Slim

Slim is a powerful, extensible tool for creating, manipulating, using and reusing RenderMan shaders.

Slim brings the power of RenderMan shaders into the hands of artists. Users can construct RenderMan shaders by interactively combining modules into networks, editing their parameters, and previewing the results. This is all possible without writing any code.

The following pages continue with introduction. To jump to a specific section of the documentation, select a page on the left.

Slim

Slim is a powerful, extensible tool for creating, manipulating, using and reusing RenderMan shaders.

Slim brings the power of RenderMan shaders into the hands of artists. Users can construct RenderMan shaders by interactively combining modules into networks, editing their parameters, and previewing the results. This is all possible without writing any code.

The following pages continue with introduction. To jump to a specific section of the documentation, select a page on the left.

Using the Documentation

Select a section or a specific page using the index to the left.

About Slim

The About Slim section introduces basic concepts and includes the page you are reading now.

User Interface

This section describes the components of the Slim user interface, and gives specific information about using them.

User Interface

The Organization and Reuse section describes Slim features that help you effeciently use and reuse shaders and their components.

Slim Scripting

The Slim Scripting section introduces how to control Slim via scripts, and includes a complete glossary of the procedures and methods available in the Slim scripting environment.

Slim Templates

This section discusses Slim's template system, which allows you to modify, add, and customize modules that define shaders created using Slim.

Slim Files

The Slim Files format gives details on `.slim` and `.splt` files, their contents and how Slim handles special cases.

Slim Terminology

Appearances

Appearances refer to the nodes in Slim used to construct and represent RenderMan shaders. RenderMan shaders imported into Slim are represented by a single appearance. Shaders constructed in Slim consist of a network of appearances, also called **functions**.

Templates

Templates define the set of functions that can be combined by users to create shaders. Templates are Slim extensions that are loaded by Slim at startup.

Properties

Every appearance includes a set of properties that define its characteristics. Properties come in many forms, such as shader **parameters**, RIB **attributes**, and property **collections**.

Extending Slim

Slim is designed to be extended and customized. The building blocks of shaders are defined using *templates*, programmable modules with RenderMan Shading Language at their core. These templates are extensions to Slim and are ready to be modified, replaced, or combined with new templates in order to meet the needs of your studio.

You can automate actions by writing scripts for Slim. Scripting has a variety of uses including generating new shaders, analyzing and modifying existing ones, or simply providing a richer experience when writing templates.

Scripts can be developed interactively within the Slim *console*. These scripts can also be loaded at startup, and even invoked by users via custom menu items.

For more information, see these sections of the documentation:

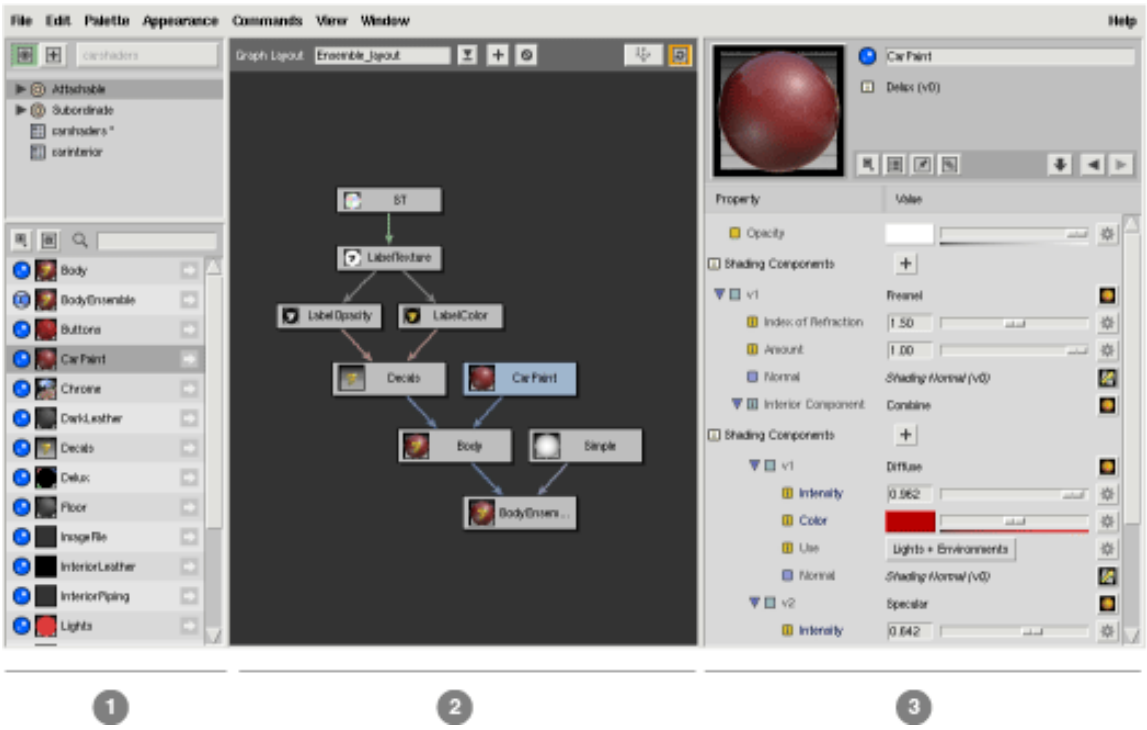
Slim Scripting

Slim Templates

User Interface

The Slim Window

The majority of your operations you perform in Slim will be within one window:



This window is divided into three main sections:

1. The Palette View
2. The Network View
3. The Appearance View

These views all work together to provide increasing levels of detail: The Palette View displays all of your active palettes and their contents. The Network View displays how appearances in those palettes combine to form shaders. The Appearance View displays parameters and a preview for a specific appearance within a network.

These views are window panes, and their relative size can be changed by dragging the mullions (or "sashes") between them. The entire window itself can be resized as well.

Menus

Slim can be invoked in different modes. Some of the menu items described below are not always visible, depending on whether Slim is invoked in *client* or *standalone* mode, and depending on whether Slim is being used as a piece of RenderMan Artist Tools (RAT) or RenderMan Studio (RMS).

File Menu

The File menu contains functions related to reading and writing files.

New Palette

Create a new palette file and set it as the current palette

Open Palette

Open an existing palette file and set it as the current palette

Save Session (RMS only)

Save the current session file

Save Palette

Save the current palette file

Save Palette As...

Save the current palette to a new filename

Revert Palette...

Revert the current palette to a previously-saved version

Remove Palette (when working with a session / scene file)

Remove the current palette from the existing session / scene file

Import Appearance...

Import an appearance or shader from a file

Export Appearance...

Export an appearance to a file

Reload

Reload the template or shader for the current appearance

Upgrade Template

Upgrade the template for the current appearance. This might be necessary if you have loaded a palette with templates from a previous version of Slim.

Quit *(Only when used with RfM or in standalone mode)*

Quit Slim

Edit Menu

The Edit menu contains commands related to editing:

Undo

Undo the last action performed

Redo

Redo the last action undone

Select All

Select all appearances in the current palette

Delete

Delete the selected appearance(s)

Revert Values

Revert all of the parameter values for the selected appearance to their defaults

Preserve Values

Freeze all of the current parameter values for the selected appearance. This will protect those values from any changes to the values in the appearance's source.

Palette Menu

The Palette menu provides commands that operate on the current palette.

Create Subpalette

Create a new subpalette in the current palette. You can drag/drop appearances and other

subpalettes into and out of this subpalette.

Create Smart Palette

Create a new smart palette within the current palette. This will open the Smart Palette Editor, enabling you to select the criteria for your new smart palette.

Edit Library Palette

Edit the contents of a selected Library Palette.

Close Package

Close the current package.

Saved Icon Settings

Change the settings for saving icons in the current palette: The first set of radio buttons controls which appearances within the palette will have their icons saved. The second set controls the resolution of those icons. Saving icons within a palette increases its file size, which can affect the speed at which the palette is loaded.

Uniqueify All

Reassign the unique identifiers for all nodes within the palette. This is necessary after copying a palette file to make sure its identifiers do not conflict with those in the original.

Delete Unused *(only when used with MTOR)*

Delete any appearances that aren't in use by MTOR.

Delete Disconnected

Delete any non-attachable appearances that aren't connected to any attachable appearances.

Appearance Menu

The Edit menu contains commands related to editing:

Attach *(MTOR only)*

Attach the selected appearance to the object picked in Maya

Detach *(MTOR only)*

Detach the selected appearance from the object picked in Maya

Pick Objects *(MTOR only)*

Pick the objects in Maya that have the selected appearance attached

Pick Objects (Calculated) *(MTOR only)*

Pick the objects in Maya that have the selected appearance attached in the current adaptor context

Add to Scene *(RfM only)*

Add the selected shader to the current scene

Render Icon

Render the preview swatch for the selected appearance

Render Flipbook

Render a flipbook for the current appearance

Select

This provides a set of commands allowing you to select instances, upstream or downstream appearances for the selected appearance.

Duplicate

Duplicate the selected appearance.

Duplicate Network

Duplicate the selected appearance and all upstream appearances.

Create Instance

Create an Instance of the selected appearance.

Create Shader Instance

Create a Shader Instance of the selected appearance.

Package

Enclose the selected appearances in a Package.

Unpackage

Extract the contents of the selected package and delete it.

Show Package Contents

Show the contents of the selected package.

Edit Package Parameters

Edit the Parameter interface for the selected package.

Edit Master

This provides a pair of commands that allow you to *globalize* (give an absolute path to) or *localize* (give a relative path to) the shader generated by the selected appearance.

Commands Menu

The Commands menu lists a set of procedures that can be customized and configured. Information on creating these commands is available in the Scripting documentation.

By default, a few commands are available for the current palette:

Recompile All

Recompile all of the shaders in the current palette.

GenerateReport

Generate a verbose report providing copious amounts of data. This report enumerates external resources and reports warnings. The report is printed to the Slim Message Logger.

GenerateReport (Short)

Generate a short version of the above report enumerating warnings. The report is printed to the Slim Message Logger.

String Search & Replace

A general search and replace command. The procedure used by this command is also accessible from the scripting environment using the ::Slim::SearchReplace procedure.

Instance All

Instantiate each of the current templates. This can be useful procedure for testing your set of templates.

View Menu

The View menu controls visibility of various elements in Slim.

Toggle Palette View

Toggle the Palette View between Browse and Create mode.

Show Network View

Toggle the visibility of the Network View.

Maximize Network View

Maximize the Network View by temporarily hiding the Palette View.

Two-Column View

Arrange the Slim Window in a Two-Column View.

Three-Column View

Arrange the Slim Window in a Three-Column View (the default).

Show Appearance Notes

Toggle the visibility of notes within the Appearance View.

Show Appearance Tags

Toggle the visibility of tags within the Appearance View.

SL Source *(Only with Expert Menus enabled)*

Display the shading language source for the selected appearance in a text editor.

Slim Source *(Only with Expert Menus enabled)*

Open the Slim source for the selected appearance in a text editor. This is how the appearance will be written into the palette file. Any changes made and saved in the text editor will be incorporated back into the appearance after you close the editor.

Window Menu

The Window menu gives you access to all of the other windows of Slim.

Workspace Editor *(RAT only)*

Open the Workspace Editor

Message Log

Open the Message Log

Console

Open the Slim Scripting console

Preferences

Open the Preferences editor

What follows is a list of windows for any appearances that are being edited in a separate window (if any).

