toward sophisticated looks, complex illumination models, anti-aliased shaders and volume shaders. This is why shading can eat up a non negligible slice of the total rendering time. It is really worth the effort to carefully optimize shaders.

- **Use `uniform` variables whenever possible**. Remember that *3Delight*'s shaders run in SIMD, which means that an operation processes many shading samples at once. `uniform` variables are computed only once *per grid* contrary to `varying` variables which are computed once *per micro polygon*. Consider the following example:

```
surface test( float i_angle = PI * 0.5; )
{
    float half_angle = i_angle * 0.5;

    ... more code goes here ...
}
```

  There is a "hidden" overhead in this shader since *i_angle* is `uniform` and *half_angle* is `varying`[2]. This means that *half_angle* is initialized to the same value (*i_angle*/2) for *each micro polygon* on the grid. Although the shader compiler optimizes some `varying` variables into `uniform`, declaring *half_angle* as `uniform` guarantees to avoid unnecessary operations.

- **Avoid using conditionals in shaders**. Conditionals can stall *3Delight*'s SIMD pipeline which has an impact on shaders' execution speed.

- **Use light categories**. Evaluating light sources is a non negligible part of total shading time. Light categories can help avoid evaluating light source shaders which are not needed in some particular context. A very good example is an atmosphere shader that uses a "ray marching" algorithm to compute volume's translucency and color: for each sample along the marched ray `illuminance()` is called to compute incoming light, which triggers the evaluation of all attached light sources. To limit shader evaluation to only a subset of lights which effectively contribute to atmosphere, one can use the following:

```
/* Only evaluate light sources that are part of the
   "foglight" category" */

illuminance( "foglight", currP )
{
    ... compute volume attenuation+color here ...
}
```

  Listing 7.15: Using light categories

  Note that using message passing such as in the following example is **not** the right way to reject light sources from illuminance loops.

[2] Remember that shader's parameters are `uniform` by default and that local variables are `varying` by default

```
illuminance( currP )
{
    uniform float isfoglight = 1.0;
    lightsource( "isfoglight", isfoglight );

    if( isfoglight == 1.0 )
    {
        ... compute volume attenuation+color here ...
    }
}
```

Listing 7.16: Erroneous use of message passing

In the erroneous example, light sources are evaluated but not included in the result, *leading to little or no time gain*. Light categories insure that light sources are not evaluated, saving shader execution time. It is not uncommon to have tens of light sources in a scene that is why it is *important to use light categories, especially in volume shaders*.

- **Compile shaders with `-O3`**. Default option is `-O2` which does not include some useful optimization.

- **Compile shaders into machine code with `--dso`**. This pays off with very expensive shaders, such as atmosphere shaders.

- **Look for expensive shaders by using the `"__CPUtime"` output variable**. It is possible to output an image where the luminosity represents the cost of shading by using a `Display` statement such as:

```
Display "+cputime.tif" "tiff" "__CPUtime".
```

Objects which are more expensive to shade will be brighter. You can use this image as a guide to which shaders might be worth simplifying.

7.17 Using Ri Plug-in Filters

Ri plug-in filters are an important extension to the RenderMan interface, providing flexibility unknown to non RenderMan systems and pipelines. In short, a *Ri plug-in* is a component lying between the application that generates Ri calls (such as a RIB reader like `renderdl`) and the rendering core. Being in such a strategic place, the plug-in can alter or filter, in any desirable way, all Ri commands received by the renderer.

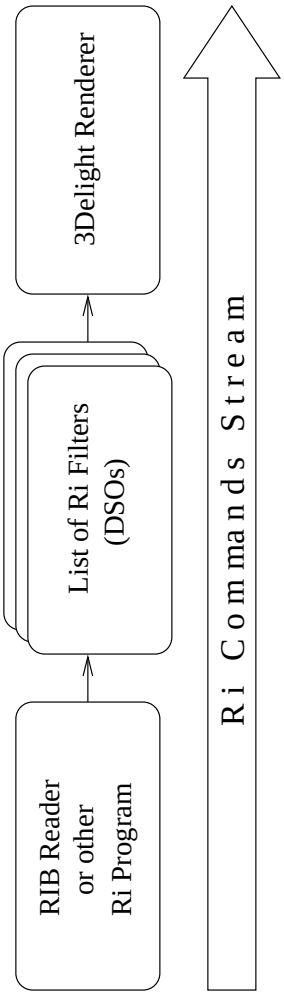


Figure 7.7: Ri plug-in filters lie between the program that generates the Ri calls (such as the RIB reader, `renderdl`) and the 3DELIGHT renderer.

7.17.1 Ri Plug-in Anatomy

As pictured in Figure 7.7, a plug-in is a DSO[3] that is loaded prior to the scene description phase. This section explains the general ideas behind the structure and execution of Ri plug-ins, a formal description of all functions available to plug-ins can be found in Section 10.1.3 [Ri Filter Plug-ins], page 212.

Initialization

Each DSO provides a `RifPluginManufacture` entry point that creates an instance of the plug-in. Each such plug-in derives from the `RifPlugin` abstract class. At this stage, the plug-in decides which Ri commands will be filtered and what to do with Ri commands that are not processed by the plug-in: either *continue* execution of other plug-ins in the chain or *terminate* immediately in the chain of execution. Note however that a plug-in can decide to run in the "continue" mode but

[3] Or DLL on Windows.

switch to the "terminate" mode later on, and vice-versa. As an example, Listing 7.17 shows the initialization code of a very simple plug-in filter that computes the total area of all spheres in some given scene.

```
#include "ri.h"
#include "rif.h"

#include <stdio.h>
#include <string.h>
#include <math.h>

class Example : public RifPlugin
{
public:
    Example()
    {
        m_filter.Filtering = RifFilter::k_Continue;
        m_filter.SphereV = myRiSphereV;
        m_area = 0.0f;
    }
    virtual ~Example()
    {
        fprintf( stdout, "Total RiSphere area: %f\n", m_area );
    }
    virtual RifFilter &GetFilter() { return m_filter; }
private:
    float m_area;
    RifFilter m_filter;
    static RtVoid myRiSphereV(
        RtFloat radius, RtFloat zmin, RtFloat zmax,
        RtFloat tmax, RtInt, RtToken[], RtPointer[]);
};

RifPlugin *RifPluginManufacture( int argc, char **argv )
{
    return new Example();
}

void RifPluginDelete( RifPlugin *i_plugin )
{
    delete i_plugin;
}
```

Listing 7.17: Example of a simple Ri plug-in filter.

In the constructor, we initialize the default filtering mode to '`k_Continue`' which means that all ignored Ri calls simply continue their normal execution path. Additionally, we set the `SphereV` member of `RifFilter` to point to our function since we desire to compute sphere's area. We also set `m_area` to zero since this is our sphere area counter.

Execution

When 3DELIGHT executes an Ri command, it will first process loaded plug-in filters. Filters will be executed in the order they were loaded. Here is a pseudo-code of the renderer's first action when it executes an Ri command:

```
00: while ( current filter )
01:    if( current filter catches this Ri command )
02:        call filter's entry point for Ri command
03:        return;
04:    else if( current filter has the terminate flag )
05:        /* current Ri command is not processed by the filter and
06:           the terminate flag is set: exit skipping all remaining
07:           filters in the chain */
08:        return;
09:    end if
10:
11:    /* current Ri command is not processed by the filter, but the
12:       continue flag is set, so continue with other filters in the
13:       chain */
14:
15:    current filter = next filter in chain;
16: end while
17:
18: ... continue execution with renderer's Ri command ...
19:
```

Listing 7.18: Pseudo code explaining what the renderer does when executing a Ri command.

As shown in lines 02 and 03, as soon as as the renderer encounters a plug-in filter that handles the Ri command, it calls it and exits from the Ri command. This means that if the filter doesn't call the Ri command explicitly, the execution will not continue "down the chain". Continuing Listing 7.17, the `myRiSphere` procedure would look like this if the wanted action is to continue with renderer's `RiSphere` code:

```
RtVoid Example::myRiSphereV(
    RtFloat radius, RtFloat zmin, RtFloat zmax,
    RtFloat tmax, RtInt n, RtToken tokens[] , RtPointer params[] )
{
    Example *current_plugin = (Example *)RifGetCurrentPlugin();
    current_plugin->m_area += 4.0f * M_PI * radius * radius;

    /* Call RiSphereV to continue in the filter chain */
    RiSphereV( radius, zmin, zmax, tmax, n, tokens, params );
}
```

Note, in the code snippet above, how we obtain the current active plug-in using `RifGetCurrentPlugin()`. This is necessary since `myRiSphereV()` is *static* and only the renderer knows which instance of the '`Example`' plug-in is running. At the end of the function, calling the `RiSphereV` command insures that execution continues normally and that the sphere is rendered.

7.17.2 Using `renderdl` to Load Ri Plug-ins

The simplest way to load Ri plug-in filters is by specifying them as a command line argument to `renderdl`. As explained in Section 3.1 [Using the RIB renderer], page 7, `renderdl` accepts the `-rif`,

`-rifargs` and `-rifend` parameters to handle plug-in filters. If we want to run our example filter on a simple RIB:

```
% renderdl -rif example.so
Translate 0 0 10
WorldBegin
Sphere 1 -1 1 360
Sphere 0.5 -0.5 0.5 360
WorldEnd
^D  (This is a CTRL-D)
Total RiSphere area: 15.707964
```

Passing arguments to Ri plug-ins is straightforward using the `-rifargs` and `-rifend` parameters:

```
% renderdl -rif somerif.so -rifargs -arg1 -arg2 -arg3 -rifend  somerib.rib
```

This example will load `somerif.so` and pass three arguments to the `RifPluginManufacture()` function. Chaining multiple Ri plug-ins on the command line is accomplished by using the `-rif` option many times:

```
% renderdl -rif somerif.so -rigargs -s -rifend \
        -rif anotherrif.so -rigargs -p -rifend
```

In this case, two filters will be loaded and executed in sequence: the first one, `somerif.so`, will receive `-s` as an option and the second one will receive `-p`.

It is sometimes useful to run a RIB through one or more filters and store the filtered RIB back on disk. This is accomplished using the `-catrib` command line option. As an example, lets modify the `myRiSphereV` function defined in Listing 7.17 so that it outputs `RiDisk`s instead of spheres:

```
RtVoid Example::myRiSphereV(
    RtFloat radius, RtFloat zmin, RtFloat zmax,
    RtFloat tmax, RtInt n, RtToken tokens[] , RtPointer params[] )
{
    Example *current_plugin = (Example *)RifGetCurrentPlugin();
    current_plugin->m_area += 4.0f * M_PI * radius * radius;

    /* Call RiDisk instead for more fun ... */
    RiDisk( 0.0f, radius, tmax, RI_NULL );
}
```

When running `renderdl` on the following simple RIB scene:

```
% renderdl -catrib -rif example.so
Translate 0 0 10
WorldBegin
Sphere 1 -1 1 360
Translate 0 0 -1
Color 1 0 0
Sphere 0.5 -0.5 0.5 360
WorldEnd
^D
```

We obtain the following RIB:

```
Translate 0 0 10

WorldBegin  # {
  Disk 0 1 360
  Translate 0 0 -1
  Color [ 1 0 0 ]
  Disk 0 0.5 360
WorldEnd  # }
```

Spheres have been replaced by disks as intended.

Note: `renderdl` uses 3DELIGHT's `RifPluginLoad()` function to load plug-in filters. Plug-ins are unloaded using `RifUnloadPlugins()`. These entry points are available to all programs that link with 3DELIGHT and are further described in Section 10.1.3 [Ri Filter Plug-ins], page 212.

8 Display Drivers

3Delight comes with a set of useful display drivers. Since *3Delight* uses the "standard" RenderMan display driver interface, it is possible to use third parties display drivers also. Each display driver is described with more detail in the following sections.

8.1 The idisplay display driver

This display driver redirects *3Delight*'s output to `i-display`. The `i-display` application is a standalone program designed for image sequence playback; when used with *3Delight* it enables the user to perform the following tasks:

1. **Accumulate multiple renders into a centralized application**. Different renders send their images to the same `i-display`, thus avoiding a proliferation of windows on the desktop. Additionally, many *3Delight* processes can send their images in parallel.
2. **Re-render specific images**. An image can be rendered again by using the `Reload` function. Note that the best way to use this feature is to launch the render using the `renderdl -id` command, *without* specifying an `RiDisplay` in the RIB. If using `RiDisplay` is necessary, it is mandatory to put the name of the RIB in the *filename* field:

   Display "test.rib" "idisplay" "rgba"
3. **Re-render crops of a specific image**. By holding the `CTRL` key and the left mouse button pressed and then specifying a recangular area on the screen starts a re-render of that particular screen region. Note that the RIB shouldn't contain any `RiCropWindow` statements for this feature to work correctly. Comments in (2) also apply here.
4. **Save rendered images to disk**. Images can be saved as TIFF files[1].
5. **Use color lookup tables (LUT)**. This handy feature is intended for high dynamic range output and enables color correction using *3D lookup tables*.

Other features include: zoom, color picker, realtime playback, gamma correction, full screen mode, mipmap viewer, image blending and more. The user should refer to `i-display`'s documentation for more information about its features, menus and shortcuts; it is available at `http://www.3delight.com/ZDoc/i-display.html`.

8.2 The framebuffer display driver

This display driver opens a window and displays the rendered image in it. It supports the following parameter:

`"string autoclose" "off"`
: If this parameter is set to '`on`', the window is automatically closed at the end of the render. The default behavior is to leave the window open until the user closes it.

Note that when *3Delight* finishes the render, it exits without waiting for the user to close the window opened by this display driver, letting the user start multiple renders while keeping the windows open. Of course, if `autoclose` is enabled, the window is closed automatically.

[1] They can also be saved to other formats using a user provided conversion application, refer to i-display's documentation

8.3 The TIFF display driver

This display driver writes the rendered image into a TIFF file. It is a full featured display driver that is recommended for production work. The TIFF is written *incrementally*, meaning that one could display the image at any time; this can be useful for long or aborted renders. Compressed TIFFs and cropping are supported. This display driver understands the following parameters.

`"string fullimage" ["on"]`
: When using `RiCropWindow`, this parameter enables you to create an image of the original size, with the cropped region placed inside. Areas that are outside the crop window are set to black (and alpha 0). This option is useful when compositing a cropped image with a matching complete image. The default value is '`off`'.

`"string compression" ["lzw"]`

`"lzw"` LZW compression, default.

`"none"` No compression.

`"packbits"`
: Run length encoding.

`"deflate"` Deflate compression (zip).

`"logluv"`[2] LogLuv compression, perfect for high dynamic range images. Expects floating point data as input. A limitation of this compression scheme is that the alpha channel cannot be stored in the same TIFF directory as the image, so it is written separately in a 2nd directory. The white point is also stored in the TIFF when using this compression.

`"string description" "s"`
: Specifies a description string which is stored in the TIFF.

`"string extratags" ["on"]`
: By default, this display driver writes some useful informations to the TIFF by using the tagging mechanism[3]; however, some tags are not supported by all TIFF software (See table below), this option can turn those tags to '`off`'. Tags marked with an '*' can be disabled using this option.

[2] Visit http://positron.cs.berkeley.edu/~gwlarson/pixformat/tiffluv.html for more informations about this encoding scheme.

[3] For a detailed specification of the TIFF format, refer to http://www.libtiff.org.

Tag name	Description
SOFTWARE	Is set to something like "3Delight 0.9.6 (Sep 26 2002) "Fidelio"".
IMAGEDESCRIPTION	Is set to a user provided description, if any.
XPOSITION	The x cropping position in pixels, only active when cropping and "fullimage" is "off" (default).
YPOSITION	The y cropping position in pixels, only active when cropping and "fullimage" is "off" (default).
WHITEPOINT	White point when using LOGLUV compression.
PIXAR_IMAGEFULLWIDTH*	Set when an image has been cropped out of a larger image; reflects the width of the original (non cropped) image.
PIXAR_IMAGEFULLLENGTH*	Set when an image has been cropped out of a larger image; reflects the height of the original non cropped image.
PIXAR_TEXTUREFORMAT*	Set to "Image"
PIXAR_MATRIX_WORLDTOSCREEN*	World to Normalized Device Coordinates transformation matrix.
PIXAR_MATRIX_WORLDTOCAMERA*	World to Camera transformation matrix.

Table 8.1: Tags written by the TIFF display driver.

8.4 The texture display driver

This display driver writes a texture file which is readily usable by the `texture()` shadeop. It accepts the `"mode"`, `"smode"` and `"tmode"` string parameters to set the wrap mode of the texture file to either `"black"` (the default), `"clamp"` or `"periodic"`. An example usage would be:

```
Display "file.tdl" "texture" "rgba" "string mode" ["black"]
```

Note: This display driver operates by first writing a normal TIFF file and then running `tdlmake`. This means that `tdlmake` should be accessible.

8.5 The IFF display driver

This display driver writes a IFF file that is suitable for the *Maya* software. RGB and RGBA formats are supported, in 8 bits. The only supported parameter is '`fullimage`' which is documented in Section 8.3 [dspytiff], page 169.

8.6 The zfile display driver

A `zfile` contains depth information for each pixel in the image and is meant to build a shadow map. The format of a `zfile` is described in Table 8.2.

Offset	Type	Name	Description
0	int32	magic	Typically 0x2f0867ab. Reading 0xab67082f means that swapping "endianness" in the entire file is necessary.
4	short	xres	The x resolution in pixels.
6	short	yres	The y resolution in pixels.
8	float[16]	Np	World to Normalized Device Coordinate transformation matrix, in row major order.
72	float[16]	Nl	World to Camera matrix, in row major order.
136	float[xres*yres]	z data	Top to bottom, left to right ordering.

Table 8.2: `zfile` file format.

To build a shadow map using a zfile, use `"tdlmake -shadow"` as described in Section 3.3 [Using the texture optimizer], page 15 or issue a `RiMakeShadow` command in a RIB file.

8.7 The shadowmap display driver

Use this display driver to create shadow maps without passing by a zfile. Shadow maps in *3Delight* are normal TIFFs. The following parameter can be passed to this display driver.

`"uniform string zcompression" "lzw|zip|none"`
: Enables or disables compression. Disabled by default.

`"integer aggregate" [0]`
: By default, the shadowmap display driver will overwrite the shadow map if it already exists. Setting this parameter to '`1`' will cause the display driver to *concatenate* the currently generated shadow map to the existing one. This very useful feature can create shadow maps for point lights: rendering all six views to a single shadow map file is much more usable than dealting with six different shadow maps. More about his feature in Section 7.2.4 [Aggregated Shadow Maps], page 124.

There is no need to run `tdlmake` *on files produced by this display driver*, since they are already in the right format.

8.8 The DSM display driver

This display driver produces deep shadow maps. As explained in Section 7.2 [Shadows], page 122, DSMs support opacity, prefiltering and motion blur. It is suggested to use DSMs whenever possible.

`"float tolerance" [0.02]`
: DSMs tend to grow quite large, unfortunately. The display driver tries to compress them using an algorithm that shrinks file size without exceeding some specified error threshold. A value of '`0`' (that is, no error allowed) produces large DSMs and should **never** be used and a value of '`1`' compresses DSMs too much and results are unsatisfactory. If this parameter is not specified or set to -1, the display driver computes a default value (currently always 0.02).

`"string compressionlevel" ["2"]`
: This option turns on additional compression which makes DSMs even smaller without affecting quality. `"0"` means no additional compression is not recommended. `"1"` uses compression which has no significant speed impact. `"2"` is the default and usually creates quite smaller files at the expense of slightly longer generation time.

`"string nomipmap" ["off"]`
: Disables or enables mipmapping for this deep shadow map. Mipmapping is enabled by default but one could save more than 25% of storage space by using raw DSMs. Be careful though, since mipmaped DSMs usually produce nicer results (especially when viewed from far away).

`"string volumeinterpretation" ["discrete"]`
: Specifies whether the deep shadow map is used to compute shadows cast by solid objects or those cast by participating media (fog, clouds, smoke, etc). When rendering participating media shadows, one should specify `"continuous"` to this parameter. Default is `"discrete"`.

EXAMPLE

```
# Write out a deep shadow map for participating media
Display "shadow.dsm" "dsm" "rgbaz"
    "string volumeinterpretation" ["continuous"]
```

There is not need to run `tdlmake` on files produced by this display driver since they are already in the right format. This display is limited to resolutions which are a power of two.

Particle and volume self-shadowing

Particles rendered into a deep shadow map will appear as flat objects. Thus, when viewed from a different location than the light, the half closer to the light will not be shadowed while the other half will have self-shadowing. To work around this problem, *3Delight* allows shaders to specify the thickness of a surface by outputting a `"float __thickness"` variable. This will spread the opacity computed by the shader over the specified depth around the surface. The depth should be specified in `"current"` space by the shader. For particles, using width as the thickness is usually correct:

```
surface particle_shader(
    varying float width = 0;
    output varying float __thickness = 0; )
{
    __thickness = width;
    // More shading code...
}
```

3Delight will also automatically compute a thickness for interior and atmosphere shaders. It is possible to disable that feature by having the shader output a `__thickness` of 0.