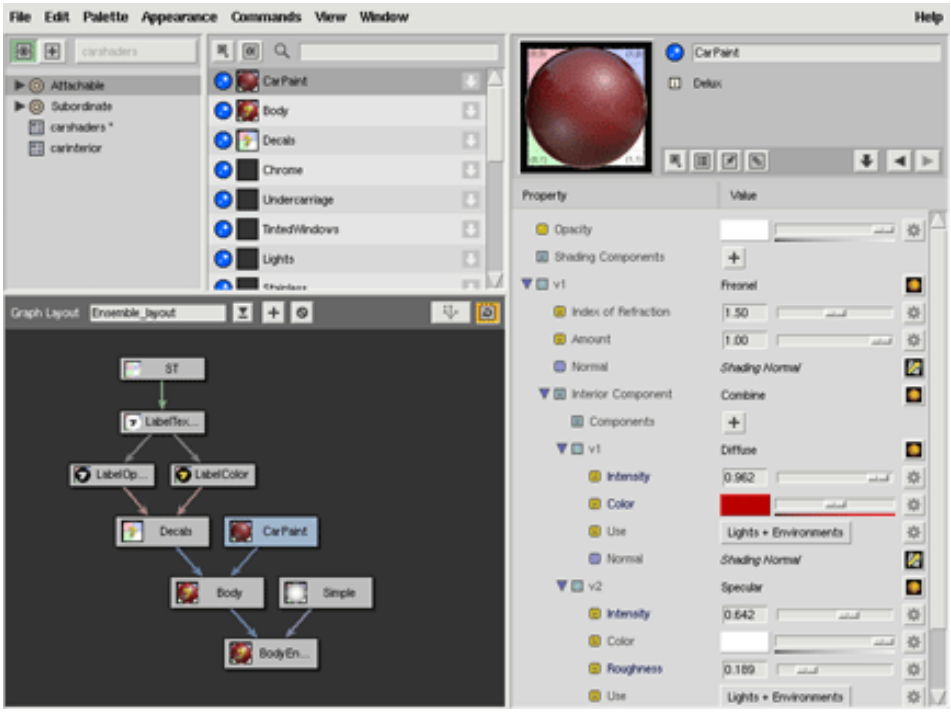
Help Menu

The Help menu contains an **About Slim** command that displays build and version information as well as links to this documentation and the RenderMan forums.

Because this menu can be customized, you may find additional links to documentation specific to your studio.

Two-Column View

With the Network View visible, the Palette View and Network View can also be arranged in a "Two-Column" View. Depending on your palette, networks, and monitor, you may find this arrangement a better use of space. It can be selected from the View Menu.



Prev | Next

The Palette View

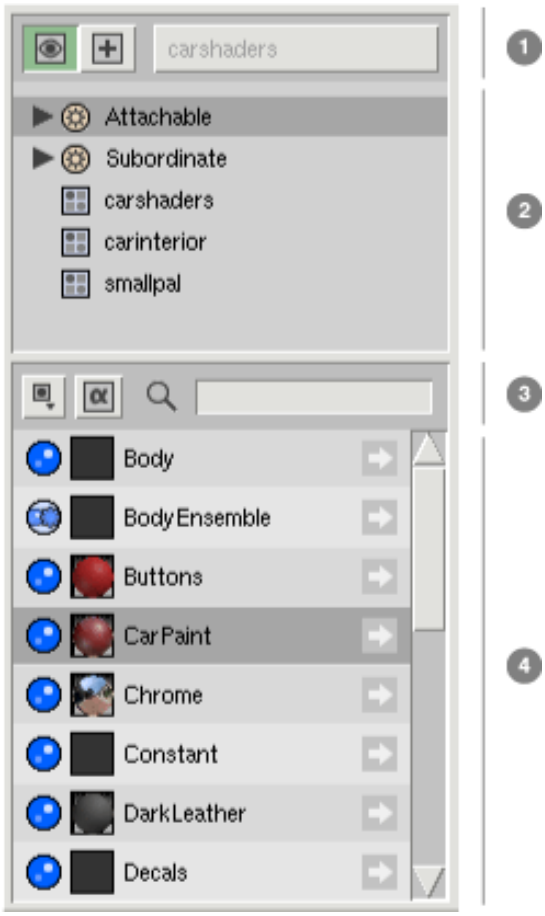
The Palette View is where you will manage palettes and packages. It allows you to quickly see the contents of each palette in your session, and, when using *smart palettes*, to navigate and refine lists of all active appearances.

The Palette View is also a creation tool. When placed in Create Mode, the palette view allows you to navigate the templates that you can use to create new appearances.

The Palette View consists of two sections. The top section displays a hierarchical list of active palettes, subpalettes, and packages. The bottom section displays the contents of the selected palette.

Here is a quick survey of the pieces that make up the Palete View:

1. **Palette Listing Controls**
   These control the Palette View Mode and Current Palette.
2. **Palette List**
   The list of active palettes, subpalettes, smart palettes, and packages.
3. **Content Listing Controls**
   These include a control over the size of thumbnails, a toggle for sorting alphabetically, and a field to use to filter the contents by name.
4. **Palette Contents**
   Contents of the selected palette.



Browsing Palettes

By default, the Palette View is in *Browse* mode, enabling you to browse the existing appearances in your palettes. Selecting an appearance displays the network for the shader(s) it affects in the Network View, and displays parameters and other details in the Appearance View.

Each time you select an appearance, the Network View will update to show the graph layout containing that appearance. To add an appearance to the current graph layout, use the arrow button to the right of the appearance listing.

Current Palette

Working with multiple palette files presents some ambiguity when it comes to operations like Save and Close. For this reason, the **Current Palette** is always displayed at the top of the Palette View, within the Palette

Listing Controls.

The Current Palette is updated whenever you select a palette file in the Palette Listing. Also, whenever you select an appearance, the palette containing that appearance is set to be the Current Palette.

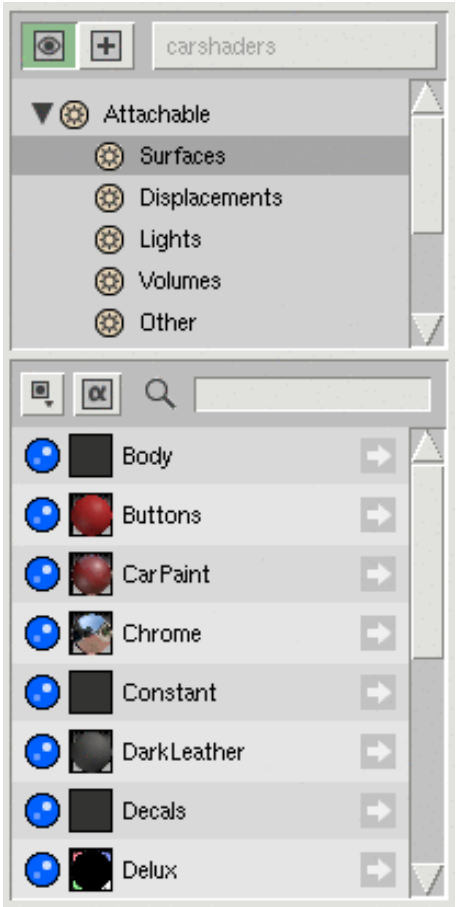
Browsing with Smart Palettes

Often, rather than seeing all of the appearances within a file, you will instead wish to see all of the appearances (in all palette files), that share a common quality (such as attachment type). *Smart Palettes* provide this capability.

Listed above your palette files are your current set of smart palettes. Select a Smart Palette, and it will display all of the appearances that meet that smart palette's configured criteria.

Smart Palettes can be *nested*, which provides for refinement within criteria. Select the **Attachable** Smart Palette, and all appearances that can be attached to geometry will be displayed. Select the **Surfaces** Smart Palette (shown on the right), and the list of appearances will be refined to only those appearances that make attachments of type "surface."

The set of session-wide Smart Palettes are saved as preferences and can be reconfigued and rearranged. For more information, refer to the Smart Palettes documentation.

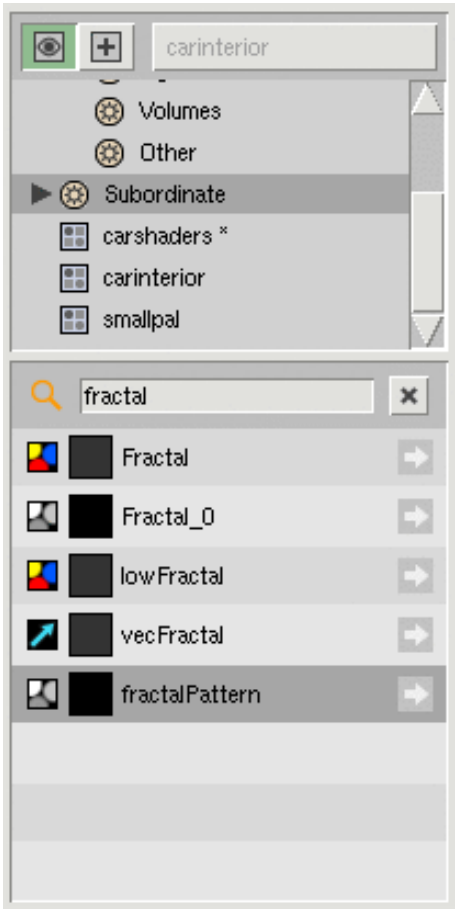


Filtering Contents

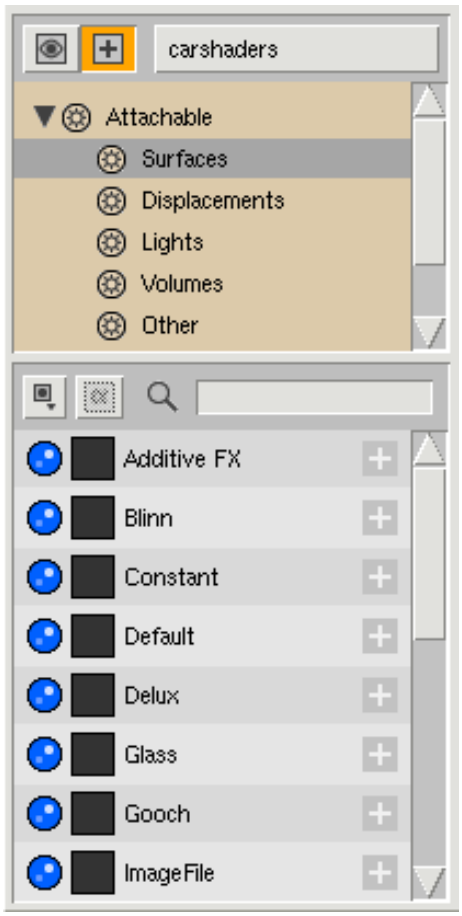
You can quickly filter the appearances displayed by their label using the Filter filed in the Content Listing Controls.

To filter contents, simply start typing in the entry field. The contents will update for each key you press.

Tip #1**: You can access this filter at any time using the forward-slash key: **/
***Tip #2**: You can clear the filter using the Escape key*



Create Mode



To create new appearances, place the Palette View in *Create* mode by pressing the **+** button in the Palette Listing Controls section. To remind you that you're in Create mode, the Palette Listing will adopt an orange background.

Now, the smart palettes in the palette listing can be used to navigate **templates** rather than loaded appearances. They work in the same way: select the *Surfaces* smart palette and you'll see the list of templates that create Surface appearances.

In Create mode, the Current Palette indicator becomes a pulldown menu. Use it to select the destination palette for the appearances you create.

To instantiate a template, simply select it. Selecting an appearance will add it to the current graph layout. Using the template icon's **+** button, however, will create a new layout in the Graph View for the appearance.

As in Browse mode, you can quickly find a template by using the Filter field.

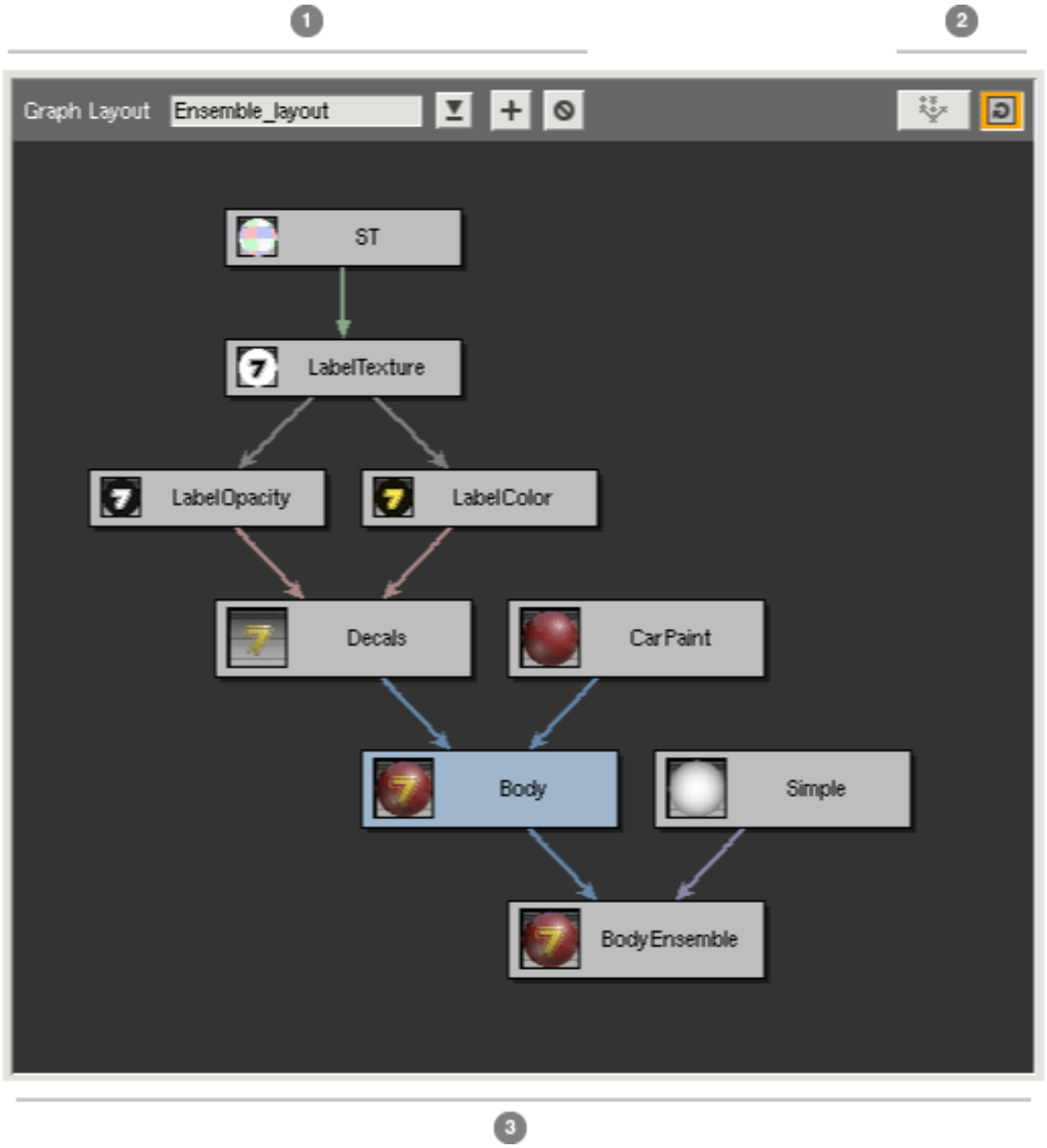
Tip

You can quickly toggle between *Browse* and *Create* mode using the backtick key: ` (also known as a grave accent).

The Network View

The Network View is where you can understand how your appearances interact to create a shader. The Network View is always providing context about where you are in a shader's network and how the changes you make affect downstream appearances.

The Network View also allows you to change the construction of your network by making and breaking connections between appearances. The Network View consists of these components:



1. **Graph Layout Controls**
   Control the Graph Layout being displayed for the current selection.
2. **Arrangement Controls**
   Control whether appearances are arranged automatically or manually.
3. **Graph Canvas**
   This is where nodes for the current network are displayed

Graph Layouts

Whenever any nodes are visible in the Network View, it is displaying the contents of a *Graph Layout*. The Graph Layout tracks the appearances being displayed, the current focus, and how nodes within the layout are arranged.

Whenever an appearance is selected (or created), the Graph Layout containing that appearance is loaded into the Network View. If no Graph Layout exists, one is created. The name of the current Graph Layout is displayed in the Graph Layout Controls. A Graph Layout's name is automatically chosen when it is created. You are free to change this name.

Next to the name field for the Graph Layout is a pulldown menu. When the selected appearance exists in multiple layouts, you can use this pulldown to select which layout to display.

The last set of buttons in the Graph Layout Controls allow you to create a new Graph Layout or delete the current one.

Arranging Nodes

Every Graph Layout has an arrangement mode. By default, a Graph Layout is configured for **automatic** arrangement. In this mode, Slim automatically arranges the appearances in a network. Any changes to the network will cause the layout to be rearranged in an optimal manner. Automatic arrangement is indicated by the illuminated toggle on the right of the Arrangement Controls.

If you prefer to arrange the appearances in a layout yourself, disable Automatic Arrangement using the aforementioned toggle. With Automatic Arrangement disabled, you can position nodes as you wish. If you wish to invoke Slim's arrangement method, press the Arrange Graph button on the left of the Arrangement Controls.

Navigation

To pan around a graph layout, hold down **Ctrl** and the middle mouse button and drag. To zoom in and out of a graph, hold down **Ctrl** and the left and middle mouse buttons while dragging left and right. If your mouse is equipped with a scrollwheel, you can also use this to zoom in and out of a graph layout. *(Except on OS X, where the Command key is unavailable, these transport controls should be the same as those in Maya.)*

You can use the **F** key to focus on the select appearance. This will pan the layout until your selection is visible.

Creating Appearances

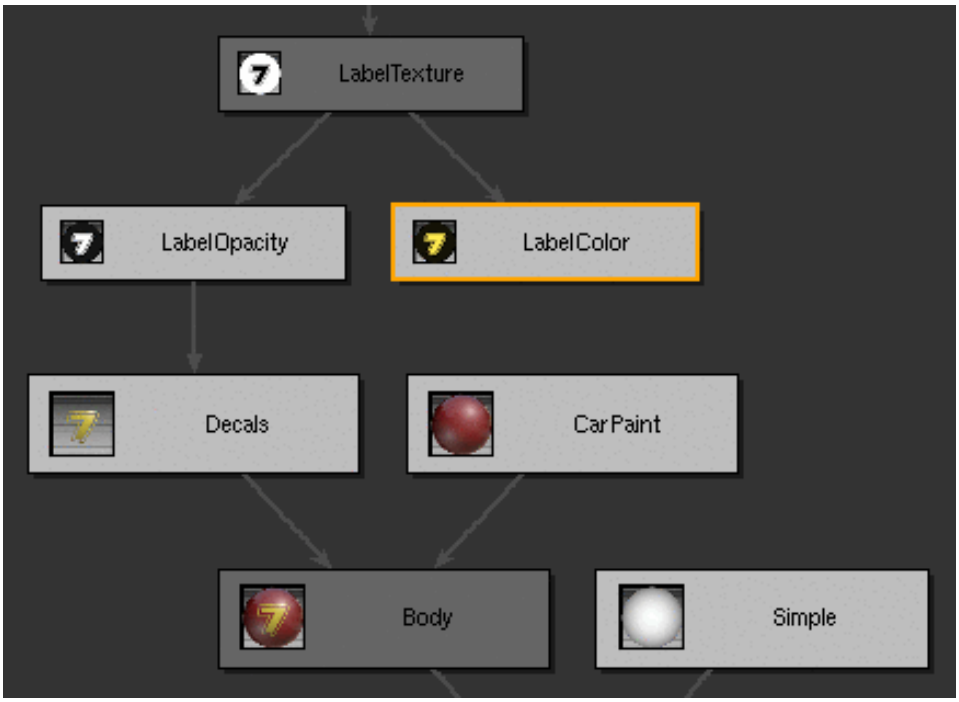
There are several ways to create new appearances in Slim. Any of these methods will affect the Network View.

- If you use the **Palette View** in Create Mode, any new appearances will be given a new Graph Layout unless you use the  button to explicitly add the appearance to the current layout.
- If you create a connection in the **Appearance View** via the Property Menu, the new appearance will automatically appear in the current layout.
- You can invoke the Graph Context menu by holding down the right mouse button on the background of the graph canvas. This will present a few graph commands, along with a menu of possible apppearances

to create. Appearances created in this method will automatically be added to the current layout.

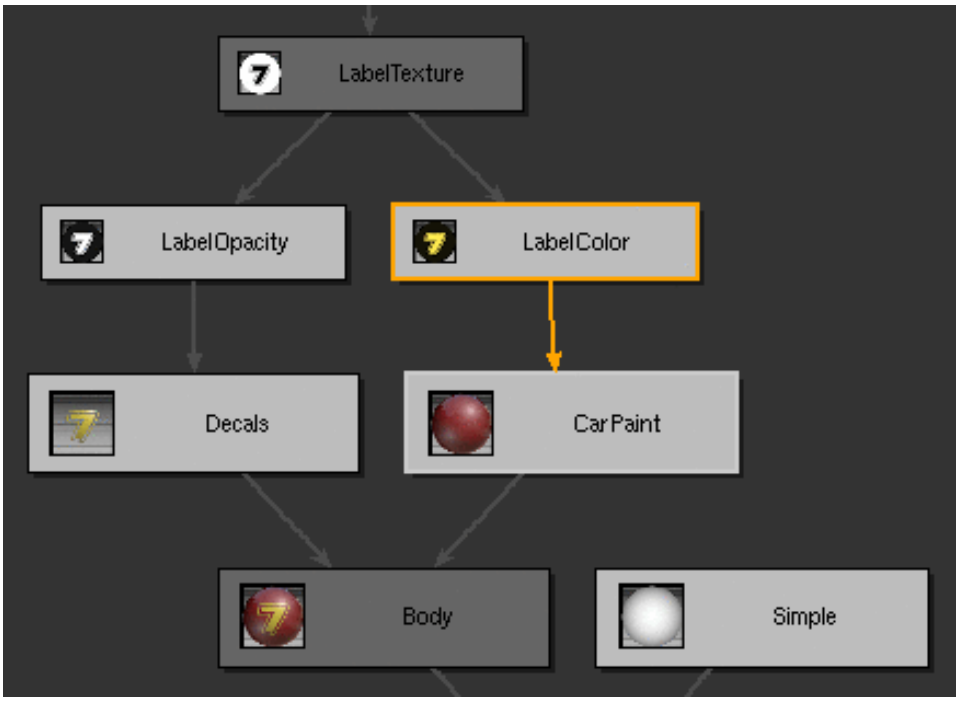
Connecting Appearances

You can use the Network View to connect the output of an appearance to the input of another. To begin a connection, click on the source — or *upstream* — appearance with the middle mouse:



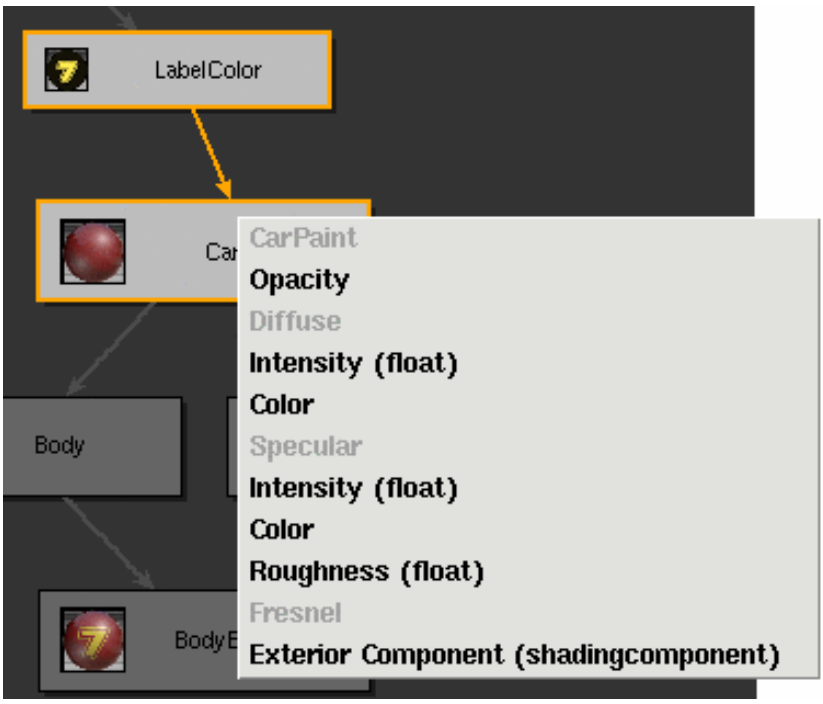
The selected appearance will be illuminated in orange. If the appearance has multiple outputs, you will be presented with a menu from which to select the output to connect. Otherwise you can begin selecting the destination — or *downstream* — appearance.