Note that this features requires that the `"volumeinterpretation"` parameter of the DSM display driver be set to `"continuous"`.

8.9 Encapsulated Postscript display driver

Use this display driver to produce `.eps` files. Those are Postcript files that can be used inside `.pdf` and `.ps` files. The alpha channel is ignored in an `.eps` file. Only one option is supported:

`"uniform string fullimage" "off"`
: When using a crop window, this parameter enables the user to create an image of the original size, with the rendered crop window placed inside. Areas that are outside the crop window are set to black. The default value is '`off`'.

8.10 Kodak Cineon display driver

This display driver writes Kodak's Cineon files. Each channel is encoded in 10 bits using a logarithmic scale (regardless of `Exposure` and `Quantize`). Such an encoding is appropriate for film recording since it is designed to cover film's color range. A color encoded using the Cineon format can be much brighter than white, therefore it is recommended to use un-quantized values for this display driver to avoid clipping output colors to (1,1,1).

The alpha channel is supported by this display driver but it is not guaranteed to be readable by other software. We suggest to use RGB (no alpha) for maximum portability.

Three parameters are recognized by this display driver:

`"uniform integer setpoints[2]" [ref_white ref_black]`
: This sets the reference black and the reference white values. The default values are 180 for reference black and 685 for reference white.

`"uniform integer ref_white"` *value*
: Only sets the reference white.

`"uniform integer ref_black"` *value*
: Only sets the reference black.

An explanation of those parameters can be found in the Cineon documentations available on the web. The standard extension for a Cineon file is `.cin`.

IMPORTANT: this display driver expects un-quantized floating point values so it is important to specify this using `RiQuantize`:

```
Quantize "rgb" 0 0 0 0
```

Or directly to the display driver as in:

```
Display "image.cin" "cineon" "rgb" "quantize" [0 0 0 0]
```

8.11 Radiance display driver

Use this display driver to write files compatible with Radiance. Floating point values are required for this display driver since it is intended as a HDRI output. The following parameter is recognized by this display driver:

`"uniform string colorspace" ["rgb"]`
: Specifies in which color space to encode the values. The two possible values are `"rgb"` and `"xyz"` and the default is `"rgb"`. Be careful when using `"xyz"` color space since it implies a RGB to XYZ conversion inside the display driver and RGB values are assumed to be **linear**, so setting `Exposure` to 1 1 is essential. Note that run-length encoding of Radiance files is not supported.

EXAMPLE

```
# Write out a radiance file, using XYZ colorspace.
Exposure 1 1
Quantize "rgb" 0 0 0 0  # use floating point data
Display "hdri.pic" "radiance" "rgb" "string colorspace" "xyz"
```

8.12 OpenEXR display driver

This display driver writes out ILM's EXR files using the '`half`' and '`float`' formats. There is no need to install any external libraries to use this display driver. Floating point input is requested for this display driver, so the RIB should contain the following lines before the display driver is invoqued:

```
Quantize "rgba" 0 0 0 0
```

The following parameters can be specified to the EXR display driver:

`"uniform string exrcompression" ["zip"]`
`"uniform string compression" ["zip"]`
Specifies which compression to use. Valid compression schemes are:

- `"none"` Disable compression.
- `"zip"` Enable '`deflate`' compression using the `zlib` library. This is the default compression scheme.
- `"zips"` Like '`zip`' but each scanline is compressed separately. Compression is not as good but the files will load faster in some software.

`"uniform string exrlineorder" ["increasing"]`
Specifies if the image is to be written in '`increasing`' or '`decreasing`' Y. Default is '`increasing`'. *When writing an image in decreasing Y order, the display driver has to buffer the entire image before writing it to the file. This could require a significant amount of temporary disk space.*

`"uniform string exrpixeltype" ["half"]`
Specifies the precision of image data in the file. Set to '`float`' for full precision, at the cost of larger files.

`"int asrgba" [0]`
When set to 1, the channels will be recorded as Y, RGB or RGBA instead of being named based on the variable being output. This only works for 1, 3 or 4 channels and is useful for compatibility with poor software.

`"int autocrop" [0]`
When set to 1, this will shrink the data window of the file to the smallest window containing all non black pixels. Since there is less data written to the file, other applications may load it faster. This option causes the data to be buffered as for exrlineorder.

`"string comment" [""]`
Attaches a comment to the produced EXR file.

`"string exrheader_attrname" [""]`
Adds a string attribute named '`attrname`' (which can be any desired name) to the file's header.

`"int exrheader_attrname" [0]`
As above but adds an integer attribute.

`"int exrheader_attrname[2]" [0 0]`
As above but adds a '`v2i`' attribute.

`"int exrheader_attrname[3]" [0 0 0]`
As above but adds a '`v3i`' attribute.

`"int exrheader_attrname[4]" [0 0 0 0]`
As above but adds a '`box2i`' attribute.

`"float exrheader_attrname" [0]`
: As above but adds a float attribute.

`"float exrheader_attrname[2]" [0 0]`
: As above but adds a '`v2f`' attribute.

`"float exrheader_attrname[3]" [0 0 0]`
: As above but adds a '`v3f`' attribute.

`"float exrheader_attrname[4]" [0 0 0 0]`
: As above but adds a '`box2f`' attribute.

`rendermn.ini` Entries

The EXR display drivers recognizes the following parameter:

`/display/exr/compression`
: Contains the default compression to use if no compression mode is passed to the display driver.

`/display/exr/pixeltype`
: Contains the default pixel type to use if none is specified to the display driver.

`/display/exr/asrgba`
: Changes the name of channels written to the file. See `"asrgba"` in the table above.

More about the `rendermn.ini` configuration file in Section 5.5 [configuration file], page 59.

8.13 The PSD display driver

This display driver writes Photoshop's PSD files. Each channel is saved within its own layer, either in RGB or in RGBA, or in GrayScale. All layers use the same format: if a layer is in GrayScale, all of them are in GrayScale; meaning that each specified channel has only one component. If only one channel has 3 or 4 components, all layers will use RGB or RGBA: channels with one component will be converted. Layers in Photoshop file format are restricted to 8 bits (unsigned) per component.

```
Declare "amb" "varying color"
Declare "diff" "varying color"
Declare "spec" "varying color"

Quantize "amb" 255 0 255 0.5
Quantize "diff" 255 0 255 0.5
Quantize "spec" 255 0 255 0.5

Display "output_multi_layer.psd" "psd" "amb,Ci,diff,alpha,spec"
```

8.14 The PIC display driver

This display driver writes SOFTIMAGE's PIC files. RGB and RGBA formats are supported, in 8 bits. The following parameters can be passed to this display driver.

`"uniform string compression" "rle|none"`
: Enables or disables compression. Disabled by default.

`"string fullimage" ["on"]`
: See explanation in Section 8.3 [dspytiff], page 169.

An example call:

```
Display "image.pic" "pic" "rgb" "string compression" "rle"
```

8.15 The JPEG display driver

This display driver output JPEG images and is able to write 1 channel gray scale and 3 channel RGB images. One parameter is accepted to control the quality of the JPEG image as outlined in the example below.

```
Display "image.jpg" "jpeg" "rgb" "int quality" [80]
```

9 Error Messages

This is a list of most of the error messages 3Delight can produce along with a brief description of possible causes for each.

A2010 `RiProcDynamicLoad: cannot find shared object '`*`libname`*`'`
Specified shared object was not found. This is usually a search path problem.

A2011 `RiProcDynamicLoad: '`*`libname`*`' is not a shared library (system error: `*`syserror`*`)`
Search paths may be in incorrect order or the file is corrupted.

A2012 `RiProcDynamicLoad: 'a' does not contain the necessary entry points (system error: `*`b`*`)`
Library is missing one of the `ConvertParameters`, `Subdivide` or `Free` entry points and cannot be used.

A2037 `RiReadArchive: recursive instantiation of 'a'`
An archive requires itself recursively. It cannot be read properly.

A2055 `RiProcRunProgram: cannot communicate with program '`*`program`*`'`
This message only appears on WINDOWS platforms and means that 3DELIGHT was unable to write a request to the specified program.

A2056 `RiProcRunProgram: error reading output of program '`*`program`*`'`
This message only appears on WINDOWS platforms and means that 3DELIGHT was unable to read the result back from the specified program.

A2057 `RiProcRunProgram: error reading output of program '`*`program`*`'. Missing request response delimiter (0xFF).`
The program procedural doesn't output the needed end-of-stream delimiter.

A2058 `RiProcRunProgram: program '`*`program`*`' has died`
program terminated ungracefully. This means that `RiProcRunProgram` was not completed correctly. It may have crashed or simply not have been written with the required loop to process more than one request.

A2059 `RiProcRunProgram: cannot launch program '`*`program`*`'`
3DELIGHT is unable to start the specified program.

A2060 `RiProcRunProgram: system returned error '`*`syserror`*`'`
3Delight was unable to run the specified program. An error string is provided as a debugging help.

A2061 `RiProcRunProgram: pipe creation failed`
On unix systems this means the the `pipe()` system call has failed. This should never happens and could mean serious system overload.

A2062 `RiProcRunProgram: cannot create child process (memory problems?)`
On unix systems this means the the `fork()` system call has failed. This should never happens and could mean serious system overload.

A2063 `RiProcRunProgram: system error in the child process`
A child process which was `fork()`ed is unable to issue a `dup2()` command. This should never happens and could mean serious system overload.

A2064 `RiProcRunProgram: cannot execute '`*`program`*`'`
On unix systems this means the the `execl()` system call has failed. The specified program may not be executable (verify permissions).

A2065 `RiProcRunProgram: cannot open reading stream for program 'program'`
3DELIGHT was not able to establish a reading connection with *program*. This should never happens and could mean serious system overload.

A2066 `RiProcRunProgram: program 'program' seems to have crashed or terminated early. It should handle multiple requests; one per line.`

A2067 `RiProcRunProgram: cannot find program 'program'`
This usually means bad search path settings.

A2084 `cannot find archive 'archivename'`
The specified archive could not be found. If it is an inline archive, it may have gone out of context. If it is a file, the search paths may be wrong.

A2085 `cannot open archive file 'archivename'`
Verify permissions.

C2046 `RiBegin not called before RiCall`
This function has to be called before any other Ri Calls.

C2047 `non closed condition block (missing RiIfEnd)`
RIB conditions are not balanced properly.

C2049 `[filename:line]: premature call to RiMotionEnd`
Some motion steps have not been described yet. An example would be:

```
MotionBegin [0 0.5 1.0]
Translate 0 0 1
    # Missing two more steps
MotionEnd
```

C2080 `RiBegin not called`
RiBegin should be the first function called when using the Ri API.

D2002 `cannot connect to display driver 'displayname' (system error: syserror)`
Some entry points are missing.

D2003 `cannot open display driver 'displayname' (system error: syserror)`
The dso could not be loaded. It could be a library problem.

D2004 `failed to open display driver 'displayname' (display driver returned 'displayerror')`
Display driver's `DspyImageOpen()` entrypoint returned a failure code. A code of 3 is usually for an unsupported combination of number of channels and format. A code of 4 usually occurs when the file cannot be created (no permission of the directory does not exist).

D2005 `display driver 'a' did not deallocate its data properly`
Display driver's `open()` returned a failure and did not free the image data.

D2006 `failed to close 'filename', display driver 'displayname' (display driver returned 'displayerror')`
Display driver's `close()` entry point returned a failure code.

D2007 `rendering aborted, display driver 'displayname' will be closed`
Rendering was interrupted, all display driver will receive the `DspyImageClose()` call.

D2008 `failed to write bucket to 'a' (display driver returned 'b')`
Display driver returned an error when attempting to write a bucket.

D2009 `query to display driver 'a' failed (display driver returned 'b')`
Display driver did not specify a image resolution. A default one is provided.

D2045 `PixelFilter reset to 'box' 1x1 (display driver requirement)`
The display driver cannot accept the specified pixel filter. Forced to 1x1. The only display driver that could do this is the '`dsm`' display driver.

D2094 `failed to open display driver 'displayname' (display driver returned with a null handle)`
Display driver's `DspyImageOpen()` entrypoint did not set the image handle.

D2202 `dspy_dsm: cannot create a temporary file (system error: message)`
The deep shadow map display driver requires some temporary files during the render. This message means it was unable to create then, most likely because of incorrect permissions on the directory it tried to use. On linux and OS X, the following directories are tried, in order: '$DL_TMPDIR', '$TMPDIR', '/tmp/' and '.'. On windows, it is '%TMP%' and '\'.

L2029 `cannot setup networking (licensing will not work)`
Licensing mechanism requires a network in order work properly.

L2030 `cannot contact license server 'servername' (system error: syserror)`
Make sure the license server is running on the computer named *servername*.

L2031 `make sure the license server is running on 'servername'`
The renderer is unable to contact the license server that has been specified in the `rendermn.ini` file. Make sure that the `rendermn.ini` file is right and that the license server (`licserver`) is running.

L2032 `please complete the 'rendermn.ini' file as explained in 'InstallationGuide.pdf'`
This means that you are running without a license and that a watermark will be displayed. Completing the `rendermn.ini` file with the `/3delight/licserver` line will solve the "problem".

L2033 `invalid license key`
Server was contacted, but license key is invalid. It may have expired or the setup is not done properly.

L2034 `all licenses are taken, waiting for a free license (delay) ...`
There are not enough licenses available to proceed with the render. 3DELIGHT will wait until a license is available. If the preferred behavior in this case is to exit immediately, please add the following line to 'rendermn.ini':

```
/3delight/waitforlicense 0
```

L2035 `cannot resolve license server 'servername'`
The computer named *servername* could not be found. This looks like an incorrect setup in the 'rendermn.ini' file.

L2036 `no more than 'numservers' license servers can be specified`
Some special licenses allow many license servers to run (for geographically separated sites for example). This error message indicates that you are trying to run more license servers than permitted.

L5034 licserver: can't connect to service control manager

L5035 licserver: service does not exist

L5036 licserver: can't open existing service

L5037 licserver: error uninstalling service

L5038 licserver: error installing service

L5039 Manages licenses for 3Delight and related products.

L5040 licserver: service uninstalled

L5041 licserver: service installed

L5059 License does not support this version of *'product'*. Use an older version or contact sales to renew support.

P1000 invalid grouping membership *'groupname'*
Format is invalid. There might be multiple '+' or '-' in the membership.

P1001 catmull-clark subdivision mesh *'objectname'*: edge between vertices *vertex1* and *vertex2* has more than two attached faces
The mesh definition is not a manifold. Some faces will be ignored.

P1002 catmull-clark subdivision mesh *'objectname'*: face has the same vertex index twice
A face is using the same vertex more than once. Some vertices will be ignored.

P1003 catmull-clark subdivision mesh *'objectname'*: vertex ID out of range
A face is using an unknown vertex. It is ignored.

P1004 catmull-clark subdivision mesh *'objectname'*: face with less than 3 vertices
A face has no area. This error may occur with more vertices if they are duplicated.

P1005 catmull-clark subdivision mesh *'objectname'*: facevertex variables are inconsistent so discontinuities may occur
The facevertex variables specified do not form well-defined zones. This can be caused by a bad mesh design or small errors in the values.

P1006 catmull-clark subdivision mesh: tag *'tagname'* has invalid number of arguments
Refer to the specification of RiSubdivisionMesh for the required number of arguments.

P1007 catmull-clark subdivision mesh: tag *'tagname'* contains an invalid vertex index *'index'*
Some vertices may be missing.

P1011 catmull-clark subdivision mesh: tag 'hole' contains an invalid face index *'faceindex'*
Some vertices may be missing.

P1012 catmull-clark subdivision mesh: unknown tag
There may be typing error in the name or it is not supported yet.

P1013 display parameter *'param'* has wrong type
The required parameter is not declared properly. It is not compatible with the display server.

P1015 `RiBlobby: truncated description`
The blobby description is missing entries.

P1016 `RiBlobby: empty description`
The blobby has no description in the RIB file

P1017 `RiBlobby: unknown field opcode` *op*
The operation *op* is not recognized. Maybe a parameter in a previous operation code was omitted.

P1018 `RiBlobby: needs at least one non-constant leaf`
All leaves are constants. There is no possible surface.

P1020 `RiBlobby: invalid operand index`
An operator parameter (operand index) is out of bounds. It may have been omitted.

P1021 `RiBlobby: invalid float index`
A float index is out of bounds. Might be invalid or there are not enough float values.

P1022 `RiBlobby: leaf requires more values`
There are not enough values for a leaf. Matrices take 16 floating point numbers for example.

P1023 `RiBlobby: less leaves than specified`
There are less leaves in the blobby than declared.

P1024 `PointsGeneralPolygons: polygon is either non planar or self-intersecting`
Unable to generate geometry for the given polygon, probably because of topological problems with the mesh.

P1025 `unexpected geometry inside transformation motion block`
A motion block cannot combine both geometry and transforms.

P1026 `more Ri calls than there are time steps in motion block`
There should be as many Ri commands as motion steps inside a motion block.

P1027 `primitives inside motion block are incompatible (object '`*obj*`')`
Both primitives must have the same general structure.

P1028 `invalid bounding box for primitive (or part of) '`*object*`'`
This happens when a bounding box is pushed behind the viewing plane because of bad displacement bounds.

P1029 `cannot find display driver '`*display*`'`
Display driver '`display`' could not be found. Possible reasons are bad search paths or misspellings.

P1030 `cannot display string variable '`*v*`'`
String variable cannot be displayed.

P1031 `unknown display variable '`*var*`' ('`*displayname*`' display driver)`
The variable to display is unknown to 3DELIGHT.

P1032 `invalid frame aspect ratio '`*aspectratio*`'`
The provided image aspect ratio is either too small (too close to zero) or negative.

P1033 `degenerate screen window`
The `RiScreenWindow` is flat on one of its axis (left=right or top=bottom)

P1034 `invalid clipping planes`
Values passed to `RiClipping` are swapped or equal (far <= near)

P1035 `degenerate crop window`
The `RiCropWindow` is flat on one of the axis (left=right or top=bottom).

P1036 `invalid crop window`
One or more values passed `RiCropWindow` is negative or greater than 1.0.

P1037 `camera to screen matrix too general, reverting to identity`
Current transformation when calling `RiProjection` is not a 2D (x,y) affine transformation.

P1038 `RI_FOV out of range, setting to 90 degrees`
Maximum field of view for `RiProjection` is 180 degrees.

P1039 `unrecognized projection '`*`projection`*`'`
Only perspective or orthographic projections are recognized by `RiProjection` (`RI_PERSPECTIVE` or `RI_ORTHOGRAPHIC` respectively).

P1040 `invalid pixel samples`
The supplied value is either negative or excessively large.

P1041 `invalid pixel filter width or height`
The given pixel width or height passed to `RiPixelFilter` filter is smaller than 1.

P1042 `invalid depth of field argument(s)`
At least one of the arguments passed to `RiDepthOfField` is less than 0.

P1043 `invalid shutter range`
Shutter close time is less than shutter open time.

P1044 `invalid exposure argument(s)`
At least one of the arguments passed to `RiExposure` is less or equal to 0.

P1045 `invalid quantize arguments`
Quantize '`max`' is less than '`max`' or dithering amplitude is less than 0.

P1046 `invalid variable '`*`variable`*`' passed to RiQuantize`
The variable specified by *variable* has not been declared. Declare it using `RiDeclare` or use an inline declaration.

P1047 `invalid depth filter 'a'`
The filter name passed to `RiHider` is invalid. More about hider filters in [hider-parameters], page 31

P1049 `unknown RiHider parameter '`*`param`*`'`
param is not a valid parameter to `RiHider`. More about `RiHider` parameters in [hider-parameters], page 31

P1051 `[`*`filename`*`:`*`line`*`]: invalid context for '`*`command`*`'`
The issued `Ri` command is not allowed in the current context. An example would be to issue a `RiOption` inside the world block.

P1052 `invalid declaration '`*`declaration`*`' for variable '`*`var`*`'`
The variable *var* is badly declared (syntax error).

P1054 `variable '`*`type var`*`' cannot be assigned to shader variable of the same name but of type '`*`type`*`' (shader is '`*`shader`*`' and primitive is '`*`prim`*`')`
There is a type mismatch between a variable attached to a primitive and its destination parameter in the shader.

P1055 `invalid parameter passed to RiOrientation: 'orientation'`
Only '`lh`' and '`rh`' are accepted as parameters to `RiOrientation` (`RI_LH` and `RI_RH` respectively).

P1056 `invalid parameter passed to RiSides: 'sides'`
`RiSides` only accepts '`1`' or '`2`' as a parameter.

P1057 `invalid parameter passed to RiGeometricApproximation: param`
The only valid parameters are '`motionfactor`' and '`focusfactor`'.

P1058 `light source with handle 'lighthandle' was already declared (the original light remains active but is no longer accessible)`
lightid is taken by another light source. Since it will be reserved for the new light source, the old one will not be accessible for subsequent `RiIlluminate` calls.

P1059 `invalid light handle 'lighthandle' passed to RiIlluminate`
lighthandle was not assigned to any light source.

P1060 `invalid argument value 'shadingrate' passed to RiShadingRate`
The given `RiShadingRate` is either 0 or negative.

P1061 `invalid argument 'interpolation' passed to RiShadingInterpolation`
`RiShadingInterpolation` only accepts '`constant`' or '`smooth`' as a parameter (`RI_CONSTANT` and `RI_SMOOTH` respectively).

P1067 `invalid argument to RiMotionBegin`
The error could only be triggered when using the `C` API. Here is an example:

```
RiMotionBegin( 0, RI_NULL );
```

P1068 `invalid polygon passed to RiPointsPolygons`
A polygon (or hole) must have at least three vertices.

P1080 `RiNuPatch (NURBS) specified with inconsistent parametric min|max values`
A *max* parametric value (either in '`u`' or '`v`') is less than a *min*.

P1081 `RiNuPatch (NURBS) specified with illegal uorder|vorder`
uorder or *vorder* parameters specified to `RiNuPatch` is less than two.

P1082 `RiNuPatch (NURBS) specified with invalid uknot vector`
A knot vector, in the '`u`' parametric direction, is not in an increasing order.

P1083 `RiNuPatch (NURBS) specified with invalid vknot vector`
A knot vector, in the '`v`' parametric direction, is not in an increasing order.

P1084 `invalid object handle passed to ObjectInstance: objecthandle`
`RiObjectInstance` was called with a handle which was not previously declared using a `RiObjectBegin`

P1096 `unknown coordinate system 'coordsys' passed to 'shadeop', using 'world' instead`
An unknown coordinate system was specified to either `occlusion()`, `indirectdiffuse()` or `gather()`. The renderer will revert to using the '`world`' space. This could be a typo or a coordinate system not being properly declared by `RiCoordinateSystem`.

P1108 `wrong number of values for 'class name type[dim]': found numfound value(s)`
This happens when the number of values in a RIB does not make sense for the type. For example, giving 7 values for "P" instead of some multiple of 3.

P1109 `required parameters not provided, required 'parmaname' is missing`
The required parameter was not provided in the RIB. This may also happen if there was not at least one complete value, for example if "P" is specified with a single float.

P1113 `unknown filter name 'filter'`
3DELIGHT knows the following filters: '`box`', '`triangle`', '`catmull-rom`', '`b-spline`', '`gaussian`', '`sinc`', '`bessel`' and '`mitchell`'.

P1117 `invalid error handler name: handler`
Valid handlers are '`print`', '`ignore`' and '`abort`'.

P1123 `unknown built-in RiProcedural 'procedural', expected 'DelayedReadArchive', 'RunProgram' or 'DynamicLoad'`
The RenderMan interface only describes `DelayedReadArchive`, `RunProgram` or `DynamicLoad` as valid procedurals for use with the Procedural RIB statement.

P1124 `invalid geometry after displacement for primitive 'object'`
This happens when a displacement shader computes invalid values (NaN or infinity). The shader must be fixed.

P1165 `undefined coordinate system 'coordsys' in shader`
The coordinate system name given to one of the transform functions in the shader was not declared.

P1170 `invalid display channel 'name'`
The name of the channel to display could not be understood. Check the declaration.

P1171 `RiIfBegin/RiElseIf: expression parse error in 'expression'`
The given expression could not be understood.

P1176 `invalid relative shading rate: 'ratemultiplier'`
The value is either 0 or negative.

P1181 `invalid subset 'groupname'`
Format is invalid. There might be multiple '`+`' or '`-`' in the subset.

P2375 `primitive is too large for RIB parser`
A primitive has exceeded an internal limitation of the RIB parser. This usually happens because the RIB is corrupted.

P2378 `invalid shading model 'shadingmodel', reverting to 'matte'`
valid shading models for photon mapping are enumerated in Section 7.4.2 [Photon Mapping], page 131

P2379 `no caustic or global photon maps declared, photon mapping will have no effect`
At least one global or caustic photon map should be declared in order to use photon mapping. Here is an example on how to do that:

```
Attribute "photon" "globalmap" ["global.ph"]
Attribute "photon" "causticmap" ["caustic.ph"]
```

More about photon mapping in Section 7.4.2 [Photon Mapping], page 131.

R0500 `unsupported transmission mode 'mode', will use 'Os'`
The specified transmission mode is not supported, the default mode will be used.

R0501 `texture 'texturename' is of an unsupported file format`
The file format is not supported directly. It should be converted using tdlmake first. See Section 3.3 [Using the texture optimizer], page 15.

R0502 `cannot output string variable 'varname' because it is of type 'type'`
Outputting string variables is not supported by 3DELIGHT.

R0504 `'ricall' is not implemented yet`
ricall API entry is not supported yet.

R0505 `normals of class 'normalclass' are not supported for ricall.`
The renderer does not accept normals of class *normalclass* for `RiCurves` and `RiNuCurves` primitives.

R0513 `motion blur of 'attribute' is not implemented`
3DELIGHT doesn't support motion blur for attributes. An example would be:

```
MotionBegin [0 1]
    Color 1 0 0
    Color 0 1 0
MotionEnd
```

R0516 `RiSubdivisionMesh: unsupported subdivision scheme 'scheme'`
Only catmull-clark subdivision surfaces are supported.

R0519 `unsupported option 'opt'`
The given `RiOption` is not supported.

R0530 `RiHierarchicalSubdivisionMesh: unsupported subdivision scheme 'scheme'`
Only catmull-clark subdivision surfaces are supported.

R0531 `unsupported hit mode 'mode'`
The specified ray hit mode is not supported.

R0532 `Camera 'cameraname' is not defined.`
The provided camera name has not been declared with `RiCamera[V]`.

R1172 `RiIfBegin/RiElseIf: unknown variable 'varname' in conditional expression`
An undefined variable was referenced during the evaluation of a conditional expression.

R1183 `parameter 'parametername' of shader 'shadername' should be made varying for best performance when ray tracing object 'objectname'`
For best performance, setting uniform shader parameters from uniform primitive variables should usually be avoided. The primitive variable should be made constant or the shader parameter should be made varying.

R2000 `object 'prim' (displacement 'shader', surface 'shader') has no displacement bound but was displaced by 'dispbound' (in eye space)`
A displacement shader was attached to an object that doesn't have any displacement bounds. This could cut off displaced parts of the object. A displacement bounds should be specified for the object (*dispbound* must be increased).

R2001 `object 'rim' (displacement 'shader', surface 'shader') exceeded its displacement bound by dispbound%`
A displacement bounds is be specified for the object but is not big enough. This could cut off displaced parts of the object. Maybe the bounds should be enlarged.

R2054 `primitive (or parts of) 'prim' discarded during eye-split`
Object is too close to the camera. Parts of it are clipped.

R2075 `cannot allocate 'size' Kbytes for dynamic array`
3Delight ios running out of memory.

R2076 `cannot create file 'filename' for baking (system error: syserror)`
3Delight was unable to create a file passed to `bake`. This can happen for many reasons and the provided system error should help with the debugging.

R2077 `trying to access element index of array 'arrayname' of size arraysize, run-time check failed (line line of file shadername)`
There was an out of bound access in the specified shader on the specified array.

R2078 `$DELIGHT environment variable is not set`
Proper working of 3DELIGHT depends on this. Please fix.

R2086 `incomplete (or invalid) parameters set for subsurface scattering`
Detected some out of bound parameters for subsurface scattering. Please refer to Section 7.4.5 [Subsurface Scattering], page 136 for guidelines.

R2093 `object 'name' (displacement 'shader', surface 'shader') used only usedbound% of its displacement bound`
The displacement bound specified for the object is much larger than needed. Performance could be increased by reducing it.

R2377 `automatic photonmaps enabled but no lights cast photons`
No lights in the scene have been enabled for photon tracing, this is required for the automatic photon mapping feature. The following option should be used to enable photon tracing for a particular light:

```
Attribute "light" "tracephotons" ["on"]
```

Further details about automatic photon mapping in Section 7.4.2 [Photon Mapping], page 131

R2385 `vtransform() or ntransform() with a projection matrix will yield unexpected results`
The vtransform and ntransform shadeops should not be used with projection matrices or with coordinate spaces which involve projections (eg. "raster" if using perspective projection). It is impossible to correctly project a vector without knowing where it is in space. The solution is to turn the vector into two points, transform those and build a new vector from the transformed points.

R5010 `renderdl: '-rifend' option missing`
A `-rifargs` option was specified to `renderdl` but not corresponding `-rifend` was found.

R5011 `renderdl: a RI filter must be specified before '-rifargs'`
A `-rifargs` option was specified to `renderdl` but not RI filter was specified using the `-rif` option.

R5012 `unable to load RI filter 'filter' (syserror)`
3DELIGHT is unable to load the specified filter. The system's error message is shown in *syserror*.

R5013 `missing 'RifPluginManufacture' in RI filter 'filter'`
3DELIGHT was able to load the specified filter but was unable to find the mandatory `RifPluginManufacture` symbol.

R5014 `missing 'RifPluginDelete' in RI filter 'filter', this causes a memory leak under Windows`
On Windows platforms, RI filters should declare the `RifPluginDelete` function to delete a specific plugin. This is needed on windows since objects should be deallocated in the DLL that did the allocation.

R5017 `procedural object 'identifier' exceeded its bounds by excedent%`
The bounds supplied to the RiProcedural call were not large enough for the objects which the procedural generated upon expansion.

R5018 `unknown or unavailable interface requested to RixContext (interfaceid)`
A shader plugin (RslPlugin) called GetRixInterface with an interface identifier which is either invalid or inappropriate for the current context (eg. k_RixLocalData in an initialization function).

R5019 `string values are not allowed in 3D texture files (see parameter '`*`parametername`*`' of bake3d()/texture3d())`
A shader tried to bake or retrieve a string value. This is often caused by a misnamed parameter.

R5027 `failed to open bake file manager service socket (`*`errormessage`*`)`
This happens when using the

R5031 `error writing to point cloud file '`*`filename`*`' (`*`errormessage`*`)`
There was an error while writing to a given point cloud file. The error returned by the system is included in the error message.

R5042 `unknown camera '`*`camera`*`'`
Any camera provided to `RiDisplay` must be declared using `RiCamera`.

R5044 `Invalid (empty) camera name specified to RiCamera`

R5045 `Token '`*`tokenname`*`' is reserved by 3Delight. Please use another name`
Some strings are reserved to *3Delight* and cannot be used to name some resources (such as cameras). Please choose another name.

R5046 `RiBlobby: can't load plugin '`*`plugin`*`' (`*`error`*`)`
You specified an invalid plug-in to `RiBlobby`. The system error is returned as part of the error message. A common error is invalid path to the plug-in.

R5047 `RiBlobby: can't find ImplicitFieldNew entry point in plugin '`*`plugin`*`' (check for correct "C" linkage)`
Every `RiBlobby` plug-in must have a `ImplicitFieldNew` entry point that is correctly exported.

R5049 `point cloud file '`*`ptcfile`*`' does not contain radiance for subsurface`
Make sure to bake the radiance channel into any point cloud file that is to be used with the `subsurface()` shadeop. Refer to Section 7.4.5.2 [Two Pass Subsurface Scattering], page 138 and [subsurface shadeop], page 104.

R5050 `Cannot run more than one `*`licens`*` free license at once (`*`errorcode`*`)`
Only one free *3Delight* instance can be run at once on some given local network. For example, running *3Delight* on a laptop which is connected to a school network could cause this error since some other user might run the free version of *3Delight* as well. The displayed error code at the end of the message can be used for troubleshooting and should be sent to us if you think that there are no good reasons to see this message.

R5051 `'importance' parameter to rayinfo() must be varying`
In an SIMD execution context, each shading sample can have a different importance. That is why it is necessary for the output of `rayinfo("importance", ...)` to be varying.

R5052 `Two different displays can't write to the same file (will keep '`*`display`*`' and ignore '`*`display`*`')`
This will happen if you are trying to write into the same file using two different display drivers. For example,

```
Display "test.tif" "tiff" "rgb"
Display "+test.tif" "png" "a"
```

R5053 `channel `*`channel`*` is out of range for texture '`*`texturename`*`' (last valid channel is `*`maxchannel`*`)`
This happens when asking for a texture channel that is out of range. For example, if the texture has only one channel, then the following code will produce an error:

```
uniform float channel = 1; /* out of range for mono textures. */
color test = ptexture( "monochromatic.ptex", channel, faceindex );
```

R5054 `filter 'badfilter' is unkown, will default to 'defaultfilter'`
A wrong fitler name was specified to a texturing function (such as texture or ptexture).

R5055 `please use ptexture shadeop to open ptexture files (ptexture is ptexturename)`
When working with ptextures, one has to use the `ptexture()` shadeop.

R5056 `invalid volume intersection strategy 'strategy' (valid tokens are "additive" and "exclusive")`

R5057 `invalid number of parameters for distribution 'distribution'`
The named distribution given to `trace()` needs a specific number of parameters. For example the '`blinn`' distribution needs one parameters which is the roughness:

```
color result = trace( P, N, "samples", 16, "distribution", "blinn", "roughness", 0.1, "wo", -I )
```

R5058 `The specified distribution 'distribution' was invalid.`

R5060 `memory limit exceeded, rendering aborted`
Rendering was interrupted because memory use exceeded the "processmemory" option.

R5062 `The specified bsdf 'bsdf' needs 'udir' direction and 'roughnessv' value.`
For anisotropic BSDFs (such as the *Ashikhmin-Shirley* BRDF), the renderer expects a vector for anisotropy direction and a second roughness value.

R5063 `The subsurface() shadeop should be specified with an "irradiance" channel for correct rendering and better performance when using global illumination.`
See Table 6.13.

R5064 `Cannot outpout deep data (including for deep shadow maps) when using ray tracer in progressive or editable mode.`
When using the '`raytrace`' hider, deep image data can only be output when `progressive` and `editable` are disabled. For example, the following `RiHider` declaration will produce this error message (won't output any deep image data to the display driver):

```
Hider "raytrace" "int progressive" [1]
Display "image.exr" "deepexr" "rgba"  # Won't work because of the above.
```

S0523 `encountered invalid operand 'operand' during the evaluation of shader 'shadername'`
The shader evaluator encountered an unknown opcode. This could happen when reading corrupted or *very* old shader files.

S2050 `cannot find shader 'shadername', will use 'default'`
3DELIGHT was unable to find the specified shader. A possible cause is wrong or unset shaders paths (refer to [Search Paths], page 39). 3DELIGHT will try loading a default shader to replace the missing one.

S2051 `cannot load shader 'shadername', will use 'default'`
3DELIGHT was unable to load the specified shader and will try loading a default shader to replace the missing one.

S2052 `shader 'shadername' is of the wrong type`
3DELIGHT detected a mismatch between the `Ri` call and shader's type (such as specifying a surface shader for a lightsource).

S2068 `shader 'shader' uses a different interface version: please recompile`
The specified shader was compiled with a different version of 3DELIGHT and should be recompiled. Note that 3DELIGHT can perform automated re-compilation if the shader was compiled with the `--embed-source` option. More about this option in [shaderdl options], page 11.

S2069 `the interface of shader 'shader' is invalid`
You have specified an invalid shader to 3Delight. Try recompile the shader from the source code.

S2071 `cannot open shader 'shader'`

S2072 `the shader 'shader' uses a different interface version. Automatic conversion was successful. It should be recompiled for better performance.`
The specified shader was compiled for an older version of *3Delight* but it has been compiled with the `--embed-source` option. This means the *3Delight* is able to recompile it on runtime. Still, re-compiling it manually once and for all is a better solution.

S2073 `'paramname' is not a parameter of shader 'shader'`
You are trying to set a parameter which is not part of the specified shader.

S2074 `cannot assign variable of type 'type' to parameter 'typeinshader varname' (shader: 'shadername')`
3Delight detected a type mismatch between a shader parameter and the value provided for the shader.

S2088 `cannot find dso shadeop 'shadeop'`
3DELIGHT is unable to find the DSO that contains the shadeop.

S2089 `cannot load dso shadeop 'shadeop' (systemerror)`
3DELIGHT is unable to load the DSO that contains the shadeop.

S2090 `cannot find 'func' function in dso shadeop 'shadeop' (may be missing extern "C")`
After loading the DSO, 3DELIGHT is unable to get the required function.

S2100 `co-shader 'handle' not found in parameter 'parametername' of shader 'shadername'`
The requested co-shader could not be found.

S2101 `bake3d(): bad value replaced by 0 in file 'bakefile'`
bake3d() was given infinite or NaN values to bake. They were replaced by 0.

S2102 `bake3d(): point with bad geometry skipped in file 'bakefile'`
bake3d() was given infinite or NaN values for a point's geometry. That point was skipped.

S2105 `cannot find RtxPlugin 'plugin'`
3DELIGHT is unable to find the DSO that contains the plugin.

S2106 `cannot load RtxPlugin 'plugin' (systemerror)`
3DELIGHT is unable to load the DSO that contains the plugin.

S2107 `cannot find 'func' function in RtxPlugin 'plugin' (may be missing extern "C")`
After loading the DSO, 3DELIGHT is unable to get the required function.

S2383 `not enough parameters for format string`
The format string given to a function such as printf or format required more arguments than were given.

S2384 `parameter type does not match format string (got` *`type`* `but expected` *`expectedtype`*`)`
The format string given to a function such as printf or format requires that the following arguments be of a certain type. The actual arguments did not match the requirement.

T0400 `'-`*`optionname`*`' option is not useful for '`*`algoname`*`' algorithm`
The option specified by *optionname* is not used for the asked *algoname* algorithm. For example, specifying the `-nscales` option to the "simple" algorithm is useless since `-nscales` is only valid for local tone mapping. More about `hdri2tif` options in Section 3.5 [Using hdri2tif], page 22.

T0425 `missing space/time coordinates for '-skymap' option ('-h' for help)`
Parameter set passed to the `skymap` option is incomplete, please refer to Section 3.3 [Using the texture optimizer], page 15.

T0426 `invalid space/time coordinates '`*`coordinate`*`' for '-skymap' option`
Parameter set passed to the `skymap` option is invalid, please refer to [tdlmake options], page 15.

T0427 `missing field of view for '-fov' option ('-h' for help)`
The `fov` option requires one scalar value as decribed in [tdlmake options], page 15.

T0430 `missing wrapping mode for '-smode', '-tmode' or '-mode' option ('-h' for help)`

T0450 `'-`*`optionname`*`' option is not useful for this texture type`
The option specified by *optionname* is not used for the asked conversion. For example, specifying the `-fov` option to a simple texture conversion is useless since `-fov` is only valid for environment maps. More about `tdlmake` options in [tdlmake options], page 15.

T0451 `output is the same as input`
Overwriting the input file is not permitted.

T0455 `tdlmake: warning, reading texture '`*`texturename`*`' may take a large`
tdlmake: amount of memory. Please refer to user's manual if you are tdlmake: unable to convert this file See [Working with Large Textures], page 20.

T0456 `tdlmake: error reading previous mipmap level`
`tdlmake` accesses the destination texture in both writing and reading when building mip map levels and this error indicates the mixing of those two operations did not succeed. We encountered such errors when textures are being created on SAMBA drives (WINDOWS platforms). The problem can be fixed by updating to a newer SAMBA device driver or creating the texture on a local drive.

T0459 `tdlmake: cannot read file '`*`texture`*`' (system error: `*`systemerror`*`)`
`tdlmake` is unable to read the file

T0460 `tdlmake: invalid zfile '`*`filename`*`'`

T0461 `missing factor or size for '-preview' option ('-h' for help)`

T0462 `size '`*`value`*`' ('-preview' option's argument) is not a valid scalar`

T0463 `'-preview' option is ignored (size '`*`value`*`' is out of range)`
An out-of-range value has been specified to the `-preview` option. See Section 3.3 [Using the texture optimizer], page 15.

T0464 `missing input gamma for '-gamma' option ('-h' for help)`

T0465 `input gamma 'value' ('-gamma' option's argument) is not a valid number`
The parameter should be a positive scalar.

T0466 `missing input gamma for '-rgbagamma' option ('-h' for help)`

T0467 `input gamma 'progname' ('-rgbagamma' option's argument) is not a valid number`
One of the four values provided to `rgbagamma` is not valid. Each of the four values should be a positive scalar.

T0468 `input gamma 'value' ('-option' option's argument) is not a positive number`

T0469 `missing color space for '-colorspace' option ('-h' for help)`

T0470 `color space 'colorspace' ('-colorspace' option's argument) was not recognized`
See Section 3.3 [Using the texture optimizer], page 15.

T2013 `netcache: cannot create cache directory 'a'`
This looks like some permission problems.

T2014 `netcache: cannot create 'directory', network caching disabled`
The network caching algorithm was unable to create a mutex file for concurrent file access. For safety reasons 3DELIGHT will disable all network caching logic.

T2015 `netcache: cache directory is located on a slow access device ('a')`
The directory is probably on the network (instead of a local drive), which defeats the purpose of a network cache. 3DELIGHT will still use the specified directory but there might be no performance gains.