All valid downstream appearances will remain brightly lit. Any appearances that are dimmed cannot accept a connection from your upstream appearance.

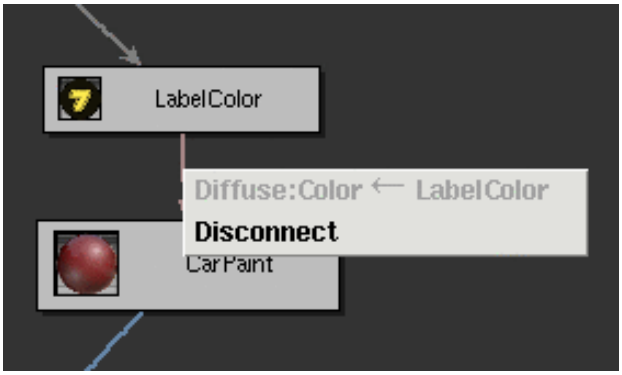


Move the mouse to drag the connection to your destination appearance of choice. Note that you do not need to continue holding the mouse button; this allows you to pan or zoom the graph as you search for your downstream appearance. To cancel the connection, simply click on the background of the graph canvas.

To complete the connection, click the middle mouse button again on your selected downstream appearance. You will then be presented with a menu of inputs in the downstream appearance. Select from the menu the parameter you wish to connect and the connection is made.

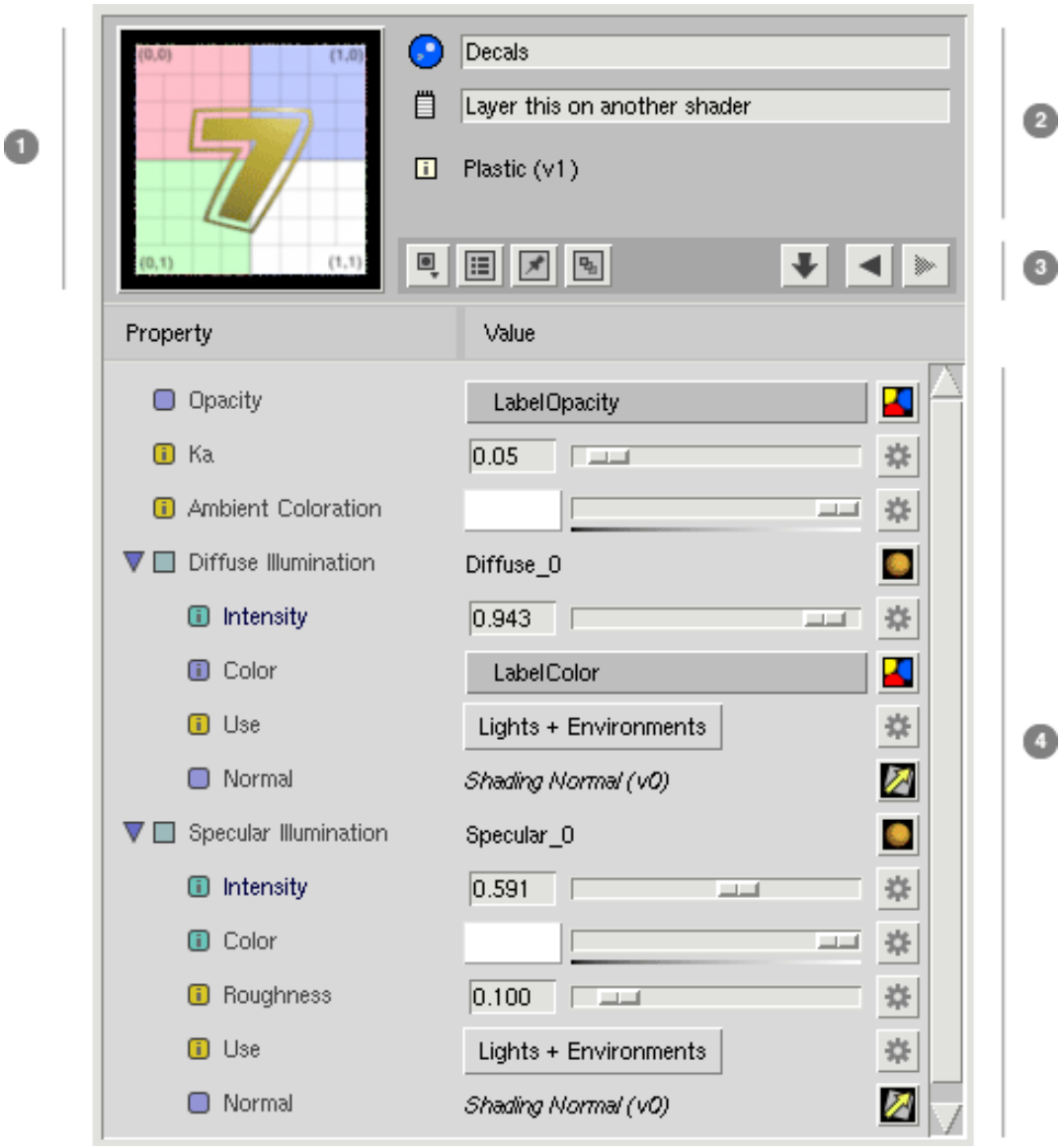


If you wish to break a connection, select the edge representing that connection with the middle mouse. This will present you with a menu with which you can disconnect:



The Appearance View

The Appearance View is where you will edit the properties associated with a specific appearance. The Appearance View also allows you to preview how your shader will look when rendered.



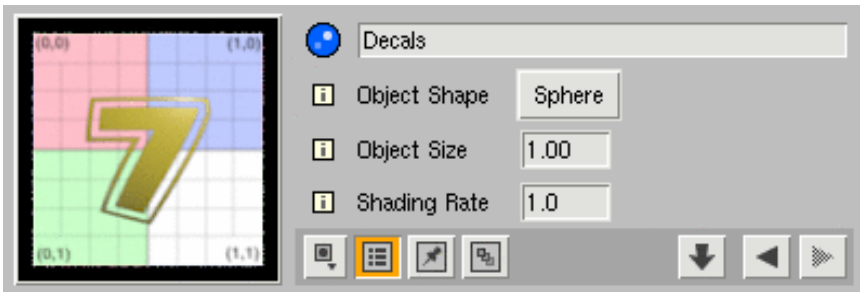
The Appearance View interface can be divided into these sections:

1. **Preview Swatch**
   This swatch gives you a preview of your shader using Slim's embedded RenderMan renderer. Click the swatch to update it.
2. **Appearance Information**
   The name of your appearance is always displayed here. In this example, we also see Notes associated with this appearance and the template used to create it.
3. **Settings and Navigation**
   This bar contains settings for the Appearance View along with buttons for navigating among appearances.
4. **Appearance Properties**
   This area displays the properties of your appearance and an editor for each.

Preview Swatch

The Preview Swatch allows you to preview your shader on a simple piece of geometry using Slim's embedded RenderMan renderer.

The swatch renders using a number of settings defined in your Preferences. You can override these settings for a particular appearance by revealing the Preview Render Settings, accessed using the toggle illuminated below:



This gives you access to these settings:

Object Shape

The shape used during preview rendering. Choose from among: Sphere, Cylinder, Torus, Plane, Cube, Teapot.

Object Size

The approximate size of the preview shape. The camera is automatically repositioned so that the apparent size of the ***geometry*** on screen is the same. Use this control to get a sense of the scale of various ***patterns*** inherent in your appearance.

Shading Rate

Ae basic quality / speed tradeoff control. Higher numbers render faster but at a lower quality level.

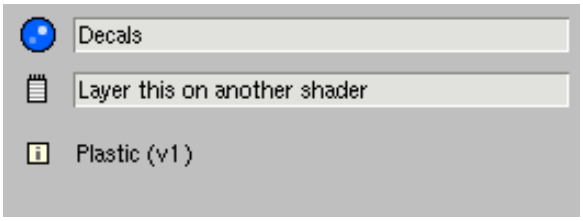
Frame

The current frame for the purposes of preview rendering. Useful if your shader is dependent on `time`. *(By default, this setting is hidden. To reveal it, enable it in Preferences.)*

Appearance Information

This section gives you information about the current appearance. The appearance's **type**, displayed using an icon, and **label**, displayed as a text entry field are always visible. You can change an appearance's label here.

What follows depends on current settings and the current appearance. Preview Settings are displayed here when enabled (as shown above). Appearance notes and tags may also be visible, depending on options set in the View menu. In the example below, appearance **notes** are shown.



Lastly, we see the appearance's **source**. The type of information displayed depends on the class of appearance you are viewing:

- A **function** instantiated from a template will list the name of the template here (see above), along with a help icon giving you access to the template description. Users who have enabled Expert Menus will also see the version of the template. When the template used is an old version that requires upgrading, this field will appear in red.
- An **instances** will list its orignal appearance here in the form of a button. This button will take you to the original appearance.
- An **imported shader** will list the name of the RenderMan shader – or *master* &ndash here. The accompanying button will toggle this between a *relative* and *absolute* reference.