T2016 `netcache: cannot compute cache size, caching disabled`
The network caching algorithm is not able to compute cache's size. For safety reasons 3DELIGHT will disable all network caching logic.

T2017 `netcache: cannot perform a locking operations to clean the cache`
At the end of a render, 3DELIGHT's network cache has to perform some maintenance tasks. This error message signifies that those tasks were not executed because 3DELIGHT is unable to get a system lock. The effect of this will be generally unnoticeable and could only result in some files being accessed on the network instead of the file cache. A possible cause is a restrictrive permission on the network cache directory.

T2018 `netcache: cannot resolve path for 'filename', this file won't be cached`
This is a low level system error. If it happens, contact us.

T2019 `netcache: cannot open lock file 'lockfile'`
3DELIGHT needs to access a lock file in the networking cache directory in order to work properly (because of potential concurrent renders). This means no caching will be performed and that 3DELIGHT will access the file(s) on the network. A possible cause is a restrictrive permission on the network cache directory.

T2020 `netcache: cached file 'file' lost but lock exist, remove lock manually (lock name is 'lockname')`
When 3DELIGHT uses a cached file, it locks it so that no other concurrent render removes the file while it is being used. This error messages signifies a problem in that a lock exists but the file doesn't. This can happen if cached files are deleted manually (which is not a good practice). To solve the problem, the lock file needs to be removed manually (removing the entire network cache directory will remove all the locks).

T2021 `netcache: lock for 'file' exists but cached file is desynchronized, will use original`

Looks like the found cached file was not supposed to be there. Refreshing it.

T2022 `netcache: invalid lock file file, content: text`

Network cache lock file is probably corrupted. This will not cause any rendering artifacts since 3DELIGHT will safely revert to the original, non-cached, file. Removing the lock file manually will fix the problem.

T2023 `netcache: system clock skewed, please fix (detected on file 'filename')`

3DELIGHT detected an invalid timestamp on some file (a timestamp in the future for example). When this happens, the renderer is not able to update the cache with new files and this could affect performance. Fixing the clock, or removing the file will fix the problem.

T2024 `netcache: cannot delete old lock file 'lockfile', please do it manually`

3DELIGHT detected a "leftover" lock file (probably because of an aborted render) and is trying to delete it but couldn't. This could be caused by restrictive file permissions and the file should be deleted manually.

T2025 `netcache: cannot create lock file 'lockfile'`

For every cached and used file, 3DELIGHT creates a corresponding lock file. This message means that no such lock file was created and 3DELIGHT will safely revert to the original, non-cached, file. A possible cause is a restrictrive permission in the network cache directory.

T2026 `netcache: 'file' is larger than maximum cache size, consider increasing cache size`

A file is too big for the cache. It is either really big or the cache size is not big enough. Cache size is adjusted through an option:

```
Option "netcache" "int cachesize" [n]
```

Where '`n`' is the size of the cache.

T2027 `netcache: cannot copy 'file' to cache`

3DELIGHT was unable to transfer to file from its network location to the cache. A possible cause is a restrictrive permission on the file or the network cache directory.

T2028 `netcache: lock file 'a' has disappeared`

A lock file was deleted too early. This is an indication that a texture file could have been removed from the cache while being read by another process.

T2038 `file 'filename' is not a valid TIFF ('error')`

The specified file is indeed a TIFF but it 3DELIGHT doesn't know how to handle it. This could happen if data encoding in the TIFF is not supported. Make sure you ran `tdlmake` on that file. Please refer to [supported formats], page 18 to see what is supported by `tdlmake`.

T2039 `file 'filename' is not a valid texture file (error)`

The file format seems invalid or corrupted.

T2040 `'filename' not found`

The specified texture or shadow file cannot be found. This usually indicates a problem with search paths.

T2041 `texture 'texturename' was not generated by 'tdlmake' (performance and/or quality will suffer)`

Textures used by 3DELIGHT work better when generated with 'tdlmake'.

T2042 `'filename' texture is not adequate for 3delight, try using 'tdlmake'`
3Delight cannot read the specified texture file directly. `tdlmake` is a utility that can convert pretty much any image file format to a readable texture so it should be tried on the input texture. See Section 3.3 [Using the texture optimizer], page 15.

T2087 `netcache: could not copy file from 'sourcefile' to 'targetfile'`
When using write caching, the output file could not be written back to the network once complete. This could be caused by a permission issue or a full target disk.

T2092 `netcache: could not update timestamp of file 'filename'`
When using write caching, the file in the cache is touched after being written back to the network. This message is shown when this fails, most likely because of a permission issue in the cache.

T2104 `netcache: 'filename' has modification time in the future, it will not be cached`
The file has an incorrect modification time. This prevents proper operation of the netcache and should be fixed.

T2372 `netcache: more than one source for output file 'filename'; only one will be cached`
This is typically caused by more than one display having the same target.

T2373 `cannot read 3D texture file 'filename' (message)`
The renderer is unable to open the specified file. The 3D texture file is either a point cloud file or a brick map file.

T2389 `error estimating light contribution for photon generation`
3DELIGHT was unable to properly spread photons between lights. Photon map quality may be suboptimal.

T5061 `Can't access directory where MARI textures are (for file mari_texture_name)`
`tdlmake` is trying to find MARI textures but is unable to list the files in the containing directory.

10 Developer's Corner

10.1 3Delight Plug-ins

10.1.1 Display Driver Plug-ins

3Delight comes with a set of standard display drivers that are suitable for most applications (see Chapter 8 [Display Drivers], page 168). However, it is possible to write custom display drivers if some specific functionality is needed. Basically, a display driver is a *dynamic shared library* (DSO or DLL in short) which implements an interface that *3Delight* understands. This interface, along with all the data types used, is described in the `$DELIGHT/include/ndspy.h` header file and is further investigated in the following sections.

10.1.1.1 Required Entry Points

A display driver must implement four mandatory entry points:

`PtDspyError DspyImageQuery` (*PtDspyImageHandle* **`image`**, *PtDspyQueryType* **`type`**, *int* **`size`**, *void* ***`data`**)
: Queries the display driver about format information.

`PtDspyError DspyImageOpen` (*PtDspyImageHandle* * **`image`**, *const char* ***`drivername`**, *const char* ***`filename`**, *int* **`width`**, *int* **`height`**, *int* **`paramcount`**, *const UserParameter* ***`parameters`**, *int* **`formatcount`**, *PtDspyDevFormat* ***`format`**, *PtFlagStuff* ***`flags`**)
: Opens a display driver.

`PtDspyError DspyImageData` (*PtDspyImageHandle* **`image`**, *int* **`xmin`**, *int* **`xmax_plus_one`**, *int* **`ymin`**, *int* **`ymax_plus_one`**, *int* **`entrysize`**, *const unsigned char* ***`data`**)
: Sends data to display driver.

`PtDspyError DspyImageClose` (*PtDspyImageHandle* **`image`**)
: Closes the display driver.

An optional entry point is also defined:

`PtDspyError DspyImageDelayClose` (*PtDspyImageHandle* **`image`**)
: Closes the display driver in a separate process. This means that renderer will exit leaving the display driver running. This is particularly useful for framebuffer-like display driver[1].

Every function is detailed in the following sections.

DspyImageQuery

This function is called for two reasons:

1. *3Delight* needs to know the default resolution of the display driver. This may happen if the user did not call `Format`.
2. *3Delight* needs to know whether the display driver overwrites or not the specified file (not used at the moment).

Parameters are:

type Can take one the following values:

[1] `i-display` defines this function call

`PkSizeQuery`
: Queries a default size for the image.

`PkCookedQuery`
: Queries whatever the display driver wants "cooked" (pixels) or "raw" (*fragment list*) data. Refer to Section 10.1.1.3 [Accessing 3Delight's Deep Buffer], page 199 for more information about fragment lists. If this query is not handled, the rendered assumes "cooked" data.

`PkOverwriteQuery`
: Unsupported.

For each query, a different data structure needs to be filled. The structures are declared in `$DELIGHT/include/ndspy.h`. A brief description follows:

size Maximum size of the structure to fill.

data A pointer to the data to fill. Copy the appropriate structure here.

See Section 10.1.1.4 [display driver example], page 202.

DspyImageOpen

Called before rendering starts. It is time for the display driver to initialize data, open file(s), ... Here is a description of all the parameters passed to this function.

image This opaque pointer is not used in any way by *3Delight*. It should be allocated and used by the display driver to pass information to `DspyImageData` and `DspyImageClose`. For instance, a TIFF display driver would put some useful information about the TIFF during `DspyImageOpen` so that `DspyImageData` could access the opened file.

drivername Gives the device driver name as specified by `Display`. For example:

```
Display "super_render" "framebuffer" "rgb"
```

provides `framebuffer` in *drivername*.

filename Gives the filename provided in the `Display` command. For example:

```
Display "render.tif" "tiff" "rgb"
```

provides `render.tif` in *filename*.

width
height Gives the resolution of the image, in pixels. If the image is cropped, *width* and *height* reflect the size of the *cropped* window.

paramcount
: Indicates total number of user parameters provided in this call.

UserParameter
: An array of user parameters, of size *paramcount*. `UserParameter` is defined as:

```
typedef struct
{
    const char *name;
    char valueType, valueCount;
    const void *value;
    int nbytes;
} UserParameter;
```

name is the name of the parameter, *valueType* is its type, which can be one of the following: '`i`' for an integer type, '`f`' for an IEEE floating point type, '`s`' for a string type and '`p`' for a pointer type. *valueCount* is used for parameters that have more than

one value, such as matrices and arrays. *value* is the pointer to the actual data, except for the 'p' type where it is the data itself. A set of standard parameters is always provided; those are described in Table 10.1.

formatcount
Number of output channels.

formats An array of channel descriptions of size *formatcount*. A channel description contains a name and a type:

```
typedef struct
{
    char *name;
    unsigned type;
} PtDspyDevFormat;
```

The *type* can take one of the following values and is modifiable by the display driver:

```
PkDspyFloat32
PkDspyUnsigned32
PkDspySigned32
PkDspyUnsigned16
PkDspySigned16
PkDspyUnsigned8
PkDspySigned8
```

Additionally, the display driver may choose the byte ordering of the data by "or-ing" the type with one of the following two values:

```
PkDspyByteOrderHiLo
PkDspyByteOrderLoHi
```

flags Modifiable flags to control the way data is sent to the display driver. This flag can be set to `PkDspyFlagsWantsScanLineOrder` to receive the data per scanline instead of per bucket.

Parameters can be passed to a display driver when issuing the `Display` command:

```
Display "render" "my_display" "rgb" "string compression" "zip"
```

In this case, `my_display` driver receives the parameter "compression".

Name	Type	Count	Comments
NP	'f'	16	World to Normalized Device Coordinates (NDC) transform
Nl	'f'	16	World => Camera transform
near	'f'	1	Near clipping plane, as declared by `Clipping`
far	'f'	1	Far clipping plane, as declared by `Clipping`
nthreads	'i'	1	Number of rendering threads
origin	'i'	2	Crop window origin in the image, in pixels
OriginalSize	'i'	2	Since *width* and *height* provided to `DspyImageOpen` only reflect the size of the cropped window, this variable gives the original, non cropped window size
PixelAspectRatio	'f'	1	Pixel aspect ratio as given by `Format`
Software	's'	1	Name of the rendering software: "3Delight"
errorhandler	'p'	-	A pointer to the standard error handler, of type `RtErrorHandler`. The display driver can use this function to print messages through 3DELIGHT's error handler[2], as declared by `RiErrorHandler`. Refer to Section 10.1.1.4 [display driver example], page 202.
3dl_quantize_info	'f'	5*fc	Quantize information for each of the `formatcount` channels. Tuples of 5 floats for zero, one, min, max and dither. Comes from either RiQuantize or the "quantize" and "dither" display parameters.

Table 10.1: Standard parameters passed to `DspyImageOpen()`.

DspyImageData

3Delight calls this function when enough pixels are available for output. Most of the time, this happens after each rendered bucket. However, if `DspyImageOpen` asks for scanline data ordering, a call to this function is issued when a *row* of buckets is rendered.

image Opaque data allocated in `DspyImageOpen`

xmin
xmax_plus_one
ymin
ymax_plus_one
Screen coordinates containing provided pixels data.

entrysize Size, in bytes, of one pixel. For example, if eight bit RGBA data was asked, entrysize is set to four.

data Pointer to the actual data, organized in row major order.

DspyImageClose

Called at the end of each rendered frame. It is time to free all resources that were used and free *image* (which was allocated by `DspyImageOpen`).

DspyImageDelayClose

If this entry point is defined in the display driver, it is called instead of `DspyImageClose()` with the difference being that the call occurs in a separate process so that *3Delight* can exit without waiting for the display driver. The `framebuffer` display driver uses this functionality (see Section 8.2

[2] This is better than using `printf()` or the like.

[framebuffer], page 168). This is not supported on Windows so it should be avoided whenever possible.

10.1.1.2 Utility Functions

IMPORTANT: The following functions are useful to search through the parameters list and to perform some other low-level tasks. Note that there is a slightly annoying side-effect: it becomes mandatory to link with `3delight.dll` on the Windows operating system see Section 10.3 [linking with 3delight], page 244. This means that display drivers that use these functions are not directly usable with other RenderMan-compliant renderers and will have to be re-compiled for them (although with no other modfications).

`PtDspyError DspyFindStringInParamList` (*const char* ***name**, *char* ****result**, *int* ***parameters_count**, *const UserParameter* ***parameters**)
`PtDspyError DspyFindFloatInParamList` (*const char* ***name**, *float* ***result**, *int* ***parameters_count**, *const UserParameter* ***parameters**)
`PtDspyError DspyFindFloatsInParamList` (*const char* ***name**, *int* ***result_count**, *float* ***result**, *int* ***parameters_count**, *const UserParameter* ***parameters**)
`PtDspyError DspyFindIntInParamList` (*const char* ***name**, *float* ***result**, *int* ***parameters_count**, *const UserParameter* ***parameters**)
`PtDspyError DspyFindIntsInParamList` (*const char* ***name**, *int* ***result_count**, *float* ***result**, *int* ***parameters_count**, *const UserParameter* ***parameters**)
`PtDspyError DspyFindMatrixInParamList` (*const char* ***name**, *float* ***result**, *int* ***parameters_count**, *const UserParameter* ***pthe arameters**)

All these functions search a parameter of a specific type and size in the provided parameters list (as specified by `DspyImageOpen`, refer to [DspyImageOpen], page 195). If the desired parameter is found and the type matches, these functions will fill *result* with the found values and return `PkDspyErrorNone`. So for example, to find a `float gamma[3]` parameter, one could issue:

```
float display_gamma[3];
float array_size;
PtDspyError result = DspyFindFloatInParamList(
    "gamma",
    &array_size, &display_gamme, /* the ouputs */
    paramCount, parameters);
if( result == PkDspyErrorNone )
{
    /* can read 'array_size' floats in 'display_gamme'. */
    ...
}
```

Notes:

- Colors, points, normals and vectors are considered as arrays of floats.
- Integers are converted to floating points.
- Both user provided parameters and standard parameters (see Table 10.1) are accesible with these functions.

`PtDspyError DspyReorderFormatting` (*int* ***format_count**, *PtDspyDevFormat* ***format**, *int* ***out_format_count**, *const PtDspyDevFormat* ***out_format**)

Reorders the format array in a specific order. Parameters:

format_count
format This is the format channels description as provided by [DspyImageOpen], page 195. It contains all the channels provided to this display driver.

out_format_count
The size of the *out_format* character array.

out_format The desired ordering. For example: "bgra".

If a display driver wants the channels in an "bgra" ordering it can call this function as follows:

```
/* the formats will be re-ordered in-place. */
DspyReorderFormatting( format_count, formats, 4, "bgra" );
```

`void DspyMemReverseCopy` (*unsigned char* ***target***, *const unsigned char* ***source***, *int* ***len***)
Reverse *len* bytes, from *source* to *target*.

`void DspyError` (*const char* ***module***, *const char* ***fmt***)
Output an error message using the specified formatting. For example:

```
DspyError( "dspy_tiff", "cannot open file '%s'\n", file_name );
```

`PtDspyError DspyRegisterDriverTable` (*const char* ***driver_name***, *const PtDspyDriverFunctionTable* ***pTable***);
This function allows a new display driver to be registered by an application by directly supplying function pointers to the required entry points. This is useful when 3Delight is used as a library and the output is to be displayed in the host application. For example, in 3Delight for Maya, we have:

```
PtDspyDriverFunctionTable table;
memset( &table, 0, sizeof(table) );

table.Version = k_PtDriverCurrentVersion;
table.pOpen = &MayaDspyImageOpen;
table.pQuery = &MayaDspyImageQuery;
table.pWrite = &MayaDspyImageData;
table.pClose = &MayaDspyImageClose;

DspyRegisterDriverTable( "maya_render_view", &table );
```

Where MayaDspyImageOpen, etc are the same functions which would be used for a standalone display driver.

10.1.1.3 Accessing 3Delight's Deep Buffer

Contrary to a standard framebuffer -which stores the final color at each pixel- a *deep buffer* stores a *list* of surfaces touching a specific pixel. *3Delight* provides access to such a *buffer* with a notable difference: surface lists are provided *per sub-sample* and not per-pixel. This means that the user has access to the raw, unfiltered, surface lists directly from the render engine. The surface lists, which are called *fragment lists* in *3Delight*, can be accessed using a display driver. By default, all display drivers receive pixels and not fragment lists. To enable fragment lists one has to reply consequently to the `PkCookedQuery` query, as described in [DspyImageQuery], page 194. If this is done, the display driver will receive the requested lists of fragments instead of a buffer of pixels. The fragments are received through a `PtDspyRawData` structure. The structure is shown in Listing 10.1.

```
typedef struct
{
    char ID[4];

    /* Samples per pixel in x & y */
    unsigned sppx, sppy;

    /* Data width & height in pixels */
    int width, height;

    /* Array of fragment lists. NumLists = width*height*sppx*sppy */
    PtDspyFragment **fragments;

} PtDspyRawData;
```

Listing 10.1: PtDspyRawData structure description.

Listing 10.2 illustrates how to get a pointer to the structure and then traverse the fragments.

```
PtDspyError DspyImageData(
    PtDspyImageHandle i_hImage,
    int i_xmin, int i_xmax_plusone,
    int i_ymin, int i_ymax_plusone,
    int i_entrySize,
    const unsigned char *i_data )
{
    if( !i_data )
        return PkDspyErrorBadParams;

    const PtDspyRawData *rawData = (const PtDspyRawData *)i_data;

    if (strcmp(rawData->ID, "3DL"))
        return PkDspyErrorBadParams;

    unsigned numFrags = 0;

    /* fpbx = fragments per bucket in x.
       fpby = fragments per bucket in y. */
    unsigned fpbx = rawData->sppx * rawData->width;
    unsigned fpby = rawData->sppy * rawData->height;

    unsigned numFrags = 0;

    for( unsigned i = 0; i < fpby*fpbx; i++ )
    {
        PtDspyFragment *f = rawData->fragments[ i ];

        /* we'll skip fragments with a depth of FLT_MAX */
        while( f && f->depth==FLT_MAX )
            f = f->next;

        while( f )
        {
            numFrags++;
            f = f->next;
        }
    }

    printf( "total number of fragments = %d\n", numFrags );
}
```

Listing 10.2: Accessing lists of fragments (deep buffer) in display drivers

A fragment has the following C declaration:

```
typedef struct PtDspyFragment_s
{
    /* Fragment color & opacity.  */
    float *color;

    /* "Thickness" of this sample. */
    union
    {
        float thickness;
        float *filler;
    };

    /* depth of this fragment */
    float depth;

    /* u/v */
    float u, v;

    /* fragment opacity */
    float opacity[3];

    /* Next fragment on the list (in the same subsample) */
    struct PtDspyFragment_s *next;

} PtDspyFragment;
```

Listing 10.3: `PtDspyFragment` structure.

Follows some important remarks about fragment lists:

1. The lists are sorted in Z. Furthest fragments come first in the least.
2. Lists are truncated at first opaque object unless special culling attributes are set to disable hidden surface removal (see [culling attributes], page 46). In general, lists longer than one will contain fragments that are not opaque.
3. The length of each list is not constant per sample, of course.
4. In order to obtain the final color for a pixel, the user must composite the fragments and then filter them.