Settings and Navigation

This bar contains menus and switches that affect your appearance, along with buttons to navigate among the appearances in your palette:


Preview Resolution
Change the resolution of the Preview Swatch


Preview Settings
Toggle display of Preview Settings


Pin
Toggle Appearance Pinning


Overview
Toggle Parameter Overview


Downstream Appearances
Navigate to a *downstream* appearance (displayed in a menu)

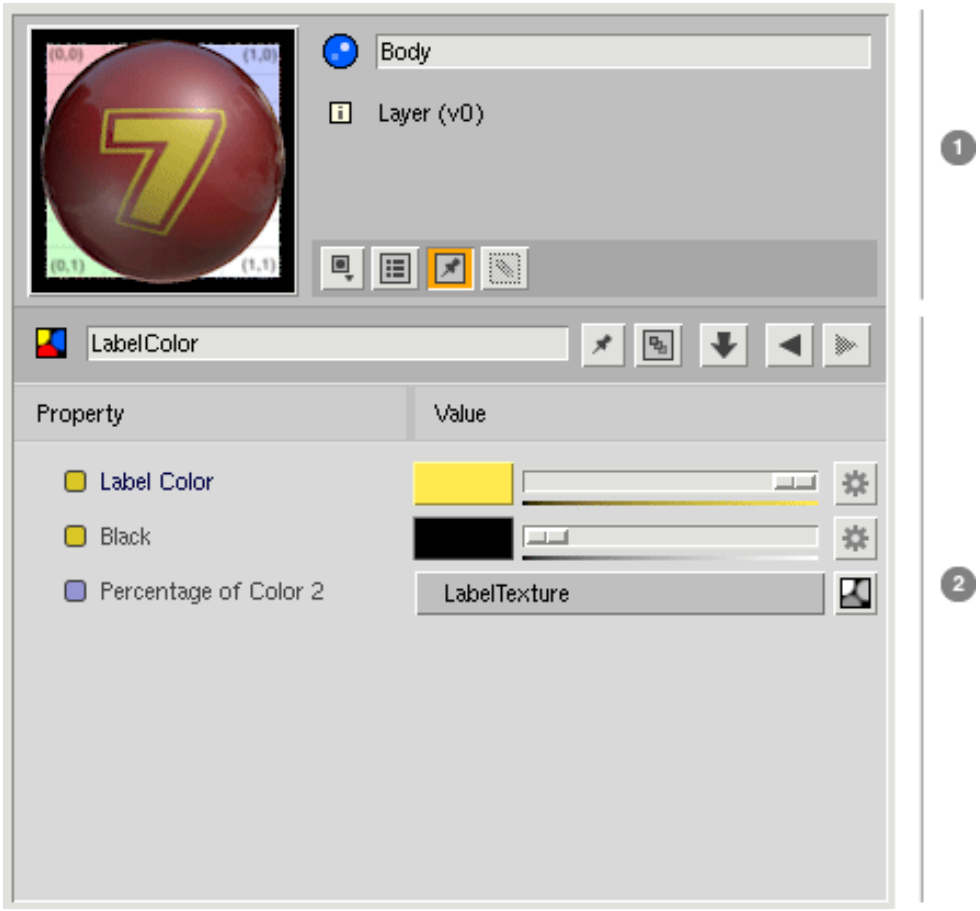

Back
Go back to the previous appearance


Forward
Go forward (enabled aafter using **Back**)

Pinning an Appearance

When editing appearances in a network, you may find you wish to edit the parameters for one appearance and see the result by previewing another. This is made possible by *pinning*.

Navigate to the appearance with the preview you wish to pin. Then select the Pin toggle. The Preview Swatch will now be locked to this appearance, and the Appearance View will be reconfigured:



With pinning enabled, the Appearance View is divided into two halves:

1. **The Pinned Appearance** This half of the Appearance View will not change as you select appearances. It will always show you the Preview Swatch for this appearance.
2. **The Current Appearance** This half will continue to update as you select appearances. Note that the navigation controls have migrated to this half, and that the current appearance is displayed.

Appearance Properties

Every appearance contains a number of properties. These are the elements of how a typical property is displayed:

1. **Property Icon / Description**
   This icon indicates the class of property displayed and its provider. The presence of an **i** indicates that a description of that property can be accessed by clicking the icon. Double-click to edit the description.
2. **Property Label**
   This is the name of the property that is being edited. Double-click this text to change it. The label will change color when a value is set for a parameter.
3. **Value Editor**
   This is an editor specific to the property's type.
4. **Property Menu**
   This menu displays a list of actions and options for your the property. The menu is specific to the class

of property displayed.

Classes of Properties

The class of a property is indicated by the icon to left of a property's name. These are the classes of properties with values that can be edited:

Parameter
By far the most common class of property, parameters of appearances correspond to parameters of shaders and/or shader functions. Note that the color of this icon will vary, depending on the parameter's value provider.

Slim Attribute
Slim Attributes are guaranteed to be constant, and will not be visible within a shader. It is for these reasons that they are useful for controlling the code generation within a DynamicFunction.

RIB Attribute
RIB Attributes are RenderMan settings that are specific to your shader. These attributes and their values will be represented in the RIB stream when the enclosing appearance is encapsulated into the RIB format. Sample RIB Attributes include:

- displacementbound
- shadingrate
- sides:doubleshaded

Much of RenderMan's ray-tracing functionality is controlled via RIB Attributes (in the Ensemble template, for example).

TOR Attribute *(MTOR only)*
TOR Attributes affect jobs created by MTOR. Map Generator templates, for example, contain TOR Attributes to control how and when those maps are generated.

Collections

Technically another class of property, collections provide a grouping mechanism for other properties. Collections can serve several different purposes:

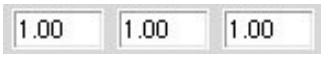
- **Grouping Similar Properties**
  The most common use of collections is simply to simplify a user interface by hiding properties within the collection. Users can open or close the collection using a disclosure widget (▶) and its state is maintained across Slim invocations.
- **Representing Arrays**
  Slim uses the collection class to represent arrays. Individual array elements appear as parameters within such a collection.
- **Representing Composite Function Types**
  The fundamental types for parameters are described below. Using a collection, these types can be combined into a composite type to provide a single connection point. The "shadingmodel" type, which contains a color and an opacity value, is represented by a collection. When a collection is used in this manner, it will appear to the user as a parameter.
- **Defining Custom Interfaces**
  A template may override the standard interface for a set of parameters, and instead declare a custom user interface or *Custom UI*. A collection is used to define the Custom UI that should be used and the set of properties that should be represented.

Value Editors

Properties come in several different types. The Appearance Editor presents a different editor for each of these types. The most common types and the editors displayed are presented below:

float
A single floating-point value, floats are edited using a widget which features both a numerical entry (or "VSlider") and a graphical slider.

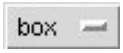
color
An RGB color, colors are edited using Slim's Color Editor, which is raised by clicking the color chip. Additionally, the intensity of a color can be quickly adjusted using the accompanying slider.



point / vector / normal
These three types represent geometric entities in three dimensional space. The X, Y, and Z components are each represented with a VSlider.

string
Plain old strings are generally not that interesting, though should you want to edit one, an editor is provided. Far more interesting are string subtypes (below).

Some properties may request an alternative editor for a given type using a *subtype*. Here are some common property subtypes (and applicable types):



selector
The selector subtype can be used for any type to define a finite list of possible values.

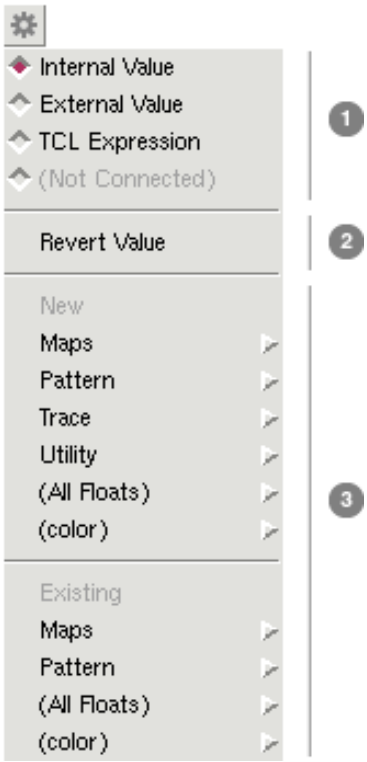
switch (float)
This presents a simple on/off switch that represents values of 1 and 0.

bigstring (string)
A string too long to be edited in one line, the bigstring subtype relies on an external text editor to edit its value.

texture (string)
A string representing the path to a texture file, the texture subtype (as well as similar subtypes for reflection, environment, shadow, etc.) gives access to a file picker, and to texture conversion tools.

The Property Menu

The property menu combines several settings and commands for your property. There are three main sections of the property menu. Depending on the traits of the property, the menu may display some or all of these sections.

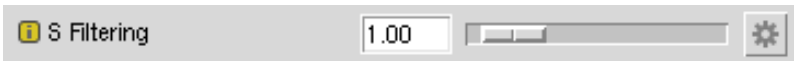


1. **Value Provider**
   This section controls the Value Provider for your property. If the value provider cannot be changed for your property (because, for example, it is a Slim Attribute, or because it must be connected), this menu will not be displayed.
2. **Commands**
   This section displays a set of commands that are relevant to the property and to its Value Provider.
3. **Connections**
   Some properties can have their value provided by another function. For these properties, a list of possible connections to make is displayed.

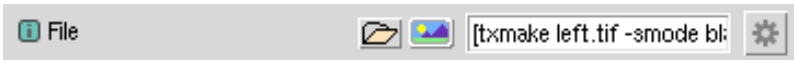
Value Provider

The default widget associated with each parameter is only one way to set its value. The value of a parameter can actually come from a number of different places and the Value Provider section allows you to choose this.

Different properties have different combinations of provider options available. The color of the property icon indicates a property's provider.



By default, the value of a parameter will usually be stored as a constant, **Internal Value**. This means that the value will be hard-coded into the shader and cannot be changed without regenerating the shader.



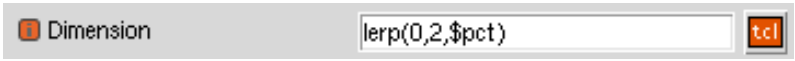
A parameter with a teal icon indicates that it has an **External Value**. External parameters are declared as parameters of the shader allowing their values to change without regenerating the shader. When flattening, or instancing a shader, only parameters with external values will be shown.

The most powerful Value Provider is **Connection**. Setting the Value Provider to Connection causes the property icon to turn purple and displays the connection type in the property menu. The value of the parameter is provided by the output of another function. Because each function can have parameters with values provided by other functions, you can build up powerful networks of functions interconnected through their parameters.

Note that the name of the connection appearance is displayed within a button. You can use this button to navigate to the connected appearance. Button-1 will edit the appearance in the current editor. Button-2 will edit the appearance in a new editor.

Some parameters may specify that their value must be provided by another function and will specify a default function to connect to. In this case, the Manifold parameter has

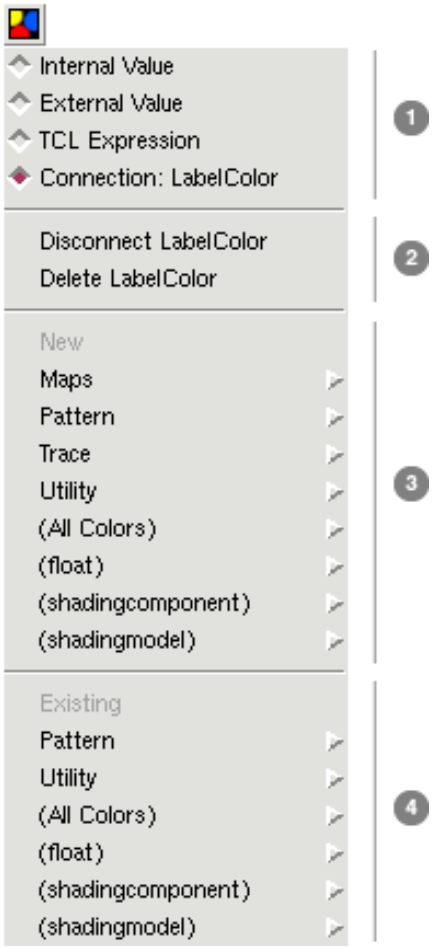
specified that by default, it should connect to a SurfacePoint function. You can override this default and connect the parameter to a different function, but the parameter must always be connected to something.



MTOR only:
Sometimes expressing the value of a parameter via a static value isn't sufficiently powerful. You might want to, for example, request that the value change from frame to frame via a TCL expression. Setting the Value Provider to **TCL Expression** causes the icon to turn orange and presents a different entry widget where you can enter a numerical expression. The property menu will contain commands to control and test your expression. For more details see documentation on Parameter Expressions.