10.1.1.4 A Complete Example

```
/*
    Copyright (c)The 3Delight Team.
    All Rights Reserved.
*/

//
// = LIBRARY
//     3delight
// = AUTHOR(S)
//     Aghiles Kheffache
// = VERSION
//     $Revision$
// = DATE RELEASED
//     $Date$
// = RCSID
```

```
//      $Id$
//

#include <ndspy.h>
#include <uparam.h>
#include <ri.h>

#include <assert.h>
#include <stdio.h>
#include <float.h>
#include <string.h>
#include <limits.h>

/* ZFile Display Driver Implementation */

const unsigned kDefaultZFileSize = 512;

static void stderr_error( int type, int severity, char *msg )
{
         /* just ignore type and severity.  */
         fprintf( stderr, "%s", msg );
}

/*
zfile format:
    zFile format is (matrices and image are row-major):
    magic # (0x2f0867ab)                                        (4 bytes)
    width (short)                                               (2 bytes)
    height (short)                                              (2 bytes)
    shadow matrices (32 floats, 16 for NP and 16 for Nl)  (128 bytes)
    image data (floats)                          (width*height*4 bytes)

    NOTE
    Matrices are stored in row major format.
*/
class zFile
{
public:
    zFile(
        const char* fileName,
        const float* np, const float* nl,
        unsigned short width, unsigned short height )

    :  m_file(0x0), m_width(width), m_height(height),
    m_currentLine(0), m_pixelsLeftOnLine(width)
    {
        m_file = fopen( fileName, "wb" );

        if( m_file )
        {
            unsigned magic = 0x2f0867ab;
```

```
            assert( sizeof(magic) == 4 );

            fwrite( &magic, 4, 1, m_file );
            fwrite( &m_width, sizeof(m_width), 1, m_file );
            fwrite( &m_height, sizeof(m_height), 1, m_file );
            fwrite( np, sizeof(float), 16, m_file );
            fwrite( nl, sizeof(float), 16, m_file );
        }
    }

    ~zFile()
    {
        if( m_file )
        {
            fclose(m_file);
        }
    }

    bool Valid() const { return m_file != 0x0; }

    unsigned GetWidth() const {return m_width;}
    unsigned GetHeight() const {return m_height;}

    bool WriteScanline(
        unsigned short y, unsigned short size, const float* data )
    {
        if( y != m_currentLine || size > m_pixelsLeftOnLine )
        {
            return false;
        }

        m_pixelsLeftOnLine -= size;

        if( m_pixelsLeftOnLine == 0 )
        {
            ++m_currentLine;
            m_pixelsLeftOnLine = m_width;
        }

        return fwrite( data, sizeof(float), size, m_file ) == size;
    }

private:
    FILE* m_file;
    unsigned short m_width;
    unsigned short m_height;

    unsigned short m_currentLine;
    unsigned short m_pixelsLeftOnLine;
};
```

```
/*
    A utility function to get user parameters ...
*/
const void* GetParameter(
    const char *name,
    unsigned n,
    const UserParameter parms[] )
{
    for( unsigned i=0; i<n; i++ )
    {
        if(0 == strcmp(name, parms[i].name))
        {
            return parms[i].value;
        }
    }
    return 0x0;
}

/*
    Open
*/
PtDspyError DspyImageOpen(
    PtDspyImageHandle *i_phImage,
    const char *i_drivername,
    const char *i_filename,
    int i_width,  int i_height,
    int i_parametercount,
    const UserParameter i_parameters[],
    int i_numFormat,
    PtDspyDevFormat i_format[],
    PtFlagStuff *flagstuff )
{
    int i;
    bool zfound = false;

    const float* nl =
        (float*)GetParameter( "Nl", i_parametercount, i_parameters );

    const float* np =
        (float*)GetParameter( "NP", i_parametercount, i_parameters );

         RtErrorHandler error_handler =
         (RtErrorHandler)GetParameter(
                          "errorhandler", i_parametercount, i_parameters );

         if( !error_handler )
         {
                  /* could happen if display driver is not run from 3Delight.  */
                  error_handler = &stderr_error;
         }
```

```
/* Loop through all provided data channels and only ask for the 'z'
   channel.  */

for( i=0; i<i_numFormat; i++ )
{
    if( strcmp(i_format[i].name, "z") != 0 )
    {
        i_format[i].type = PkDspyNone;
    }
    else
    {
        i_format[i].type = PkDspyFloat32;
        zfound = true;
    }
}

if( !zfound )
{
             /* An example on how to call the error message handler.  */
             (*error_handler)(
                      RIE_CONSISTENCY, RIE_ERROR, "dspy_z :  need 'z' in order to proceed.\n" );
    return PkDspyErrorUnsupported;
}

if( !nl || !np )
{
    (*error_handler)( RIE_CONSISTENCY, RIE_ERROR,
        "dspy_z :  need Nl & Np matrices in order to proceed.  bug.\n" );
    return PkDspyErrorBadParams;
}

if (i_width > USHRT_MAX || i_height > USHRT_MAX)
{
    (*error_handler)(
        RIE_LIMIT, RIE_ERROR,
        "dspy_z :  image too large for zfile format" \
        " (use shadowmap display driver).\n" );
    return PkDspyErrorUndefined;
}

zFile* aZFile = new zFile( i_filename, np, nl, i_width, i_height );

if( !aZFile || !aZFile->Valid() )
{
             (*error_handler)
                      ( RIE_SYSTEM, RIE_ERROR, "dspy_z :  cannot create file.\n" );

    delete aZFile;
    return PkDspyErrorNoResource;
}
```

```
    *i_phImage = (void*) aZFile;

    /* Ask display manager to provide data scanline by scanline
    */
    flagstuff->flags |= PkDspyFlagsWantsScanLineOrder;

    return PkDspyErrorNone;
}

/*
    DspyImageQuery
*/
PtDspyError DspyImageQuery(
    PtDspyImageHandle i_hImage,
    PtDspyQueryType i_type,
    int i_datalen,
    void *i_data )
{
    zFile *aZFile = (zFile*) i_hImage;

    if( !i_data )
    {
        return PkDspyErrorBadParams;
    }

    size_t datalen = i_datalen;
    switch( i_type )
    {
        case PkSizeQuery:
        {
            PtDspySizeInfo sizeQ;

            if( aZFile )
            {
                sizeQ.width = aZFile->GetWidth();
                sizeQ.height = aZFile->GetHeight();
                sizeQ.aspectRatio = 1;
            }
            else
            {
                sizeQ.width = kDefaultZFileSize;
                sizeQ.height = kDefaultZFileSize;
                sizeQ.aspectRatio = 1;
            }

            memcpy(
                i_data, &sizeQ,
                datalen>sizeof(sizeQ) ? sizeof(sizeQ) : datalen );

            break;
```

```
        }

        case PkOverwriteQuery:
        {
            PtDspyOverwriteInfo overwQ;

            overwQ.overwrite = 1;

            memcpy(
                i_data, &overwQ,
                datalen>sizeof(overwQ) ? sizeof(overwQ) : datalen );

            break;
        }

        default:
            return PkDspyErrorUnsupported;
    }

    return PkDspyErrorNone;
}


/*
    DspyImageData

    Data is expected in scanline order (as asked in DspyImageOpen()).
*/
PtDspyError DspyImageData(
    PtDspyImageHandle i_hImage,
    int i_xmin, int i_xmax_plusone,
    int i_ymin, int i_ymax_plusone,
    int i_entrySize,
    const unsigned char* i_data )
{
    zFile* aZFile = (zFile*) i_hImage;
    const float* fdata = (const float*) i_data;

    if( !aZFile || !fdata )
    {
        return PkDspyErrorBadParams;
    }

    /* Perform some sanity checks but everything should be fine really ...
       :> */

    if( i_ymax_plusone - i_ymin > 1 ||
        i_xmin != 0 ||
        i_xmax_plusone != aZFile->GetWidth() ||
        i_entrySize != sizeof(float) )
    {
```

```
        return PkDspyErrorBadParams;
    }

    if( !aZFile->WriteScanline(i_ymin, i_xmax_plusone - i_xmin, fdata) )
    {
        return PkDspyErrorNoResource;
    }

    return PkDspyErrorNone;
}

/*
    DspyImageClose

    delete our object.
*/
PtDspyError DspyImageClose( PtDspyImageHandle i_hImage )
{
    zFile* aZFile = (zFile*) i_hImage;

    if( !aZFile )
    {
        return PkDspyErrorUndefined;
    }

    delete aZFile;

    return PkDspyErrorNone;
}
```

10.1.1.5 Compilation Directives

Here is the compilation command line for the given example (`zfile.cpp`):

Linux `g++ -shared -o zfile.so -I$DELIGHT/include zfile.cpp`

IRIX `CC -shared -o zfile.so -I$DELIGHT/include zfile.cpp`

MacOS X `g++ -dynamiclib -o zfile.so -I$DELIGHT/include -arch "ppc" zfile.cpp`

Windows `cl -I"%DELIGHT%/include" -LD zfile.cpp`

10.1.2 RSL Plug-ins

It is possible to extend the capabilities of the shading language by calling C or C++ functions from inside shaders. When compiling a shader, if the compiler encounters a function it doesn't now, it automatically searches all the directories specified by the `-I` command line option (see Section 3.2 [Using the shader compiler], page 11) looking for a DSO containing a definition of the unknown function.

All this is better explained by an example:

```
/*
        A sample rsl plugin with two functions:  one to square float values and
        another to square point values.
```

```
*/
#include "RslPlugin.h"

#include <iostream>

int float_sqr( RslContext* rslContext, int argc, const RslArg* argv[] )
{
        RslFloatIter retArg( argv[0] );
        RslFloatIter arg1( argv[1] );

        unsigned numVals = RslArg::NumValues( argc, argv );
        for( unsigned i = 0; i < numVals; ++i)
        {
                *retArg = *arg1 * *arg1;
                ++retArg;
                ++arg1;
    }

    return 0;
}

int point_sqr( RslContext* rslContext, int argc, const RslArg* argv[] )
{
        RslPointIter retArg( argv[0] );
        RslPointIter arg1( argv[1] );

        unsigned numVals = RslArg::NumValues( argc, argv );
        for( unsigned i = 0; i < numVals; ++i)
        {
                (*retArg)[0] = (*arg1)[0] * (*arg1)[0];
                (*retArg)[1] = (*arg1)[1] * (*arg1)[1];
                (*retArg)[2] = (*arg1)[2] * (*arg1)[2];
                ++retArg;
                ++arg1;
    }

    return 0;
}

void plugin_init( RixContext *context )
{
        /*
                This is an initialization function for the entire plugin.  shadeop
                specific functions can also be specified in the pluginFunctions array
                below.
        */
}

void plugin_cleanup( RixContext *context )
{
        /*
```

```
                This is a cleanup function for the entire plugin.  shadeop specific
                functions can also be specified in the pluginFunctions array below.
        */
}

extern "C"
{
        static RslFunction pluginFunctions[] =
        {
                { "float sqr(float)", float_sqr, NULL, NULL },
                { "point sqr(point)", point_sqr, NULL, NULL },
                NULL
        };

        RSLEXPORT RslFunctionTable RslPublicFunctions(
                        pluginFunctions, plugin_init, plugin_cleanup );
}
```

Here is how to compile a DSO under different environments:

Linux `g++ -shared -o sqr.so -I$DELIGHT/include sqr.cpp`

IRIX `CC -shared -o sqr.so -I$DELIGHT/include sqr.cpp`

MacOS X `g++ -dynamiclib -o sqr.so -I$DELIGHT/include sqr.cpp`

Windows `cl -I"%DELIGHT%/include" -LD sqr.cpp`

When dealing with string parameters, *3Delight* provides the plug-in with pointers which are valid at least until the plug-in is unloaded. For output parameters, the plug-in should assign strings obtained from the `RixTokenStorage` interface. Here is an example which illustrates how to do this:

```
/*
        Sample plugin which concatenates two strings.
*/
#include "RslPlugin.h"

#include <stdlib.h>
#include <string.h>

int concat( RslContext* rslContext, int argc, const RslArg* argv[] )
{
        RslStringIter retArg( argv[0] );
        RslStringIter arg1( argv[1] );
        RslStringIter arg2( argv[2] );

        RixTokenStorage *token_storage =
                static_cast<RixTokenStorage*>(
                        rslContext->GetRixInterface( k_RixGlobalTokenData ) );

        unsigned numVals = RslArg::NumValues( argc, argv );
        for( unsigned i = 0; i < numVals; ++i)
        {
```

```
                size_t l1 = strlen( *arg1 );
                size_t l2 = strlen( *arg2 );
                /* NOTE: dynamic allocation may easily cause horrible performance.  */
                char *buffer = (char*) malloc( l1 + l2 + 1 );
                memcpy( buffer, *arg1, l1 );
                memcpy( buffer + l1, *arg2, l2 );
                buffer[l1 + l2] = '\0';

                /* This provides a copy of the string which will live long enough.  */
                *retArg = token_storage->GetToken( buffer );

                free( buffer );

                ++retArg;
                ++arg1;
                ++arg2;
    }

    return 0;
}

extern "C"
{
        static RslFunction pluginFunctions[] =
        {
                { "string external_concat(string,string)", concat, NULL, NULL },
                NULL
        };

        RSLEXPORT RslFunctionTable RslPublicFunctions(
                        pluginFunctions, NULL, NULL );
}
```

10.1.3 Ri Filter Plug-ins

All Rif API function and class definitions are defined in `$DELIGHT/include/rif.h`, so this file must be `#included`. API entry points are described in the following sections.

Plug-in Entry Points

The following functions should be defined in all Ri plug-in filters.

`RifPlugin* RifPluginManufacture` (*int* `argc`, *char* `**argv`)

This function is called after the plug-in is loaded. It should return an instance of the Ri plug-in filter. An example is provided in Listing 7.17. All parameters are passed using *i_argc* and *i_Argv* as for the `main()` in C. The `RiPlugin` abstract class is defined in `$DELIGHT/include/rif.h` and all plug-in filters must derive from it:

```
class RifPlugin
{
public:
    virtual ~RifPlugin() {}
    virtual RifFilter &GetFilter() = 0;
};
```

The `RifFilter` structure returned by `RifPlugin::GetFilter()` contains all the filtering information (refer to Section 7.17 [Using Ri Plug-in Filters], page 162 for usage examples):

```
struct RifFilter
{
    enum { k_UnknownVersion = 0, k_CurrentVersion = 1 };
    enum { k_Continue = 1, k_Terminate = 2 } DefaultFiltering;
    int Filtering;
    short Version;
    void *ClientData;
    char Reserved[64];
    RifFilter();
    /* Transforms */
    RtVoid (*Perspective)(RtFloat fov);
    RtVoid (*ConcatTransform)(RtMatrix transform);
    RtVoid (*CoordinateSystem)(RtToken space);
    RtVoid (*ScopedCoordinateSystem)(RtToken space);
    RtVoid (*CoordSysTransform)(RtToken space);
    RtVoid (*Identity)(void);
    ...
};
```

`RtVoid RifPluginDelete ( *RifPlugin** i_plugin )`
: It is recommended to have this entry point in the DSO, since 3DELIGHT will use it to destroy DSOs loaded using `RifLoadPlugin()` (see below). This function has been defined for Windows systems, were memory allocation and deallocation cannot span multiple DLLs. The Plug-in will still work without this entry point but will leak memory on Windows systems.

Renderer's Entry Points

The following functions are implemented in 3DELIGHT and are accessible from Ri plug-ins.

`RifPlugin* RifLoadPlugin ( const char *i_name, int i_argc, char **argv )`
: Loads the Ri plug-in specified named *i_name* and add it at the end of the plug-in chain. It is necessary to call `RifUnloadPlugins()` to free all allocated resources at the end of render.

`RtVoid RifUnloadPlugins ( void )`
: Free all allocated resources and close all DSOs that have been loaded by `RifLoadPlugin()`.

`RtVoid RifAddPlugin ( RifPlugin *i_plugin )`
: Add a plug-in at the end of the plug-in chain. The user is responsible of the actual loading and resource management tasks. It is suggested to use `RifLoadPlugin()` instead.

`RtVoid RifRemovePlugin ( RifPlugin *i_plugin )`
: Removes a plug-in from the plug-in chain. The user is responsible for releasing resources used by the plug-in.

`RifPlugin* RifGetCurrentPlugin ( )`
: Returns the currently active plug-in. This function is important since there is no way, for the plug-in, on which particular instance it is running (because all callbacks are *static*).

`RifEmbedding RifGetEmbedding ( )`
: Says whether this plug-in is run in RIB output mode or in normal rendering mode. For example, when executing `renderdl` with the `-catrib` option, plug-ins are in RIB output

mode. A particular plug-in might choose to act differently depending on the "embedding". `RifEmbedding` is defined as follows:

```
typedef enum
{
    k_RifRIB,
    k_RifRenderer
} RifEmbedding;
```

`RtInt RifGetDeclaration` (*RtToken* `i_token`, *RifTokenType* *`i_token`, *RifTokenDetail* *`i_detail`, *RtInt* *`o_array_len`)

A helper function to parse Ri in-line declarations.

`RtString RifGetTokenName` (*RtToken* `i_token`)

Another helper function to parse in-line token declarations. This one returns the name of the token or null if the declaration is invalid. The returned name does not need to be freed.

`RtVoid RifGetChainInfo` (*RtInt* *`o_current`, *RtInt* *`o_total`)

This procedure returns the current level in the Ri plug-in filter chain as well as the total number of plug-ins in the chain. Range of the *o_current* variable is [0 ... *o_total*-1]

`RtVoid RifParseFile` (*const char* *`i_filename`, *RifParseMode* `i_mode`)

This procedure enables a Ri plug-in to easily parse RIB files. The *i_mode* variables tells 3DELIGHT where, in the Ri plug-in filter chain, the RIB stream should be inserted:

`k_RifParseNextLayer`
: Inject RIB stream into next plug-in filter in the chain.

`k_RifParseThisLayer`
: Inject RIB stream back into this layer in the chain.

`k_RifParseFirstLayer`
: Inject RIB stream into first layer in the chain.

The declaration of `RifParseMode` is found in `$DELIGHT/include/rif.h`:

```
typedef enum
{
    k_RifParseNextLayer,
    k_RifParseThisLayer,
    k_RifParseFirstlayer
} RifParseMode;
```

`RtVoid RifParseBuffer` (*const char *i_buf, unsigned i_size, RifParseMode i_mode*)

Same as `RifParseFile` but parse RIB commands from memory.

10.1.4 Procedural Geometry Plug-ins

This section is meant to provide examples on how to write simple procedural primitives for *3Delight*. For more complete documentation about procedural primitives and the interface to them, refer to the RenderMan Interface Specification.

10.1.4.1 The RunProgram Procedural Primitive

This example shows how to write a procedural primitive in the form of an external program which generates RIB. It can be invoked in a RIB stream as such:

```
Procedural "RunProgram" ["sphere" "0 0.5 1"] [-1 1 -1 1 -1 1]
```

The three floats it receives as parameters represent a color, as expected by the program. The program itself outputs a sphere of that color. This example also shows how to generate a RIB using

`lib3delight`. Note that on Windows, *3Delight* searches for `sphere.exe` if it doesn't find `sphere` in its procedural search path.

Note that a poorly written program (especially one which fails to output the \377 delimiter) may easily hang the renderer. Great care was taken in *3Delight* to check for the most common errors but there are still some which are not caught. It is also important that your program be written to accept multiple requests.

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

#include "ri.h"

int main(int argc, char **argv)
{
    char buf[256];

    /*
      You can still use the standard error output to diagnose your program.
      This allows you to see that your program is started only once even if it
      is invoked several times in a frame.
    */
    fprintf(stderr, "diagnostic:  sphere program started.\n");

    /*
      Requests from the renderer to the program are passed on the standard
      input.  Each request is written on a single line.  The detail level required
      is written first, followed by the arguments passed by the user in the RIB.
      The two are separated by a single space.
    */
    while (fgets(buf, 256, stdin) != NULL)
    {
        RtFloat detail;
        RtColor color = {1.0f, 1.0f, 1.0f};
        sscanf(buf, "%g %g %g %g", &detail, &color[0], &color[1], &color[2]);

        /*
          Calling RiBegin with a file name as a parameter causes the commands
          to be output as RIB to that file when using lib3delight.  Using
          "stdout" or "stderr" will output the RIB to the standard output or
          error respectively.