Connections

This is how the Property Menu appears when a parameter receives its value from a connection:

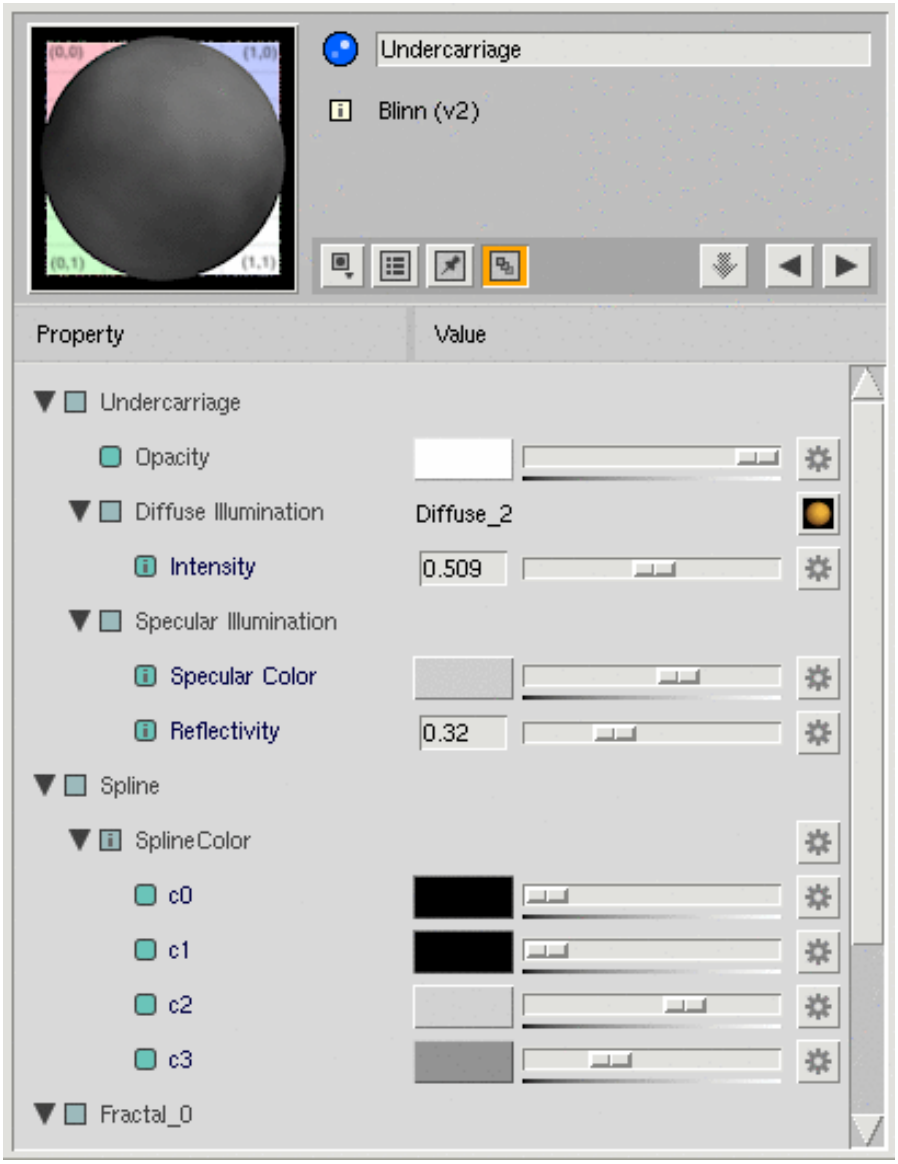


1. **Value Provider**
   Note that the Value Provider section indicates that the parameter is receiving its value from a connection to "LabelColor."
2. **Connection Commands**
   Because the Value Provider is currently a connection, the commands section changes to present two commands: Disconnect LabelColor and Delete LabelColor.
3. **Connections to New Functions**
   These cascade menus list templates that can be used to generate a new function. Selecting one of these creates a new function from the selected template and connects the parameter to the function.
4. **Connections to Existing Functions**
   This last list is of existing functions that can be connected to the parameter.

Parameter Overview

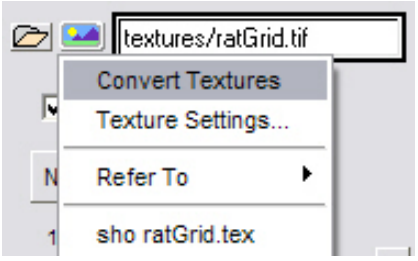
As your networks grow, you may find it difficult to keep track of all of the parameters in all of the appearances in a network.

With any attachable shader, you have the option of viewing a **Parameter Overview** (invoked by the associated toggle switch). The Parameter Overview displays all external parameters in all of the appearances of the network. This allows you to quickly adjust parameters across your entire network. It also provides a preview of the set of parameters for your compiled shader.



Working with Textures

Slim accepts both TIFF files and Pixar texture files (*.tex), a format optimized for high quality texture filtering and high performance. You can specify textures for use in your shaders through various "file" parameters associated with your shaders.



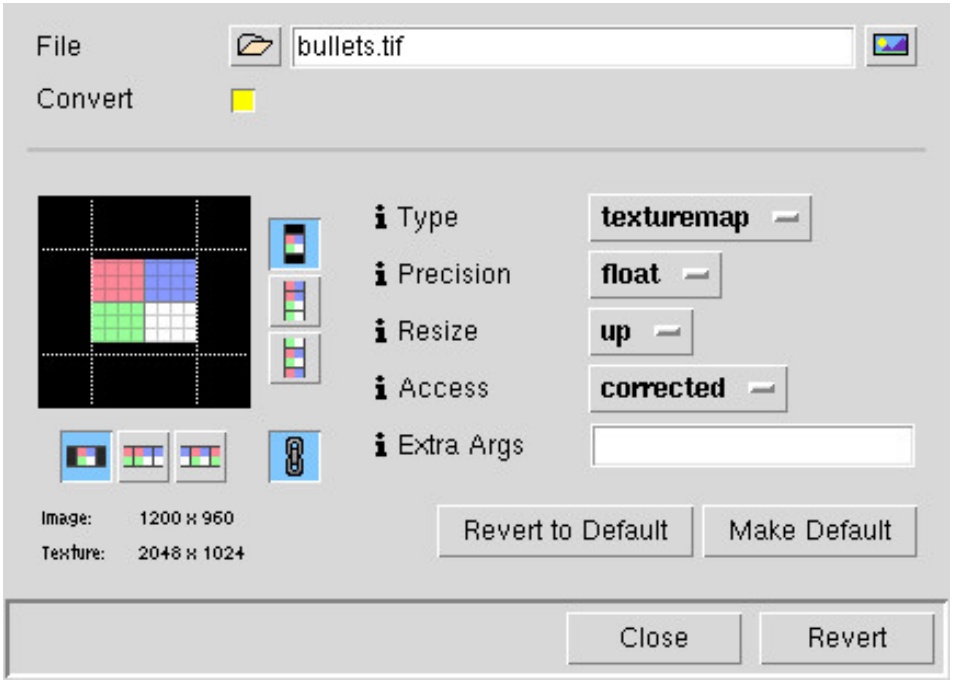
For texture parameters, Slim provides two controls adjacent to the text entry field. The left "folder" button simply invokes the File Picker, which is the most convenient means of loading in a texture file. The button on the right provides a menu of commands to perform on your image.

You can use this menu to open the Texture Settings Editor.

Texture Settings Editor

The Texture Settings Editor is a convenience provided to simplify the task of controlling the details of texture conversion. By selecting entries in this menu you are building up the argument list to the Slim provided TCL procedure: **txmake**. The TCL proc txmake is merely a cover function for RenderMan's standard texture conversion features and specifically Pixar's txmake utility program. Its purpose is to help track all texture conversion requests and to calculate the mapping between source texture name and destination.

Here are the primary controls:



File

The image file.

Convert

This switch controls whether the file should be converted to a texture. The texture controls appear when this switch is on.

Type

Type of texture to create. Type is one of:

- ***texturemap*** : standard color or grayscale image
- ***shadowmap*** : Pixar's shadowmap format, converted from a depth map.
- ***env latlong*** : environment created from a lat-long projection stored in a two-dimensional image.

Precision

Precision for each channel of each pixel. By default, each channel is represented by eight bits (a byte). Using -short will represent each channel with a 16 bit (signed) integer. Using -float will represent each channel with a 32-bit floating point number. Note that these options will only work if there is at least as much precision in the source file.

Resize

For reasons internal to the texture mapping software, texture files must be an even power of two in width and height (i.e. 256, 512, 1024, 2048, etc). Any input image which is not already a power of two in both dimensions will be resized, as described below:

- ***up*** : the image is resized up to the next higher power of two.
- ***down*** : the image is resized down to the next lower power of two.
- ***round*** : the image is resized to the nearest power of two.
- ***none*** : the image is not resized, the texture map size will be the next higher power of two. Texture file area not covered by the image will be set to black. Texture coordinates will be 0 to 1 across the resulting image.

Access

How textures will be accessed when the image is resized:

- ***corrected*** : the image will retain its aspect ratio when mapped onto a square patch.
- ***normalized*** : texture coordinates will be 0 to 1 across both dimensions.

Extra Args

Extra arguments for the txmake call.

Smode/Tmode Icons

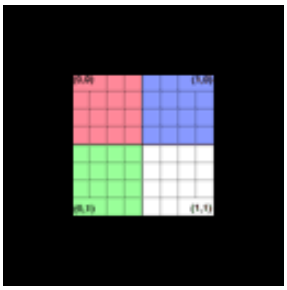
the icons next to the gridded texture determine the repeat mode for your texture in each direction. This applies when accessing the texture outside of the range 0-1. (See examples in diagram below.)

- ***black*** : fill the nether regions with black
- ***periodic*** : tile texture
- ***clamp*** : smear the pixels at the edges indefinitely

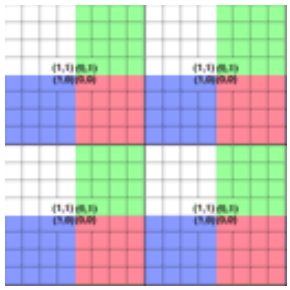
Remember that these controls are only meaningful for external, pre-existing textures. You should use the Reference Texture Menu for computed maps.

Conversion Mapping Modes

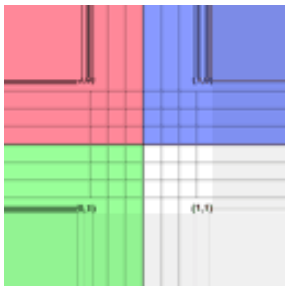
Original Texture



"Black" smode/tmode



"Periodic" smode/tmode



"Clamped" smode/tmode

Prev | Next

Interface Features

Introduction

Slim supports flexible slider capabilities and a powerful level of drag and drop. These features streamline the user interface, and give the user the luxury of developing a personalized workflow.

Sliders

Combination Sliders

This is the standard editor for a float parameter in an appearance. On the right is a graphical slider that can be moved back and forth to change the value. On the left is a numerical entry field that can be used to directly enter a value. This field can also be used as a virtual slider, or *VSlider*.

Using a VSlider

Select the numerical field, and drag the left mouse button back and forth while holding down **Ctrl**.

The value will increase or decrease as the mouse is dragged from left to right or right to left, respectively. You can accelerate the response by holding down the *Shift* key.

Adjusting Sliders

For parameters with unbound ranges, the graphical slider will always reset to the middle after a change. If it does not, the template has specified bounds for the parameter. This is a template-recommended range of values and should normally be respected. If you find you need to override this range, however, you can use the VSlider to enter a new value. When you exceed the template-recommended range, the Vslider will turn red:

You can also use the VSlider to adjust a parameter's precision. Simply enter a value carried out to three decimal places, e.g. "0.001", in an entry with a value carried out to two, e.g. "0.10", and it will change accordingly. Specifying more precision will not cause the VSlider to turn red.