          It is important to call RiBegin()/RiEnd() inside the loop (for each
          request) to ensure that the output is received properly by the
          renderer.
        */
        RiBegin("stdout");

        RiColor(&color[0]);
        RiSphere(1.0f, -1.0f, 1.0f, 360.0f, RI_NULL);

        /*
```

```
            Outputting a single 0xFF character (377 in octal) is the method to
            signal the renderer that the program has finished processing this
            request.  This can also be done manually if you choose not to use the
            library to output your RIB.
        */
        RiArchiveRecord(RI_VERBATIM, "\377");

        RiEnd();
    }

    fprintf(stderr, "diagnostic:  sphere program is about to end.\n");
    return 0;
}
```

This example can be compiled with the following commands:

Linux `g++ -o sphere -O3 -I$DELIGHT/include/ -L$DELIGHT/lib/ sphere.cpp -l3delight`

Mac OS X `g++ -o sphere -O3 -I$DELIGHT/include/ -L$DELIGHT/lib/ sphere.cpp -l3delight`

IRIX `CC -o sphere -O3 -n32 -TARG:isa=mips4 -TARG:processor=r10k -I$DELIGHT/include/ -L$DELIGHT/lib/ sphere.cpp -l3delight`

Windows `CL /Ox /I"%DELIGHT%\include" sphere.cpp "%DELIGHT%\lib\3Delight.lib"`

10.1.4.2 The DynamicLoad Procedural Primitive

This example shows how to write a procedural primitive in the form of a DSO. It is special in that it calls itself a number of times to create a Menger's Sponge[3]. It can be invoked in a RIB stream as such:

```
Procedural "DynamicLoad" ["sponge" "3"] [-1 1 -1 1 -1 1]
```

The single parameter it takes is the maximal recursion depth. It also makes use of the '`detailsize`' parameter of the subdivision routine to avoid outputting too much detail. Note that *3Delight* attempts to find `sponge.so` (or `sponge.dll` on Windows) and then `sponge` in the procedural search path. This allows you not to specify the extension.

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

#include "ri.h"

#if defined(_WIN32)
#define DLLEXPORT __declspec(dllexport)
#else
#define DLLEXPORT
#endif

#ifdef __cplusplus
extern "C" {
```

[3] Refer to http://mathworld.wolfram.com/MengerSponge.html for information about this fractal.

```
#endif

/* Declarations */
RtPointer DLLEXPORT ConvertParameters(RtString paramstr);
RtVoid DLLEXPORT Subdivide(RtPointer data, float detail);
RtVoid DLLEXPORT Free(RtPointer data);

RtPointer DLLEXPORT ConvertParameters(RtString paramstr)
{
    int* depth = (int*) malloc(sizeof(int));
    *depth = 3;                /* decent default value */
    sscanf(paramstr, "%d", depth);
    return depth;
}

RtVoid DLLEXPORT Subdivide(RtPointer blinddata, RtFloat detailsize)
{
    int depth = *(int*) blinddata;

    /* Simple usage of detailsize to avoid drawing too much detail */
    if (depth <= 0 || detailsize <= 5.0f)
    {
        /* Draw a cube */
        RtInt nverts[] = {4, 4, 4, 4, 4, 4};
        RtInt verts[] = {
            3, 7, 6, 2,        /* top face    */
            5, 1, 0, 4,        /* bottom face */
            7, 3, 1, 5,        /* back face   */
            3, 2, 0, 1,        /* left face   */
            6, 7, 5, 4,        /* right face  */
            2, 6, 4, 0};       /* front face  */

        RtFloat points[] = {
            -1, -1, -1,      -1, -1, 1,
            -1, 1, -1,       -1, 1, 1,
            1, -1, -1,       1, -1, 1,
            1, 1, -1,        1, 1, 1};
        RiPointsPolygons(
            (RtInt)6, nverts, verts, RI_P, (RtPointer)points, RI_NULL);
    } else {
        /* Recursive call, reduce depth and scale the object by 1/3 */
        RtBound bound = {-1, 1, -1, 1, -1, 1};
        int* newDepth;
        unsigned x,y,z;
        RiScale(1.0/3.0, 1.0/3.0, 1.0/3.0);

        for (x = 0; x < 3; ++x)
            for (y = 0; y < 3; ++y)
                for (z = 0; z < 3; ++z)
                    if (x % 2 + y % 2 + z % 2 < 2)
                    {
```

```
                        RiTransformBegin();
                        RiTranslate(
                            x * 2.0 - 2.0,
                            y * 2.0 - 2.0,
                            z * 2.0 - 2.0);
                        newDepth = (int*) malloc(sizeof(int));
                        *newDepth = depth - 1;
                        /* We could make the recursive call using
                           RiProcDynamicLoad but that would be more complex and
                           slightly less efficient */
                        RiProcedural(newDepth, bound, Subdivide, Free);
                        RiTransformEnd();
                    }
    }
}

RtVoid DLLEXPORT Free(RtPointer blinddata)
{
    free(blinddata);
}

#ifdef __cplusplus
}
#endif
```

This example can be compiled with the following commands:

Linux gcc -o sponge.so -O3 -I$DELIGHT/include/ -shared sponge.c

Mac OS X gcc -dynamiclib -o sponge.so -O3 -I$DELIGHT/include/ -L$DELIGHT/lib/ sponge.c -l3delight

IRIX CC -o sponge.so -O3 -n32 -TARG:isa=mips4 -TARG:processor=r10k -I$DELIGHT/include/ -shared sponge.c

Windows CL /Ox /LD /I"%DELIGHT%\include" sponge.c "%DELIGHT%\lib\3Delight.lib"

10.2 3Delight APIs

10.2.1 The Rx API

The Rx library provides access to some useful internal functions to SL DSOs writers. Also, it allows `RiProcedural` writers to query internal state of the renderer.

10.2.1.1 Noise functions

There are three noise functions. They access respectively the internal implementations of `noise`, `pnoise` and `cellnoise` in the shading language (see Section 6.4.2 [noise and random shadeops], page 91). They are accessible to both DSO shadeops and `RiProcedural` code.

`RtInt RxNoise` (*RtInt* `i_inDimension`, *RtFloat* *`i_in`, *RtInt* `i_outDimension`, *RtFloat* *`o_out`)

`RtInt RxPNoise` (*RtInt* `i_inDimension`, *RtFloat* *`i_in`, *RtFloat* *`i_period`, *RtInt* `i_outDimension`, *RtFloat* *`o_out`)

`RtInt RxCellNoise ( RtInt i_inDimension, RtFloat *i_in, RtInt i_outDimension, RtFloat *o_out )`

As input, these functions take one to four floats. The number of floats is passed by *i_inDimension* and the values by *i_in*. The expected number of dimensions as a result should be passed by *i_outDimension* (only one and three are accepted). It is important to provide a big enough *o_out* buffer to contain the output. The *i_period* parameter for `RxPNoise` should contain as many periods as there are input dimensions. A return value of '`0`' indicates a sucess.

`RtFloat RxRandom ( )`

returns a random number, exactly as the `float random()` shadeop.

10.2.1.2 Texture lookup functions

There are three texture lookup functions with several variants for each. They call respectively the implementations of `environment`, `shadow` and `texture` in the shading language (see Section 6.4.6 [texture mapping shadeops], page 107).

`RtInt RxEnvironmentPoints1 ( RtString i_fileName, RtInt i_nPoints, RtInt i_firstChannel, RtInt i_nChannels, const RtPoint *i_dir, RtFloat *o_result, ... )`
`RtInt RxShadowPoints1 ( RtString i_fileName, RtInt i_nPoints, RtInt i_firstChannel, const RtPoint *i_P, RtFloat *o_result, ... )`
`RtInt RxTexturePoints1 ( RtString i_fileName, RtInt i_nPoints, RtInt i_firstChannel, RtInt i_nChannels, const RtFloat *i_s, const RtFloat *i_t, RtFloat *o_result, ... )`

Each function takes a texture name and an input point specifying the texture location to be sampled. When these are called over all the points being shaded at once, proper derivatives are automatically computed for smooth filtering. Otherwise, the corresponding function below is called with the same point duplicated four times (ie. no filtering). *i_nChannels* channels is retrieved starting at channel *i_firstChannel* in the texture *i_fileName*. Optional parameters (described in Table 6.16) provided as name-value pairs as accepted. Named parameters which are varying must be explicitely declared as such inline (eg. `"varying float blur"`) or they will be read as uniform. All read channels are stored in *o_result*. These three functions also have their vector versions (`RxEnvironmentPoints1V`, `RxShadowPoints1V` and `RxTexturePoints1V`) which take a vector of tokens and a vector of values, similar to Ri calls. A return value of '`0`' indicates a sucess.

`RtInt RxEnvironmentPoints4 ( RtString i_fileName, RtInt i_nPoints, RtInt i_firstChannel, RtInt i_nChannels, const RtPoint *i_dir0, const RtPoint *i_dir1, const RtPoint *i_dir2, const RtPoint *i_dir3, RtFloat *o_result, ... )`
`RtInt RxShadowPoints4 ( RtString i_fileName, RtInt i_nPoints, RtInt i_firstChannel, const RtPoint *i_P0, const RtPoint *i_P1, const RtPoint *i_P2, const RtPoint *i_P3, RtFloat *o_result, ... )`
`RtInt RxTexturePoints4 ( RtString i_fileName, RtInt i_nPoints, RtInt i_firstChannel, RtInt i_nChannels, const RtFloat *i_s0, const RtFloat *i_t0, const RtFloat *i_s1, const RtFloat *i_t1, const RtFloat *i_s2, const RtFloat *i_t2, const RtFloat *i_s3, const RtFloat *i_t3, RtFloat *o_result, ... )`

These functions are the same as the previous three except that they take four points to explicitely specify the filtering area.

`RtInt RxEnvironment` (*RtString* `i_fileName`, *RtInt* `i_firstChannel`, *RtInt* `i_nChannels`, *RtPoint* `i_dir0`, *RtPoint* `i_dir1`, *RtPoint* `i_dir2`, *RtPoint* `i_dir3`, *RtFloat* *`o_result`, ...)

`RtInt RxShadow` (*RtString* `i_fileName`, *RtInt* `i_firstChannel`, *RtPoint* `i_P0`, *RtPoint* `i_P1`, *RtPoint* `i_P2`, *RtPoint* `i_P3`, *RtFloat* *`o_result`, ...)

`RtInt RxTexture` (*RtString* `i_fileName`, *RtInt* `i_firstChannel`, *RtInt* `i_nChannels`, *RtFloat* `i_s0`, *RtFloat* `i_t0`, *RtFloat* `i_s1`, *RtFloat* `i_t1`, *RtFloat* `i_s2`, *RtFloat* `i_t2`, *RtFloat* `i_s3`, *RtFloat* `i_t3`, *RtFloat* *`o_result`, ...)

These functions perform a single lookup at a time and do not provide automatic derivatives. They are present for compatibility and should not be used in new code when there are several lookups to perform. Using the Points version instead.

10.2.1.3 Transformation functions

There is a single space transformation function: `RxTransformPoints`. It looks a lot like the `transform` shadeop in the shading language (Section 6.4.3 [geometric shadeops], page 92), except that it passes an array of points to transform and accepts a time parameter (useful with motion blurred geometry). There is also `RxTransform` which allows a transformation matrix to be retrieved without actually transforming anything.

`RtInt RxTransformPoints` (*RtToken* `i_fromspace`, *RtToken* `i_tospace`, *RtInt* `i_n`, *RtPoint* `io_p`[], *RtFloat* `i_time`)

The points in *io_p* are transformed from *i_fromspace* to *i_tospace*, at time *i_time*. If there is some motion blur and *i_time* is somewhere inside the shutter time interval, the transformation interpolates points between the two enclosing time steps. A return value of '`0`' means success.

`RtInt RxTransform` (*RtToken* `i_fromspace`, *RtToken* `i_tospace`, *RtFloat* `i_time`, *RtMatrix* `o_matrix`)

Computes, in *o_matrix*, the matrix to transform points from *i_fromspace* to *i_tospace* at time *i_time*. A return value of '`0`' means success.

10.2.1.4 Message passing and info functions

These functions access state and general information. They are respectively the equivalent of `attribute()`, `option()`, `rendererinfo()` and `textureinfo()` shadeops in the shading language (see Section 6.4.8 [Message Passing and Information], page 113).

`RtInt RxAttribute` (*RtString* `i_name`, *RtPointer* `o_result`, *RtInt* `i_resultLen`, *RxInfoType_t* *`o_resulttype`, *RtInt* *`o_resultcount`)

`RtInt RxOption` (*RtString* `i_name`, *RtPointer* `o_result`, *RtInt* `i_resultLen`, *RxInfoType_t* *`o_resulttype`, *RtInt* *`o_resultcount`)

`RtInt RxRendererInfo` (*RtString* `i_name`, *RtPointer* `o_result`, *RtInt* `i_resultLen`, *RxInfoType_t* *`o_resulttype`, *RtInt* *`o_resultcount`)

`RtInt RxTextureInfo` (*RtString* `i_texture`, *RtString* `i_name`, *RtPointer* `o_result`, *RtInt* `i_resultLen`, *RxInfoType_t* *`o_resulttype`, *RtInt* *`o_resultcount`)

The information *i_name* is retrieved and stored into *o_result*, which has a size of at least *i_resultLen* in bytes. The type is returned in *o_resulttype* and the number of base elements (of type RtString or RtFloat) contained in *o_result* is stored in *o_resultcount*. These two values may be useful for user defined attributes or options. For `RxAttribute` and `RxOption`, a return value N > 0 indicates that the call failed due to a too small buffer (N indicating the number of missing bytes). In this case, take a look at *o_resulttype* and *o_resultcount* for more explanations. A return value of 0 indicates a success. A negative return value indicates an error.

10.2.1.5 File functions

`RtString RxCacheFile` (*RtString* `i_filename`)

This function allows access to *3Delight*'s network cache (see Section 7.14 [Network Cache], page 154). When the network cache is enabled, the given file is cached and a path to the cached file is returned. If caching fails, is disabled or is not required, *i_filename* is returned. This function is useful when the caching of extra resources is necessary due to excessive network loads. Example resources that could benefit from caching:

- Scene data. Such as *Maya* scene files or specific custom DSO data.
- Executables. Executing files through the network is a non negligible part of the total network traffic.

Note that 3DELIGHT already caches RIB archives so caching them using this call is not a good idea.

`RtString RxFindFile` (*RtString* `i_category`, *RtString* `i_filename`)

This function allows access to *3Delight*'s searchpath system. It will look for the requested file in the searchpaths of the given category (as given by the "searchpath" option). It will also apply any required directory mapping (see [Search Paths], page 39).

10.2.1.6 Example

Here is a simple example of a DSO using the `RxTexture` function. Note that the `rx.h` file is included instead of the `ri.h` file.

```
#include "shadeop.h"
#include "rx.h"

/*
        A simple DSO shadeop using the Rx Library

        Notes that 'extern "C"' is not necessary for '.c' files.
        Only c++ files need that.
*/

extern "C"
{
        SHADEOP_TABLE(rxTexture) =
        {
                {"color rxTexture(string, float, float, float, float, float, "
                 "float, float, float, float)", "rx_init", "rx_cleanup"},
                {""}
        };
}

extern "C" SHADEOP_INIT(rx_init)
{
        return 0x0; /* No init data */
}

extern "C" SHADEOP_CLEANUP(rx_cleanup)
{
        /* Nothing to do */
```

```
}

/*
        Given a texture file, texture coordinates and a width,
        return texture color using a gaussian filter.
        Note that the filename in argv[1] is a pointer of a string.
*/

extern "C" SHADEOP(rxTexture)
{
        RtString filter = "gaussian";

        return RxTexture(
                *((RtString *)argv[1]), 0, 3,
                *((RtFloat *) argv[2]), *((RtFloat *) argv[3]),
                *((RtFloat *) argv[4]), *((RtFloat *) argv[5]),
                *((RtFloat *) argv[6]), *((RtFloat *) argv[7]),
                *((RtFloat *) argv[8]), *((RtFloat *) argv[9]),
                (RtFloat *) argv[0],
                "width", (RtFloat*)argv[10],
                "filter", &filter,
                0);
}
```

The DSO has to be compiled with the `-shared` flag activated, as shown below for each platform:

Linux `g++ -shared -o rxexample.so -I$DELIGHT/include rxexample.cpp`

IRIX `CC -shared -o rxexample.so -I$DELIGHT/include rxexample.cpp`

MacOS X `g++ -dynamiclib -o rxexample.so -I$DELIGHT/include rxexample.cpp`

Windows `cl -I"%DELIGHT%/include" "%DELIGHT%/lib/3delight.lib" -LD rxexample.cpp`

10.2.2 The Gx API

This API allows the evaluation of RenderMan geometry. The design principal is to provide the user with an interface that is easy to use and coherent with the rest of the RenderMan protocol. All geometry types that are supported by 3Delight can be evaluated using this API, this includes: subdivision surfaces (including hierarchical subdivision surfaces), polygons, NURBS, conics, ... Etc. To use the interface one must include the `gx.h` header file.

10.2.2.1 Entry Points

`GxGeometryHandle GxGetGeometry` (*RtObjectHandle i_object, ...*)
`GxGeometryHandle GxGetGeometryV` (*RtObjectHandle i_object,*
RtInt i_n, RtToken i_tokens[], RtPointer i_params[]);
Prepares a stored object for processing with this API. The object must be declared using a `RiObjectBegin`/`RiObjectEnd` block. The *i_object* parameter is the one returned by `RiObjectBegin`.

`void GxFreeGeometry` (*GxGeometryHandle i_geo*)
Release memory and resources that have been allocated by `GxGetGeometry`. This should be done when the object and all the other resources derived from it will no longer be used.

`unsigned GxGetFaceCount` (*GxGeometryHandle i_geo*)

Returns the number of faces for a given piece of geometry. This can be different from what is specified through the RI interface as *3Delight* may transform some types of geometry into a different internal representation.

`int GxCreateSurfacePoint` (*GxGeometryHandle i_geo, unsigned i_face, float i_u, float i_v, float i_time, GxSurfacePoint *o_point*)

Initializes a surface point handle directly with a face number of parametric coordinates on the face. Parameters are:

i_geo The geometry to evaluate.

i_face The face number. Must be in the [0 .. GxGetFaceCount(i_geo)[range.

u
v The parametric coordinates on the face (local to the face).

i_time Time. Currently unimplemented[4].

o_point A pointer to the handle to be initialized. The initialized handle must be cleaned up with `FreeSurfacePoint` or resources will leak.

Possible return values all:

`RIE_NOERROR`
Everything went well.

`RIE_RANGE` Invalid face id.

`RIE_BADHANDLE`
Invalid *i_geo*.

`void GxFreeSurfacePoint` (*GxSurfacePoint i_point*)

Releases the memory associated with a surface point handle that was previously allocated by `GxCreateSurfacePoint`.

`int GxEvaluateSurface` (*unsigned i_npoints, GxSurfacePoint *i_points, RtToken i_variable, unsigned i_width, float *o_values*)

Evaluates a primitive variable at the given surface points.

i_npoints Length of the i_points array.

i_points A pre-allocated array of `GxSurfacePoint`s.

i_variable The name of the primitive variable to evaluate (eg. `"P"`). Any primitive variable that is attached to the geometry can be evaluated using this function. Note that `"dPdu"`, `"dPdv"`, `"N"` and `"Ng"` can always be passed to this function (these variables are inherent to the geometry and don't have to be passed as a primitive variable).

i_width The number of components of i_variable (eg. 3 for `"P"`).

o_values A buffer to receive the results. This should be i_npoints * i_width long.

Possible return values are:

`RIE_NOERROR`
Everything went well.

[4] Don't worry, time stills work in your reality

`RIE_BADHANDLE`
: At least one the the evaluation points is invalid.

`RIE_MISSINGDATA`
: The provided token (*i_varibale*) cannot be found in the geometry.

`RIE_CONSISTENCY`
: *i_width* is not equal to the with of the primitive variable declared in the geometry.

`RIE_BADTOKEN`
: *i_variable* is of type `RtString` so it cannot be evaluated.

10.2.2.2 Example Usage

A complete example can be found in `$DELIGHT/examples/gx`. The directory contains code for a procedural that transforms RenderMan geometry into particles. To run the example of a Linux or Mac OS X system, open a shell and execute the following commands:

```
cd $DELIGHT/examples/gx
make
```

A framebuffer should popup and some primtives will be rendered as points. The primitives are all described in `points.rib`.

10.2.3 The Sx API

This API allows the evaluation of RenderMan shaders on arbitrary user data. The API doesn't expect any scene description so its usable from a variety of contexts. Usage examples include:

- Evaluation of displacement shaders in animation packages to correctly place fur and hair on a displaced surface.
- Evaluation of RenderMan shaders on object vertices for a rough preview of RenderMan shaders in an animation package.
- Using the library to access the high quality and efficient texturing mapping shadeops.
- Using the library to design other libraries which take RenderMan shaders as inputs. This is well illustrated by the "VolumeTracer" API (see Section 10.2.7 [The VolumeTracer API], page 240).

The library operates in SIMD so many points can be evaluated at once.

10.2.3.1 Entry Points

`SxContext SxCreateContext` (*SxContext i_parent = 0*)
: Creates a shader evaluation context.

  i_parent Specifies the parent context.

`void SxDestroyContext` (*SxContext io_context*)
: Destroys the provide *io_context*.

`void SxSetOption` (*SxContext i_context, const char* i_optionName, SxType i_optionType, SxData i_values, unsigned i_arraySize*)
`void SxSetAttribute` (*SxContext i_context, const char* i_optionName, SxType i_optionType, SxData i_values, unsigned i_arraySize*)
: Sets a rendering option or attribute in the given context. All options are accessible from inside the shader through the `option()` and `attribute()` shadeops (see Section 6.4.8 [Message Passing and Information], page 113). This is similar in spirit to the `RiOption` and `RiAttribute` RenderMan commands.

i_context The `SxContext` as returned by `SxCreateContext()`.

i_optionType
Type of the option.

i_values Pointer to the actual values.

i_arraySize Total number of elements in the array (0 if the variable is not an array).

Note that all options are *uniform*.

`void SxDefineSpace` (*SxContext i_context, const char* i_spaceName, float *i_matrix*)
Binds a matrix to the specified space in the specified context. Any name can be specify except '`current`' which is reserved by this library. This function must be called before `SxCreateShader` for each space of interest for the new shader.