Sliders for Colors

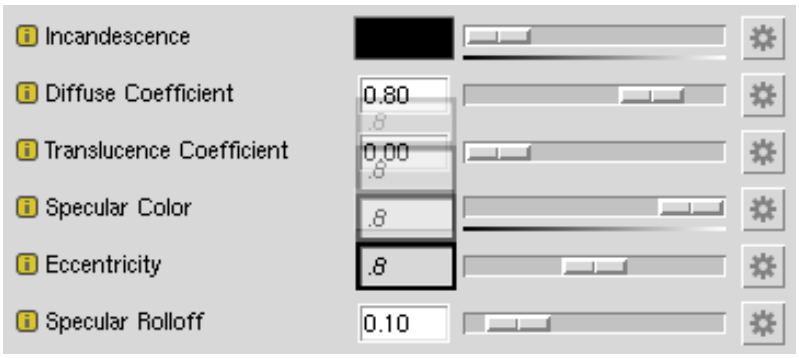
The value editor for colors also features a slider. This allows you to quickly adjust the intensity of a color without raising the full Color Editor.

Drag and Drop

The middle mouse button can be used to drag and drop values onto other values. Drag and drop is even supported between different applications; colors can be dragged from "it" to Slim, for example. For every dragging operation, the *middle mouse button* is used. Click, hold, and drag

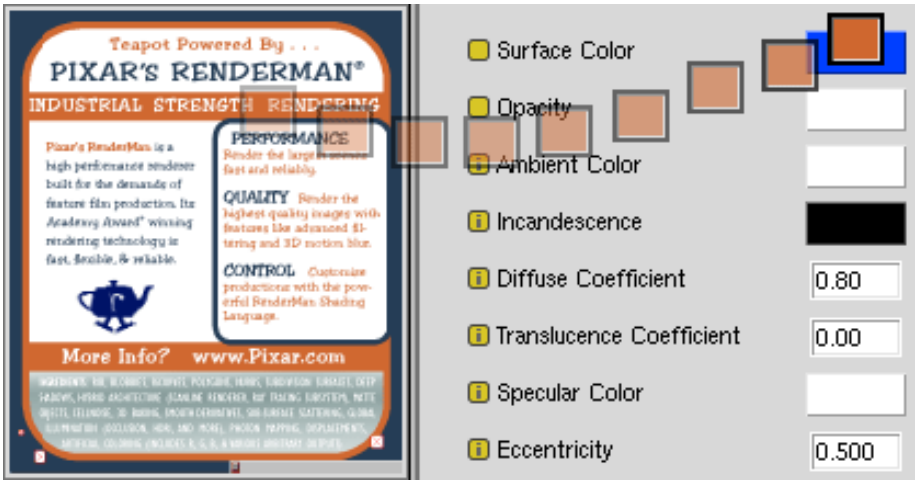
With Parameters

- **Colors** can be dragged to other colors, within the same or between different windows.
- **Text and numbers** can be dragged onto other alphanumeric fields. This works between both VSliders and ordinary sliders:



With Images

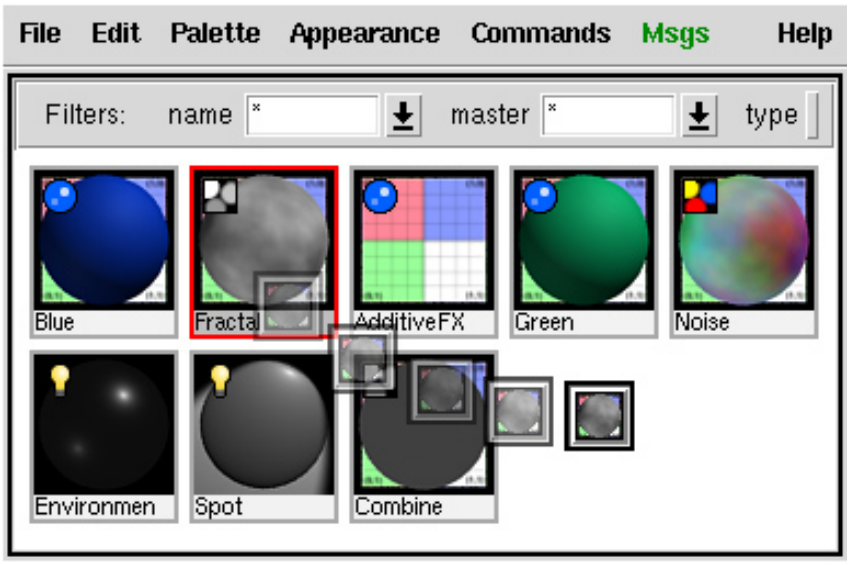
- **Colors** can be dragged from an image in an open "it" window to any color parameter or color picker:



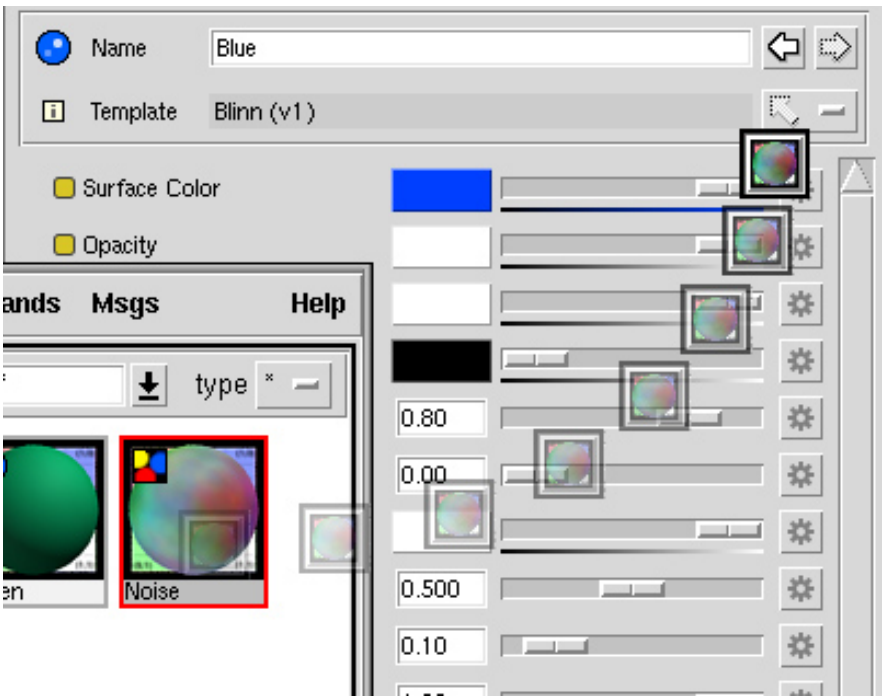
With Palettes

- Appearances in a palette can be dragged **inside** subpalettes. Drag the appearance on top of a subpalette.

- An appearance in a palette can be dragged around within a palette in order to **reorder** it. (To position a shader between two shaders in a palette, make sure you drop it *between* the other appearances in the palette.) This drag positions "Fractal" at the end of the palette:

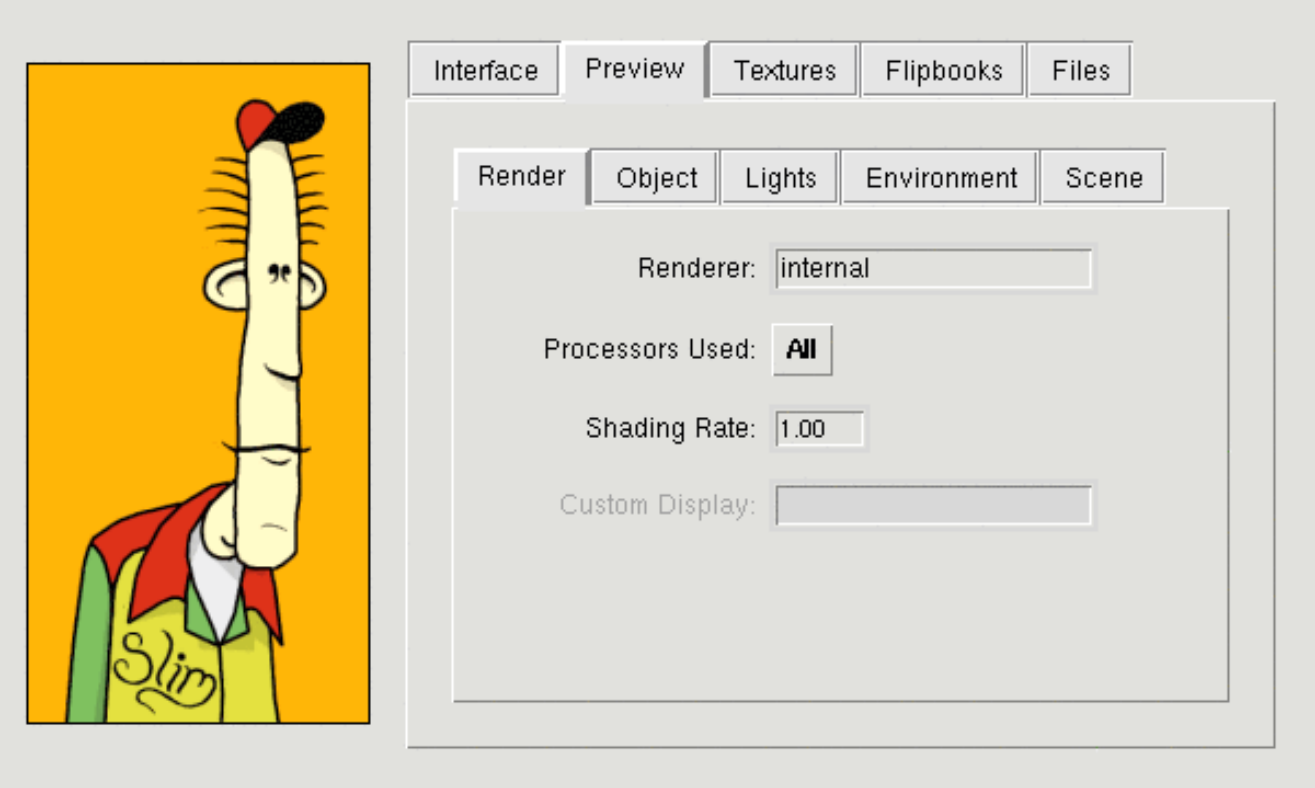


- Appearances can be dragged and **copied** into *another* palette. Open two palettes and drag an appearance from one palette to the other. Notice the entire shader network is copied and given a unique identity.
- An appearance can be dragged and **connected** to an available parameter in an open appearance editor. This powerful feature speeds up building shaders:



Preferences

The preferences window for Slim can be opened from the **Windows** menu. Preferences allow the user to control many different aspects of Slim, from tweaking render previewing to unlocking advanced features for expert users.



Interface

Raise Slim (RenderMan Artist Tools only)

When reading a scene file with Slim palettes, raise Slim.

Gain, Gamma

Gain and gamma settings for colors renders and textures.

Instance Structure

Controls the presentation of parameters in instances. By default, these parameters are arranged in a "Hierarchical" fashion that approximates the structure of the palette. "Grouped" presents a simpler arrangement, with just one level of nesting, and "Flat" lists all parameters with no nesting at all. The last option is recommended when using Slim with RenderMan for Maya.

Expert Menus

Controls whether to display extra menus tailored for extension developers and experts.