`SxParameterList SxCreateParameterList` (*SxContext i_context, unsigned i_numPoints, const char* i_spaceName*)
Creates a parameter list for use in either intializing a shader instance or running a shader on a set of points. *i_numPoints* specifies the number of values for varying parameters when evaluating the shader. If the list is used to create a shader then this should be 1 as only the first value is use to set the shader's intance parameters. The optional *i_spaceName* indicates the space in which the parameters are declared (this matters in case of points, vectors, normals and matrices). It defaults to "shader" if the list is used to create a new shader and "current" if it is used to evaluate a shader. The parameter list will be automatically freed when the context specified by *i_context* is destroyed. However, `SxDestroyParameterList()` may also be used to destroy the parameter list earlier. This is mostly useful for the list passed to `SxCallShader()`.

`void SxSetParameterListGridTopology` (*SxParameterList i_paramList, unsigned i_numGrids, const unsigned *i_uRes, const unsigned *i_vRes*)
Sets a parameter list to be interpreted as grids instead of independent points. *i_uRes* and *i_vRes* are two arrays of *i_numGrids* values which specify the number of points of each grid in both axis. Using this function affects the behavior of area and filtered functions in the shader.

`void SxSetPredefinedParameter` (*SxParameterList i_paramList, const char* i_predefParamName, SxData i_values, bool i_varying = true*)
Sets (or adds) a predefined variable to the specified parameter list. Predefined parameters are P, u, v, etc ...

i_paramList
Specified parameter list

i_predefParamName
Predefined variable's name

i_values Predefined variable's values

i_varying Predefined variable's class (optional). True for varying, false for uniform.

`void SxDestroyParameterList` (*SxParameterList i_parameterList*)
Destroys a previous constructed parameter lists.

`SxShader SxCreateShader` (*SxContext i_context, SxParameterList i_paramList, const char *i_shaderName, const char *i_shaderHandle*)
Creates a shader in the specified context.

i_context The context where the shader is created and to which it will belong.

i_paramList
: The parameter list which contains the shader instance parameter values. These are the values which will not change for different calls to the shader. This parameter may be null if no list is needed.

i_shaderName
: The name of the shader file to load (without the .sdl part).

i_shaderHandle
: The name by which the shader will be referenced. This should be null for shaders which you do not intend to use as co-shaders.

`unsigned SxGetNumParameters` (*SxShader i_shader*)
: Returns the number of parameters declared for the specified shader.

`unsigned SxGetPredefinedParameters` (*SxShader i_shader, const char* o_names*[], *unsigned i_maxNames*)
: Returns the number of predefined parameters needed by *i_shader* and fills *o_names* with pointers to their names.

i_shader The shader of interest.

o_names An array that will be filled with the parameters' names. May be `null` when *i_maxNames* is set to `0`.

i_maxNames
: The maximum number of strings that can be written to *o_names*. It can be useful to set this parameter to `0` when only the number of predefined parameters (and not their names) is needed.

`const char*` *SxGetParameterInfo* (*SxShader i_shader, unsigned i_index, SxType* o_type, bool* o_varying, SxData* o_defValues, unsigned * o_arraySize, const char** *o_spaceName = 0, bool* o_output = 0*)
: Returns parameter's name and all information about the parameter which matches the specified index for the specified shader.

i_shader The shader of interest.

i_index Specified index

o_type Parameter's type (would be one of SxFloat, SxPoint, SxColor, SxString, SxMatrix, SxVector or SxNormal).

o_varying Parameter's class (would be true for varying, false for uniform).

o_defValues
: Parameter's default values.

o_arraySize Parameter's array size. Would be 0 if not an array.

o_spaceName
: Parameter's space for the default value (optional).

o_output Parameter's output state (optional). Would be true for output, false otherwise.

`unsigned SxGetParameter` (*SxParameterList i_paramList, const char* i_paramName, SxData* o_values*)
: Returns the number of values associated with the specified output variable.

i_paramList
The parameter list from which to retrieve values. Normally this will be a list which was passed to SxCallShader.

i_paramName
Variable's name.

o_values Values of the predefined or output variable.

`bool SxCallShader` (*SxShader i_shader, SxParameterList i_paramList*)
Calls (executes) a shader.

i_shader Shader to evaluate.

i_paramList
Parameter list to use (created by `SxcreateParameterList`).

Result of the evaluation can be queried using the `SxGetParameter()` function on i_paramList. The shader is run with the lights and co-shaders which were created in the same Sx context. Any light shader created is added to the active light list and any shader created with a non null handle name is added to the co-shaders list.

10.2.3.2 Example Usage

10.2.4 The Point Cloud API

An easy to use `C++` API is provided with *3Delight* to read and write point cloud files as those generated by the `bake3d()` shadeop. To use the API, one has to include the `pointcloud.h` header file that is distributed in the package and link the application with the *3Delight* library (see Section 10.3 [linking with 3delight], page 244).

10.2.4.1 Point Cloud API Data Types

Only one type is defined in `pointcloud.h`

```
typedef void * PtcPointCloud
```

This `typedef` is used as a handle to a point cloud file on disk.

10.2.4.2 Point Cloud Reading

`PtcPointCloud PtcSafeOpenPointCloudFile` (*const char* ***filename***)
This function opens a given file for reading and returns a file handle than shall be used in other reading calls.

`PtcPointCloud PtcOpenPointCloudFile` (*const char* ***filename***, *int* ***nvars***, *const char* ****vartypes***, *const char* ****varnames***)

This function opens a given file for reading and returns a file handle than shall be used in other reading calls.

filename The complete path to the point cloud file

nvars A pointer to an `int` that will be filled with the total number of channels in the point cloud file.

varnames A pointer to an array of `const char *` that will hold the name of each variable.

vartypes A pointer to an array of `const char *` that will hold the type of each variable. Types are string representing the type of the variable as declared in the shading language (color, point, float, normal, ... etc).

This function could fail if the file is not accessible or is not a valid point cloud file, in which case `null` is returned. Data allocated by this function is managed by the library and not user intervention is necessary to deallocate it.

NOTE: This API call is badly designed since the caller cannot know the size of *vartypes* and *varnames* in advance. But for compatibly reasons with other software we decided to provide this API entry. A safe way to open a point-cloud file is to call `PtcSafeOpenPointCloudFile()` and then call `PtcGetPointCloudInfo()` to get *nvars*, *varnames* and *vartypes*.

`int PtcGetPointCloudInfo (` *PtcPointCloud* `pointcloud`, *const char* `*request`, *void* `*result`)

This function returns informations about a point cloud file. Accepted *request*s are:

pointcloud A handle to the point cloud file as returned by `PtcOpenPointCloudFile`

request The name of the information needed. The following requests are accepted:

'`npoints`' Number of points in the point cloud file. C++ data type is `int`

'`bbox`' The bounding box of the point cloud. Will return an array of six floats: min x, min y, min z, max x, max y and max z.

'`datasize`'
'`pointdatasize`'
The number of `float`s needed to store each data point. C++ data type is `int`.

'`world2eye`'
The world to eye (world to camera) transformation matrix. Will return an array of 16 `float`s.

'`world2ndc`'
The world to NDC transformation matrix. Will return an array of 16 `float`s.

'`format`' The resolution of the render that generated the point cloud file. Three `float`s will be returned: x resolution, y resolution and aspect ratio.

'`nvars`' Total number of variables attached to each point. In this case, *result* is considered to be of type `int *`.

'`varnames`'
'`pointvarnames`'
The name of each variable. In this case, *result* is considered to be of type `char **`.

'`vartypes`'
'`pointvartypes`'
The type of each variable. In this case, *result* is considered to be of type `char **`.

result A pointer to an array large enough to hold the returned informations.

Returns 1 if the request is successful, 0 otherwise.

NOTE: Some point cloud files generated with older *3Delight* versions may not contain the '`format`' information.

`int PtcReadDataPoint` (*PtcPointCloud* **`pointcloud`**, *float* *`point`, *float* *`normal`, *float* *`radius`, *float* *`user_data`)

Reads next point from the point cloud file. The parameters are:

pointcloud	The handle to the point cloud file as returned by `PtcOpenPointCloudFile`.
point	A pointer to a point (three floats) that will be filled with current point's position.
normal	A pointer to a point (three floats) that will be filled with current point's normal.
radius	A pointer to float that will be filled with point's radius. The area of the micropolygon that generated this sample is $radius * radius * PI$.
user_data	A pointer to a user buffer of a *size big enough* to hold all the variables attached to a point. The size of the buffer, in `float`s, can be obtained by calling `PtcGetPointCloudInfo` with *request* set to '`datasize`'.

Returns 1 if the operation is successful, 0 otherwise.

NOTE: *point*, *normal*, *radius* and *user_data* can be null if their value is not neede.

`void PtcClosePointCloudFile` (*PtcPointCloud* **`pointcloud`**)

Closes a file opened with `PtcOpenPointCloudFile`. This function releases all memory allocated by `PtcOpenPointCloudFile`.

10.2.4.3 Point Cloud Writing

`PtcPointCloud PtcCreatePointCloudFile` (*const char* *`filename`, *int* **`nvars`**, *const char* **`vartypes`**, *const char* **`varnames`**, *float* *`world2eye`, *float* *`world2ndc`, *float* *`format`)

Creates the specified point cloud file. If the point cloud file already exists, it will be overwritten.

filename	Complete path to point cloud file.
nvars	Number of variables to save in the point cloud.
vartype	A type for each variable. Types are the same as in the shading language: point, normal, color, float, matrix ... Etc.
varname	A name for each variable.
world2eye	A world to camera transformation matrix.
world2ndc	A world to NDC transformation matrix.
format	The X resolution, Y resolution and aspect ratio of the image.

`int PtcWriteDataPoint` (*PtcPointCloud* **`pointcloud`**, *float* *`point`, *float* *`normal`, *float* **`radius`**, *float* *`data`)

Adds a point, along with its data, to a point cloud file.

pointcloud	A handle to a point cloud file as returned by `PtcCreatePointCloudFile`.

point
normal
radius Position, orientation and radius of the point and data. *point* and *normal* cannot be null.

data Array of floats containing data for all variables, continuously in memory. The data must be of the same size as the sum of sizes of the variables passed to `PtcCreatePointCloudFile`.

Returns 1 if the operation is successful, 0 otherwise.

`void PtcFinishPointCloudFile ( PtcPointCloud pointcloud)`

Writes out all data to disk and closes the file.

10.2.4.4 API Example

An example is available in `$DELIGHT/examples/ptc2rib` directory. This simple application transforms a point cloud file into a RIB that is renderable using `renderdl`. An example usage is:

```
ptc2rib cloud.ptc | renderdl -d
```

10.2.5 The Slo API

Shader interrogation is possible by using the `shaderinfo` command line tool (see Section 3.6 [Using shaderinfo], page 23). It is also possible to "manually" interrogate shaders by using a number of calls implemented in `lib3delight`. The interrogating application should include `slo.h` (found in `$DELIGHT/include`) and link with `lib3delight` (see Section 10.3 [linking with 3delight], page 244).

Here is the complete list for shader interrogation calls:

`void Slo_SetPath ( char* i_path )`
Set the paths where the library looks for shaders. It accepts a colon separated list of paths. Refer to [Search Paths], page 39 for more information on search paths which also applies to this function.

`int Slo_SetShader ( char* i_name )`
Find and open the shader named *i_name*. Close any shader that was previously opened by `Slo_SetShader`. Returns 0 when the shader was successfully loaded.

`char* Slo_GetName ( void )`
Returns the name of the currently opened shader.

`SLO_TYPE Slo_GetType ( void )`
Return the type of the currently opened shader. See the file `slo.h` for details about `SLO_TYPE`.

`int Slo_GetNArgs ( void )`
Return the number of parameters accepted by this shader.

`SLO_VISSYMDEF* Slo_GetArgById ( int i )`
Return information about the *i*-th parameter of the shader. The first parameter has an id of 1. See `slo.h` for details about `SLO_VISSYMDEF` structure.

`SLO_VISSYMDEF* Slo_GetArgByName ( char *i_name )`
Return information about the parameter named *i_name*. See the file `slo.h` for details about `SLO_VISSYMDEF`.

`SLO_VISSYMDEF* Slo_GetArrayArgElement ( SLO_VISSYMDEF *i_array, int i_index )`
If a parameter is an array (as specified by `Slo_GetArgById()` or `Slo_GetArgByName`), each of its element should be accessed using this function.

`int Slo_GetNAnnotations` (*void*)
: Return the number of annotations contained in this shader. Annotations are further discussed in Section 10.4 [Attaching annotations to shaders], page 245.

`char* Slo_GetAnnotationKeyById` (*int* `i_id`)
: Returns an annotation key by it's index *i_id*. Acceptable indexes range from 1 to n, where n is the number of annotations available. Any index outside this range returns 0x0 (null pointer).

`char* Slo_GetAnnotationByKey` (*const char* *`i_key`)
: Returns an annotation specified by it's key. If *i_key* doesn't refer to any annotation, 0 (null pointer) is returned. It is possible to retrieve the different possible keys with `Slo_GetAnnotationKeyById`.

`void Slo_EndShader` (*void*)
: Close the current shader.

`char* Slo_TypetoStr` (*SLO_TYPE* `i_type`)
: Get a string representation of type *i_type*.

`char* Slo_StortoStr` (*SLO_STORAGE* `i_storage`)
: Get a string representation of storage class *i_storage*.

`char* Slo_DetailtoStr` (*SLO_DETAIL* `i_detail`)
: Get a string representation of variable detail *i_detail*.

`const char*const*` *Slo_GetMethodNames* ( )
: Returns a null terminated array of strings containing the names of the methods in the current shader.

Next is the complete source code of the `shaderinfo` utility, it is compilable on all supported platforms, see Section 10.3 [linking with 3delight], page 244.

```
/*****************************************************************************/
/*                                                                           */
/*    Copyright (c)The 3Delight Team.                                        */
/*    All Rights Reserved.                                                   */
/*                                                                           */
/*****************************************************************************/

// ===========================================================================
// = VERSION
//     $Revision$
// = DATE RELEASED
//     $Date$
// = RCSID
//     $Id$
// ===========================================================================

#include "slo.h"
#include "dlMsgShaderInfo.h"
#include "dlVersion.h"

#include <string.h>
#include <stdlib.h>
#include <stdio.h>

#include <assert.h>

typedef enum
{
          e_prman = 1,
          e_ribout,
          e_table,
          e_source,
          e_annotations,
          e_methods
} Format;

void PrintDefaultValue( const SLO_VISSYMDEF* arg )
{
          switch(arg->svd_type)
          {
          case SLO_TYPE_SCALAR:
                    printf( "%g", *arg->svd_default.scalarval );
                    break;
          case SLO_TYPE_STRING:
                    printf( "%s", arg->svd_default.stringval );
                    break;
          case SLO_TYPE_POINT:
          case SLO_TYPE_COLOR:
          case SLO_TYPE_VECTOR:
          case SLO_TYPE_NORMAL:
                    printf( "%g %g %g",
                              arg->svd_default.pointval->xval,
                              arg->svd_default.pointval->yval,
                              arg->svd_default.pointval->zval );
                    break;
          case SLO_TYPE_MATRIX:
          {
                    unsigned i=0;
```

```
                    for( ; i < 15; ++i )
                              printf( "%g ", arg->svd_default.matrixval[i] );
                    printf( "%g", arg->svd_default.matrixval[i] );
                    break;
          }
          case SLO_TYPE_SHADER:
                    printf( "nil" );
                    break;

          default:  ;
          }
}

/*
          Function to print one parameter.
*/
void PrintOneParameter( const SLO_VISSYMDEF *parameter, const Format i_format )
{
          const char *storage = Slo_StortoStr( parameter->svd_storage );
          const char *name = parameter->svd_name;
          const char *detail = Slo_DetailtoStr( parameter->svd_detail );
          const char *type = Slo_TypetoStr( parameter->svd_type );
          unsigned arraylen = parameter->svd_arraylen;
          bool is_array = parameter->svd_isarray;

          if( i_format == e_ribout )
          {
                    /* Output a string suitable for RIB syntax */

                    if( !is_array )
                              printf( "Declare \"%s\" \"%s %s\"\n", name, detail, type );
                    else
                              printf( "Declare \"%s\" \"%s %s[%d]\"\n",
                                        name, detail, type, arraylen );

                    return;
          }
          else if( i_format == e_table )
          {
                    printf( "%s,%s,%s,%s,%s,%d,",
                                  name, storage, detail, type,
                                  parameter->svd_spacename[0] ?
                                  parameter->svd_spacename :  "<none>",
                                  arraylen );

                    /* Print all default values */
                    if( !is_array )
                    {
                              PrintDefaultValue( parameter );
                    }
                    else
                    {
                              for( int i = 0; i < parameter->svd_arraylen; i++ )
                              {
                                        if( i != 0 )
                                        {
                                                  printf( " " );
                                        }
```

```
                                PrintDefaultValue(
                                        Slo_GetArrayArgElement((SLO_VISSYMDEF *)parameter, i) );
                        }
                }

                printf( "\n" );
                return;
        }

        assert( i_format == e_prman );

        printf( "    \"%s\" \"%s %s %s", name, storage, detail, type );

        if( is_array )
        {
                if( arraylen == 0 )
                        printf( "[]" );
                else
                        printf( "[%d]", arraylen );
        }

        printf( "\"\n" );
        printf( "\t\t%s", DlGetText(DlText_1725_default) );

        switch( parameter->svd_type )
        {
                case SLO_TYPE_COLOR:

                        // Even if color has a different spacename,
                        // it has been converted by evaluation.

                        printf( "\"rgb\" ");
                        break;

                case SLO_TYPE_POINT:
                case SLO_TYPE_VECTOR:
                case SLO_TYPE_NORMAL:
                case SLO_TYPE_MATRIX:

                        if( parameter->svd_spacename[0] != (char)0 )
                                printf( "\"%s\" ", parameter->svd_spacename );

                        break;

                default:
                        break;
        }

        if( is_array )
        {
                printf( "{" );
        }

        int num_elements = is_array ?  arraylen :  1;

        for( int k=0; k<num_elements; k++ )
        {
                const SLO_VISSYMDEF *elem =
```

```
                    Slo_GetArrayArgElement( (SLO_VISSYMDEF *)parameter, k );

          if( !elem )
          {
                    printf( "<error>" );
                    continue;
          }

          switch( parameter->svd_type )
          {
                    case SLO_TYPE_SCALAR:
                              printf( "%g", *elem->svd_default.scalarval );
                              break;

                    case SLO_TYPE_POINT:
                    case SLO_TYPE_VECTOR:
                    case SLO_TYPE_NORMAL:
                    case SLO_TYPE_COLOR:
                              printf( "[%g %g %g]",
                                        elem->svd_default.pointval->xval,
                                        elem->svd_default.pointval->yval,
                                        elem->svd_default.pointval->zval );
                              break;

                    case SLO_TYPE_MATRIX:
                              printf( "[%g ",  elem->svd_default.matrixval[0] );

                              for( int kk = 1; kk < 15; kk++ )
                                        printf( "%g ", elem->svd_default.matrixval[kk] );

                              printf( "%g]",  elem->svd_default.matrixval[15] );
                              break;

                    case SLO_TYPE_STRING:
                              printf( "\"%s\"", elem->svd_default.stringval );
                              break;

                    case SLO_TYPE_SHADER:
                              printf( "null" );
                              break;

                    default:
                              break;
          }

          if( k != (num_elements-1) )
          {
                    printf( ", " );
          }
     }

     if( is_array )
               printf( "}" );

     printf( "\n" );
}

void printArgs( const Format i_format )
```

```
{
          if( i_format == e_table )
          {
                    printf( "%d\n", Slo_GetNArgs() );
          }

          for( int j = 0; j < Slo_GetNArgs(); j++ )
          {
                    SLO_VISSYMDEF *parameter = Slo_GetArgById( j+1 );

                    if( !parameter )
                    {
                              fprintf( stderr, DlGetText(DlText_2259_param), j );
                              continue;
                    }

                    PrintOneParameter( parameter, i_format );
          }

          if( i_format == e_prman )
          {
                    printf( "\n" );
          }
}

int main( int argc, char ** argv )
{
          int i;
          int exitCode = 0;

          Format format = e_prman;

          int start = 1;

          /* DlDebug::InitErrorSignalsHandling(); */

          if ( argc <= 1 || !strcmp(argv[1], "-h") || !strcmp(argv[1], "--help") )
          {
                    printf(
                              "Usage:  shaderinfo [<option>] [file1 ...  fileN]\n"
                              "Options\n"
                              "  -d          :  Outputs declarations in RIB format\n"
                              "  -t          :  Outputs declarations in parsable table format\n"
                              "  -a          :  Outputs available annotation keys\n"
                              "  --source    :  Outputs shader's source (if embedded)\n"
                              "  --methods   :  Outputs shader's methods\n"
                              "  -v          :  Shows version to console\n"
                              "  --version   :  Same as -v\n"
                              "  -h          :  Shows this help\n\n"
                              "Notes:\n"
                              "- No arguments is the equivalent of passing -h\n"
                              "- All options are exclusive.  Use no more than one\n"
                              "- The options -h, -v, and --version should not be used with shader names\n"
                    );

                    return 0;
          }
```

```
if( argc == 2 && (!strcmp(argv[1], "-v") || !strcmp(argv[1], "--version")) )
{
          const char* v = DlGetVersionString();
          const char* copyright = DlGetCopyrightString();

          fprintf(stderr, "shaderinfo version %s.\n%s\n", v, copyright);

          return 0;
}

if ( argc == 1 )
{
          start++;
}
else if (!strcmp(argv[1], "-d"))
{
          format = e_ribout;
          start++;
}
else if( !strcmp(argv[1], "-t") )
{
          format = e_table;
          start++;
}
else if (!strcmp(argv[1], "-a"))
{
          format = e_annotations;
          start++;
}
else if (!strcmp(argv[1], "-source") || !strcmp(argv[1], "--source") )
{
          format = e_source;
          start ++;
}
else if (!strcmp(argv[1], "--methods") )
{
          format = e_methods;
          start ++;
}

/*
          Add '.'  to the default path so shaders are still found in the current
          directory even if the default path is modified not to include '.'
*/
Slo_SetPath( ".:@" );

for( i = start; i < argc; i++ )
{
          int err = Slo_SetShader( argv[i] );

          if( err != 0 )
          {
                    fprintf( stderr, DlGetText(DlText_2260_open), argv[i], err );
                    exitCode = 1;
                    continue;
          }

          /* Print a header indicating which shader this is */
```

```
if( format == e_table )
{
          printf( "%s\n%s\n",
                    Slo_GetName(), Slo_TypetoStr(Slo_GetType()) );
}
else if( format == e_prman )
{
          printf( "\n%s \"%s\"\n",
                    Slo_TypetoStr(Slo_GetType()), Slo_GetName() );
}
else if( format == e_methods )
{
          /* PrMan doesn't seem to print a header in --methods mode.  */
}
else if( format == e_source )
{
          /* We want the source that is output directly compilable.  So we don't
             want the name the shader in the listing.  */
}
else
{
          printf( "%s %s\n",
                    Slo_TypetoStr(Slo_GetType()), Slo_GetName() );
}

if( format == e_annotations )
{
          if( Slo_GetNAnnotations() == 0 )
          {
                    fprintf( stderr, DlGetText(DlText_2261_nokey), argv[i] );
                    continue;
          }

          printf( "%d\n", Slo_GetNAnnotations() );
          for( int id=0; id<Slo_GetNAnnotations(); id++ )
          {
                    const char *key = Slo_GetAnnotationKeyById( id+1 );

                    if( !key )
                    {
                              assert( false );
                              fprintf(stderr, DlGetText(DlText_1727_nokey), id, argv[i]);
                              return 1;
                    }

                    const char *annotation = Slo_GetAnnotationByKey(key);

                    printf( "\"%s\" \"%s\"\n", key, annotation );
          }
}
else if ( format == e_source )
{
          const char *value = Slo_GetAnnotationByKey( "source" );

          if( value )
          {
                    printf( "%s\n", value );
          }
```

```
                    else
                    {
                            fprintf( stderr, DlGetText(DlText_1796_no_source) );
                    }
            }
            else if ( format == e_methods )
            {
                    const char *const *method_names = Slo_GetMethodNames();

                    while( *method_names )
                    {
                            printf( "%s\n", *method_names );
                            method_names ++;
                    }
            }
            else
            {
                    printArgs( format );
            }

            Slo_EndShader();
    }

    return exitCode;
}
```

10.2.6 The Rix Interface API

The "Rix" API is a collection of classes giving access to useful methods. These classes and methods can be used from inside RSL plug-ins for example (see Section 10.1.2 [RSL Plug-ins], page 209). All classes are declared in file `$DELIGHT/include/RixInterface.h` and are further described in the following sections.

The `RixContext` class

All Rix Interface class *live* in a Rix Context. The only useful method in this class is `GetRixInterface`.

`RixInterface* RixContext::GetRixInterface` (*RixInterfaceID i_id*)

This method constructs and returns the Rix Interface specified by *i_id*. The *i_id* variable can take one of the following values:

`k_RixThreadUtils`
`k_RixMessages`
`k_RixStats`
`k_RixThreadData`
`k_RixLocalData`
`k_RixGlobalTokenData`

Returns an instance of the corresponding class (`RixThreadutils`, `RixMessages`, `RixThreadData`, etc...).

This method returns a pointer to `RxInterface` that needs to be cast into the appropriate class. For example:

```
/* Get a context */
RixContext *context = RxGetRixContext();
RixMessage *error_handler = (RixMessages*) context->GetRixInterface( k_RixMessages );
error_handler->Info( "This is an example" );
```

The `RixInterface` class

This is the base class for all other `Rix` classes. There is no particular functionality provided by this class apart from returning the version of the API and being the base type returned by `GetRixInterface`.

`int RixInterface::GetVersion ( ) const`
: Returns the version of the interface.

The `RixStorage` class

This class provides data storage and retrieval methods. Data is associated with a key and is accessible across *plug-in boundaries* when retrieved with the same key. Such a class is created through a `RixContext` object and can return local, global or thread-local data. For example:

```
RixContext *context = RxGetRixContext( );
RixStorate *thread_storage = (RixStorage *)context->GetRixInterface( k_RixThreadData );
thread->storage->Set( "example", somedata );
```

`void* RixStorage::Get ( const char *i_key )`
: Returns the data associated with the give key. `NULL` is returned if there is no data attached to the given key.

`void* RixStorage::Set ( const char *i_key, void *i_data, RixCleanupFunction i_clean = NULL )`
: Sets the data associated with the given key. If the key has associated data, it will be cleared and it associated cleanup function will be called (if provided with the previous `Set` function).

`void RixStorage::Lock ( )`
`void RixStorage::Unlock ( )`
: Lock or unlocks this object. Only necessary when accessing global storage.

The `RixThreadUtils` class

This class provides access to thread related methods. As of now, only one method is accessible to provide a mutex object.

`RiMutex RixThreadUtils::NewMutex ( ) const`
: Returns a RiMutex object. Mutexes are necessary to serialize concurrent access to globally memory. Please refer to [The RiMutex class], page 240 for more information. All mutex objects allocated using this method must be de-allocated using the `delete` operator.

The `RixMessage` class

Provides functions to print messages using *3Delight*'s error handler.

The `RiMutex` class

A cross-platform locking mechanism is provided through this class.

`void RiMutex::Lock ( )`
`void RiMutex::Unlock ( )`
: Locks or unlocks this object.

10.2.7 The VolumeTracer API

This API provides functions to trace rays in a voxel grid as suitable for volume rendering. It is available through a RSL plug-in so it is callable from inside regular RenderMan shaders. In summary, the VolumeTracer API allows a shader to:

- Manage a voxel grid. The voxel grid contains a certain number of attributes at each grid vertex. Those attributes are grouped into categories, each categories being computed by a RSL shader.
- Trace a bundle of rays in the voxel grid, in SIMD, and retrieve the values of all defined attributes at current ray position, computed using the indicated interpolation mode.

These provided functionalities make it easy to write efficient volume shaders without hassle.

10.2.7.1 Working Principles

To use the library one has to write at least two RenderMan shaders:

1. A classical ray-marching volume shader that uses the provided calls to trace rays, in SIMD, through 3D space.
2. A small shader that returns the volume attributes (there can be many) at any position in the volume. In practice, this shader will be called to query volume quantities at *each grid vertex*. Most commonly, this shader would return the *density* of the volume but other usages are of course possible. The name of this shader is provided to the initialization function of API (see Section 10.2.7.2 [VolumeTracer Entry Points], page 241) and is handled by the plug-in[1]. It must have a single *output* parameter : an array of floats. Any number of *input* parameters are allowed and can be initialized when creating the grid (usually to provide the name of a point cloud file).

The RSL plug-in library is located in `$DELIGHT/shaders/`.

10.2.7.2 Entry Points

`float VolumeTracer_InitGrid (` *uniform string* ***gridName****; ...* `)`

Creates and initializes a 3D grid to be used by the other API functions. Parameters are:

gridName Name of the grid. Might be used later by `VolumeTracer_NbAttributes` or `VolumeTracer_InitRay_Function`.

The other parameters of the grid are specified through name/value pairs :

uniform string *paramName*
: Name of the grid parameter.

uniform <type> *paramValue*
: Value of the parameter.

Supported grid parameters are :

'`grid:center`' (point)
: Center position of the grid in 3D space.

'`grid:size`' (vector)
: Size of the grid in each dimension, in the same 3D space as the position.

'`grid:nbcells`' (vector)
: Number of cells in the grid in each dimension.

'`grid:shader`' (string)
: Name of a surface shader that will be used to compute the attributes of the volume at selected sample points in the grid. The shader must have a single output parameter of type "float[]" (the array can be of any constant size), plus any number of input parameters. This defines an attribute category for the grid.

[1] The plug-in itself uses the Sx API to query this shader.

The size of the output array determines the number of attributes in the category. This parameter can be specified up to 3 times with different shader names in order to create multiple attribute categories.

In addition to this, parameters that need to be passed to the grid's shaders can also be specified through additional name/value pairs :

uniform string *paramName*
: Name of the shader parameter, prefixed with the shader name (as specified for the '`grid:shader`' grid parameter) and the `":"` parameter. For example to specifiy the parameter `"pointCloudFile"` of a shader named `"density"`, paramName would be `"density:pointCloudFile"`.

uniform <type> *paramValue*
: Value of the shader parameter.

This function returns a uniform float which indicates the total number of attributes the grid will handle.

`float VolumeTracer_GetParameters` (*uniform string* ***gridName****; ...*)

Retrieves some parameters of a previously created grid. The first parameter of this function, *gridName*, is the name of a grid previously created with `VolumeTracer_InitGrid`. The following parameters are an optional list of name/value pairs : each pair is composed of the name of a grid parameter to be retrieved, followed by a uniform variable of the appropriate type where the grid parameter's value is to be stored. The following grid parameters are supported:

'`nbattributes`'
: A float indicating the number of volume attributes managed by the grid (defined by the number of attributes returned by the grid's shaders). This is the number of attributes that will be returned by the `VolumeTracer_StepRay` function.

'`cellsize`' A vector indicating the size of the cells in each dimension.

'`nbcells`' A vector indicating the number of cells in each dimension of the grid.

This function returns the number of grid parameters successfully retrieved (might be 0 if none were requested[2]), or a negative value if the grid was not found or the function was called with invalid arguments.

`float VolumeTracer_InitRay` (*uniform string* ***gridName****; varying point* ***start****,* ***end****; varying float* ***stepLength****; uniform string* ***interpolation***)

Initializes a ray to be traced across the grid. Note that this function is SIMD so it will initialize many rays in practice. Parameters are:

gridName Name of the grid in which ray-tracing takes place.

start Starting position of the ray.

end Ending position of the ray.

stepLength Approximative length of steps to be made along the ray.

interpolation
: Type of interpolation to be used when computing the volume's attributes along the ray. The following values are accepted :

  '`trunc`' No interpolation - sample position is truncated to a grid vertex position

[2] This can be useful to simply test the existence of the grid.

'`nearest`' No interpolation - nearest grid vertex is used

'`linear`' Linear interpolation between surrounding grid vertices

'`cubic`' Cubic interpolation between surrounding grid vertices

'`exact`' No interpolation - the grid is ignored and the exact value is computed directly by the grid's shaders (very expensive; meant for low resolution testing only)

This function returns a varying float which is an identifier for the newly created ray. This identifier can then be passed to `VolumeTracer_StepRay_Function`.

`float VolumeTracer_StepRay (` *varying float* `rayID;` *output varying float* `length`)

`float VolumeTracer_StepRay (` *varying float* `rayID;` *output varying float* `length;` *output varying float*[] `attributes`)

Advances one step along a ray (in SIMD) and returns the volume's attribute midway along the step. All rays must belong to the same grid. Parameters are:

rayID Identifier of a ray previously created with `VolumeTracer_InitRay`.

length The length of this step.

attributes (optional)
The volume's attributes computed near midpoint along the current step, using the shaders specified to `VolumeTracer_InitGrid` and the interpolation mode specified to `VolumeTracer_InitRay`.

This function returns a varying float which indicates the fraction of the distance between the ray's start and end points that was covered until now. When 1.0 is returned, the end point has been reached and the ray cannot be advanced anymore.

`float VolumeTracer_EvaluateStep (` *varying float* `rayID;` *output varying float*[] `attributes`)

`float VolumeTracer_EvaluateStep (` *varying float* `rayID;` *output varying float*[] `attributes` *; uniform string*[] `shaders`)

Evaluates the volume's attribute midway along the previous step (in SIMD). All rays must belong to the same grid. Parameters are:

rayID Identifier of a ray previously created with `VolumeTracer_InitRay`.

attributes The volume's attributes computed near midpoint along the current step, using the density shader specified to `VolumeTracer_InitGrid` and the interpolation mode specified to `VolumeTracer_InitRay`.

shaders (optional)
The list of attribute categories that should be evaluated, those categories are identified by the name of the shader that computes them, as provided in the '`grid:shader`' parameter of `VolumeTracer_InitGrid`. By default, all attributes will be evaluated, but disabling a few of them can improve performance.

This function returns a varying float success (1.0) or failure (1.0).

10.2.7.3 Usage Example

Follows a code snippet showing how to use the API:

```
uniform string objectName = ... ;
uniform vector bboxCenter = ... ;
uniform vector bboxSize = ... ;
uniform vector nbCells = ... ;

uniform float nbAttributes;

/* Test grid existence */
if(VolumeTracer_GetParameters(objectName, "nbattributes", nbAttributes) < 0)
{
    /* Create grid */
    nbAttributes =
        VolumeTracer_InitGrid(
            objectName,
            "grid:center", bboxCenter,
            "grid:size", bboxSize,
            "grid:nbcells", nbCells,
            "grid:shader", "noisy",
            "noisy:intensity", 0.7);
}

/* Create ray */
float stepLength = ... ;
uniform string interpolation = "linear";
float rayID = VolumeTracer_InitRay(objectName, transform("object", P-I),
        transform("object", P), stepLength, interpolation);

float progress = 0.0;

float attributes[];
resize(attributes, nbAttributes);

while(progress < 1.0)
{
    float step;
    progress = VolumeTracer_StepRay(rayID, step, attributes);

    /* Use attributes ... */
}
```

The `noisy.sl` shader has to define a varying output array of floats, as in:

```
surface noisy(
    uniform float intensity;
    output varying float density[3]; )
{
    color D = noise( P ) * intensity;

    density[0] = D[0];
    density[1] = D[1];
    density[2] = D[2];
}
```

10.3 Linking with 3Delight

3Delight comes with a library which implements the RenderMan API interface. Here is how to link with it on different platforms.

IRIX `CC -o main main.cpp -I$DELIGHT/include -L$DELIGHT/lib/ -L/lib/i686 -l3delight -lm -ldl -lc`

Linux `g++ -o main main.cpp -I$DELIGHT/include -L$DELIGHT/lib/ -L/lib/i686 -l3delight -lm -ldl -lc`

Windows `cl /I"%DELIGHT%/include" "%DELIGHT%/lib/3delight.lib" main.cpp`

MacOS X `g++ -o main main.cpp -I$DELIGHT/include -L$DELIGHT/lib -framework CoreFoundation -lstdc++ -l3delight`

10.4 Attaching annotations to shaders

An annotation is some textual information (or a "meta data") indexed with a key (the *annotation key*). Annotations were primarily designed for better integration with software that need to build user friendly graphical interfaces and let the user modify shader's parameters gracefully. Such software usually need informations about ranges of each parameter in the shader as well as some description about how to present them to the user. A classical example is to present a toggle to the user when some shader parameters represent an on/off state.

Annotations are specified in the shader source with a `#pragma` instruction formatted as such:

```
#pragma annotation "<key>" "<text>"
```

Where '`<key>`' is the annotation key and '`<text>`' the annotation itself[3]. The preprocessor also supports C99's _Pragma operator:

```
_Pragma( "annotation \"<key>\" \"<text>\"" )
```

The advantage of this is that it can be used anywhere inside a macro to emit pragma directives. To perform substitution of macro parameters inside the pragma, a more advanced construct is required because substitution does not happen inside strings:

```
#define MAKE_STR2( str ) #str
#define MAKE_STR( str ) MAKE_STR2( str )

#define MAKE_PRAGMA( keysuffix, textdata ) \
    _Pragma( MAKE_STR( annotation MAKE_STR( key##keysuffix ) \
        MAKE_STR( annotation text with##textdata  ) ) )

MAKE_PRAGMA( special, a macro parameter. )
```

When run through the preprocessor, this results in:

```
#pragma annotation "keyspecial" "annotation text witha macro parameter."
```

There is no particular syntax for the annotation '`key`' or '`text`' fields and *3Delight* will not perform any syntax checking on these, it is up to the shader writer to decide the syntax depending on the software which is reading it. Follows an example shader annotated with informations about its parameters, author and version:

```
surface annotations( float Ks = 1.0; float has_shadow = 0; )
{
    /* code goes here */
}
#pragma annotation "author" "Aghiles Kheffache"
#pragma annotation "version" "1.0"
#pragma annotation "Ks comment" "Specular intensity"
#pragma annotation "Ks range" "[0..10]"
#pragma annotation "has_shadow type" "toggle"
```

Listing 10.4: An example annotated shader.

[3] When two annotations share the same key, their '`<text>`' field is concatenated.

A user interface builder could then draw a toggle for the '`has_shadow`' parameter since the shader writer specifically said that it was a toggle and not a simple scalar value.

Annotations in a compiled shader are accessible through the `shaderinfo` utility (Section 3.6 [Using shaderinfo], page 23) or through *3Delight* library calls (Section 10.2.5 [The Slo API], page 230). Here is the output of the `shaderinfo -a` command executed on the example in Listing 10.4:

```
% shaderdl annotations.sl
% shaderinfo -a annotations
surface test
5
"Ks comment" "Specular intensity"
"Ks range" "[0..10]"
"author" "Aghiles Kheffache"
"has_shadow type" "toggle"
"version" "1.0"
```

11 Acknowledgements

We would like to thank Moritz Moeller, Berj Bannayan, Paolo Berto, The `/* jupiter jazz */` group, Graeme Nattress, Jason Belec, Frederick Gustafsson, Yu Umebayashi, Daniel Moreno, Goran Kocov, Brian Perry, John McDonald and Alexandre Menard for their very helpful suggestions and bug reports. Also, thanks to Robert Coldwell, the creator of the wonderful 3Delighter and Ming Mah for the MacOS X framebuffer. And last, but not least, Jean-Jacques Maine, always on the front line when it comes to *3Delight* testing.

12 Copyrights and Trademarks

3Delight is RenderMan compliant in that it reads RenderMan RIB files and implements the RenderMan API.

```
The RenderMan (R) Interface Procedures and Protocol are:
Copyright @copyright{} 1988, 1989, Pixar. All Rights Reserved.
RenderMan (R) is a registered trademark of Pixar.
```

3Delight uses the libtiff library to read and write TIFF files. The libtiff license requires the following statements to appear in the documentation of the software:

```
The libtiff library is:
Copyright @copyright{} 1988-1997 Sam Leffler
Copyright @copyright{} 1991-1997 Silicon Graphics, Inc.

Permission to use, copy, modify, distribute, and sell the libtiff library
and its documentation for any purpose is hereby granted without fee,
provided that (i) the above copyright notices and this permission notice
appear in all copies of the software and related documentation, and (ii)
the names of Sam Leffler and Silicon Graphics may not be used in any
advertising or publicity relating to the software without the specific,
prior written permission of Sam Leffler and Silicon Graphics.
```

3Delight uses the jpeg library from the IJG. It requires that we mention that:

```
This software is based in part on the work of the Independent JPEG Group.
```

The Windows version of *3Delight* uses the regex library. The regex license requires the following statements to appear in the documentation of the software:

```
The regex library is:
Copyright @copyright{} 1992 Henry Spencer.
Copyright @copyright{} 1992, 1993 The Regents of the University of California.
All rights reserved.

The regex library code is derived from software contributed to Berkeley by
Henry Spencer of the University of Toronto. Redistribution and use in source
and binary forms, with or without modification, are permitted provided that
the following conditions are met:

1. Redistributions of source code must retain the above copyright
notice, this list of conditions and the following disclaimer.
2. 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.
3. All advertising materials mentioning features or use of this software
must display the following acknowledgement:
```

```
This product includes software developed by the University of
California, Berkeley and its contributors.
4. Neither the name of the University nor the names of its contributors
may be used to endorse or promote products derived from this software
without specific prior written permission.

THE REGEX LIBRARY IS PROVIDED BY THE REGENTS 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 REGENTS 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.
```

3Delight uses ILM's OpenEXR library to read OpenEXR files. The license requires the following to appear in the software:

```
Copyright (c) 2002, Industrial Light & Magic, a division of Lucas
Digital Ltd. LLC

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 Industrial Light & Magic 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.
```

Concept Index

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

Function Index

N

O

P

R

S

T

V

X

Y

Z

List of Figures