Preview

These settings allow you to configure the appearance of preview (or *swatch*) renders. These settings are spread across these sections:

Render

Renderer

To use Slim's internal prman renderer, set this to internal. To plug in your own renderer, place your rendering command here. You can use `%f` and `%i`, which will be replaced by the RIB filename and the Slim internal image name (respectively), e.g.:

```
prman %f
```

Processors Used

Number of processors to use for preview renders. *All* instructs Slim to use all of your computer's processors.

Shading Rate

Default Shading Rate for preview renders

Custom Display

Use this option if you'd like to send the render to a display other than Slim. This option is disabled when using Slim's internal renderer.

Object

Object Size

Default size of the object rendered in previews

Object Shape

Default shape of the preview object

Archive

Use this to select a custom RIB archive that will be used for the preview object. This option is disabled when using Slim's internal renderer.

Lights

Simple Lights

This toggles the use of the three simple lights defined below.

Key Light

The color and intensity of the key light for preview renderings. The key light is located behind, above, and to the right of the camera.

Fill Light

The color and intensity of the fill light for preview renderings. The fill light is located behind, below, and to the left of the camera.

Ambient Light

The color and intensity of the ambient light for preview renderings.

Archive

Use this to define a custom RIB archive that will be used for lights. This option is disabled when using Slim's internal renderer.

Environment

Environment Light

Toggles the use of an environment map-based light in preview renders. This environment performs an inexpensive approximation of diffuse and specular light for the purposes of previewing your shader.

Environment Map

Texture to use for the environment light in preview renders

Kd

Strength of the diffuse component of the environment map

Diffuse Blur

Amount to blur the environment map for diffuse approximation

Ks

Strength of the specular component of the environment map

Scene

Up Axis

Up Axis for preview renders

Background Texture

Texture to use for background plate in preview renders

Surrounding Sphere

Toggles the presence of a sphere that surrounds the scene. This may be useful when ray-tracing.

Sphere Texture

Texture map used on the surrounding sphere

Show Frame Control

Show the Frame setting when displaying render settings. This can be useful when testing animated shaders.

Textures

These settings apply to the rendering of textures only.

Resolution

The resolution, in pixels, of the rendered image. Textures are always rendered square.

Renderer

The renderer to employ to render the textures. The line above causes Alfred to distribute the rendering to up to six processors. Slim substitutes `%f` with the name of a RIB file in the expression. Another common choice would be `render %f`.

Flipbooks

These settings apply to the rendering of flipbooks only.

Resolution

The resolution of each frame in the flipbook, in pixels. Flipbook frames are always square.

Length

The number of frames in a flipbook.

Sequence Range

The first and last frames of your sequence. These are used by the scripting subsystem to calculate the `$pct` variable. Specifically:

```
$pct = ($f - $firstframe) / ($lastframe - $firstframe).
```

Renderer

The renderer to use for flipbook rendering. Choices are: internal, render and netrender.

Files

Save Icons

Which appearances should retain their icons when saved. Icons are useful for identifying shaders, but will make palette files larger.

Saved Resolution

Resolution for saved icons. Saving at the lower resolution will keep palette files smaller.

Full Paths

Controls whether to automatically truncate files selected via the file picker to relative pathnames. The current workspace's searchpaths are used to peform the truncation.

Master Directory

Controls whether the directory for an appearance's master is dereferenced upon creation. This can add stability at the cost of flexibility.

Checkpoint Queue Size

Sets a number of checkpoints for versioning any given palette. With the default setting of **0**, no checkpoints will be kept. Higher settings will save a checkpointed versions of the palette to disk when a palette is saved, to the maximum setting provided by this preference. The palette checkpoints will conform to a numerated ***.bk** naming convention. Palettes may be reverted via Slim's **Revert** command.

Shader Compiler

The command to execute compile shaders. Note that `%f` will be replaced with the shader source filename and that `%I` will be replaced with the standard Slim include directory. The default compiler command, then, is:

```
shader -I%I %f
```

Secondary Compiler Command to execute as a secondary shader compiler. secondary compiler may be useful when developing shaders for use in hybrid rendering schemes. As with the primary compiler, `%f` will be replaced with the shader source filename and `%I` will be replaced with the standard Slim include directory.

Organization and Reuse

Sessions and Palettes

Introduction

Managing your work in Slim is designed to be simple and virtually effortless, straight out of the box. Appearances are stored in Palettes, and Palettes are stored in Sessions. Sessions have been designed to optimize the inter-connectivity of Slim and RenderMan for Maya and their integration with your Maya workflow.

When you start Slim, you are in a default Session, working on a default Palette. These are keyed to your Maya scene and project automatically, and are saved with the scene when the scene is saved. Sessions and Palettes can also be manually saved and opened (or re-opened). If you are working with a default session and/or palette, you do not need to explicitly perform any save operation.

Important

Any explicit save of your palette will externalize the palette, substantially loosening the connection to your scene/project; you can, however, internalize any saved palette from the **File** menu.

Managing Sessions and Palettes

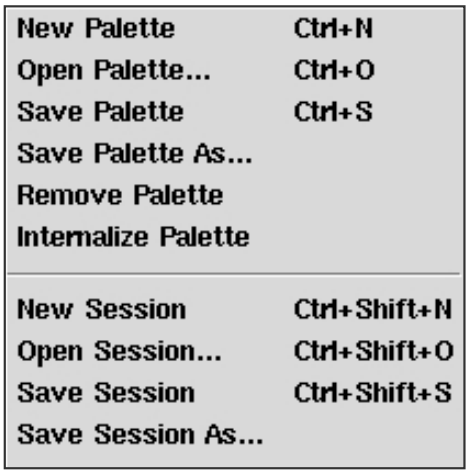
Sessions

As mentioned above, when you open Slim via Maya you create a default Session. When you save your scene the Session is saved with it, as well as a *Session Key* that tells Slim which Session is associated with the scene.

If, however, the default session displeases you, you can create a new Session. To create a new Session:

File → New Session

When you create a new Session in an existing Maya scene, the first thing you must do is give the Session a name. This will also update your Session Key to keep your current session in sync with your scene. The same logic applies if you open an existing Session (*File → Open Session*) and if you Save Session As.



*Palettes and Sessions via the **File** menu.*

> **Note**
>
> In practice, you never need to save a Session. Whenever you exit Slim or save your Maya scene the current state of the Session is automatically saved.

Sessions can also be created, reopened, and saved when you are using Slim in standalone mode. Existing sessions will retain their connection to the Maya scene they were tied to as long as the session key is not changed (e.g. by doing a Save Session As).

Palettes

Palettes are a slightly different beast. Once again, they are stored in Sessions, via references. You can have any number of Palettes in a given Session. Palettes can be named (or renamed) when in *Browse* mode by double-clicking on the label, and, as mentioned above, explicitly saving a palette will externalize it. You can open existing Palettes and they will automatically be added to your current Session, and your Session can contain both internal and external Palettes.

Internal palettes are good for ensuring that all of your Slim appearances are associated with one particular Session. External palettes may be preferable if you want to manage palettes independent of a given session; they can be shared between projects, i.e. any external palette can be referenced in multiple Slim sessions, whereas an internal palette can only be associated with one.

Sessions Behavior Modification

Here's a rundown of what's going on under the hood with Slim Sessions:

Initialization

Upon startup, Slim and RenderMan for Maya source the `RMS.ini` file (by default, it is in the `/etc` directory of your RenderMan Studio installation, but they will also source ini files in your `MAYA_SCRIPT_PATH` as well). This file contains settings for two preferences. The first (and primary) preference is `AssetStageStrategy`. In the default configuration, the relevant line of the ini file appears like so:

```
SetPref AssetStageStrategy file; # file, fileStripVersion, proj
```

This pref governs how the value of the STAGE variable is constructed. It can take one of three values: `file`, `proj`, or `fileStripVersion`.

- When the value is `file`, the STAGE variable will be set to the name of the current Maya file (no path, no extension).
- When the value is `proj`, the STAGE variable will be set to the name of the current Maya project.
- When the value is `fileStripVersion`, we refer to `AssetStripVersionExpr` (the second preference contained in the RMS.ini file) to represent the transformation that strips file version information. The default preference

  ```
  SetPref AssetStripVersionExpr {[-_]v\d+$}
  ```

  supports files named like *asset1-v1.ma*, *asset1-v2.ma*, and so on. The aforementioned default expression will strip any filename ending that is made of a dash (-) or an underscore (_), followed by a v character, followed by one or more numbers. For example, *asset1_v1.ma*, *asset1-v1.mb*,

asset1_v001.mb, and *asset1-v001.ma* would all return asset1 .

The ini file also loads the `RMSExpression.tcl` file, located in the `/etc` directory of your RenderMan Studio installation. This file outlines specific routines governing the interaction of Slim and RenderMan for Maya. As with the preferences in an ini file, you can override any of the procs in a separate `RMSExpression.tcl` file in your `MAYA _SCRIPT_PATH`.

Configuration

The Workspace

In the default configuration, the Workspace file includes:

```
SetPref WSSubdir.slimPalettes    {slim/palettes}
SetPref WSSubdir.slimSessions    {slim/sessions}
SetPref WSSubdir.slimSessionKeys {slim/sessions}
SetPref WSSubdir.slimShaders     {renderman/$STAGE/slimshaders}
SetPref WSSubdir.slimTextures    {renderman/textures}
SetPref WSSubdir.slimTmps        {slim/tmp/$STAGE}
```

Prev | Next

Smart Palettes

With Palettes and Subpalettes, you keep your appearances explicitly organized. **Smart Palettes** provide a way to organize your appearances dynamically.

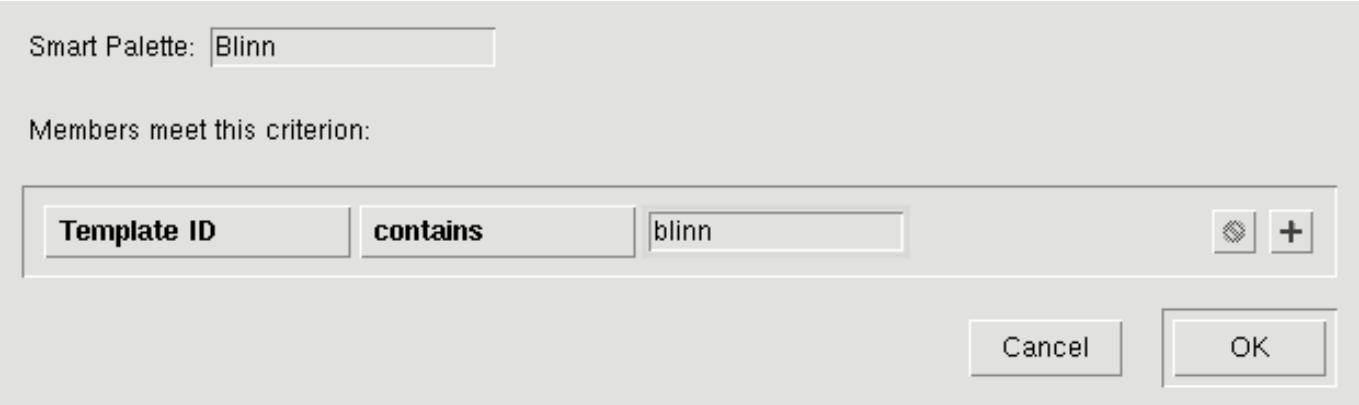
Membership in a Smart Palette is defined by one or more *criteria*. An example Smart Palette might be defined as the set of appearances that use the Blinn template. The contents of this Smart Palette will update dynamically as new appearances are added to its parent palette.

Creating and Editing a Smart Palette

To create a new Smart Palette:

> *Palette → Create Smart Palette*

This will create a new Smart Palette rooted under the selected palette and raise the **Smart Palette Editor**.



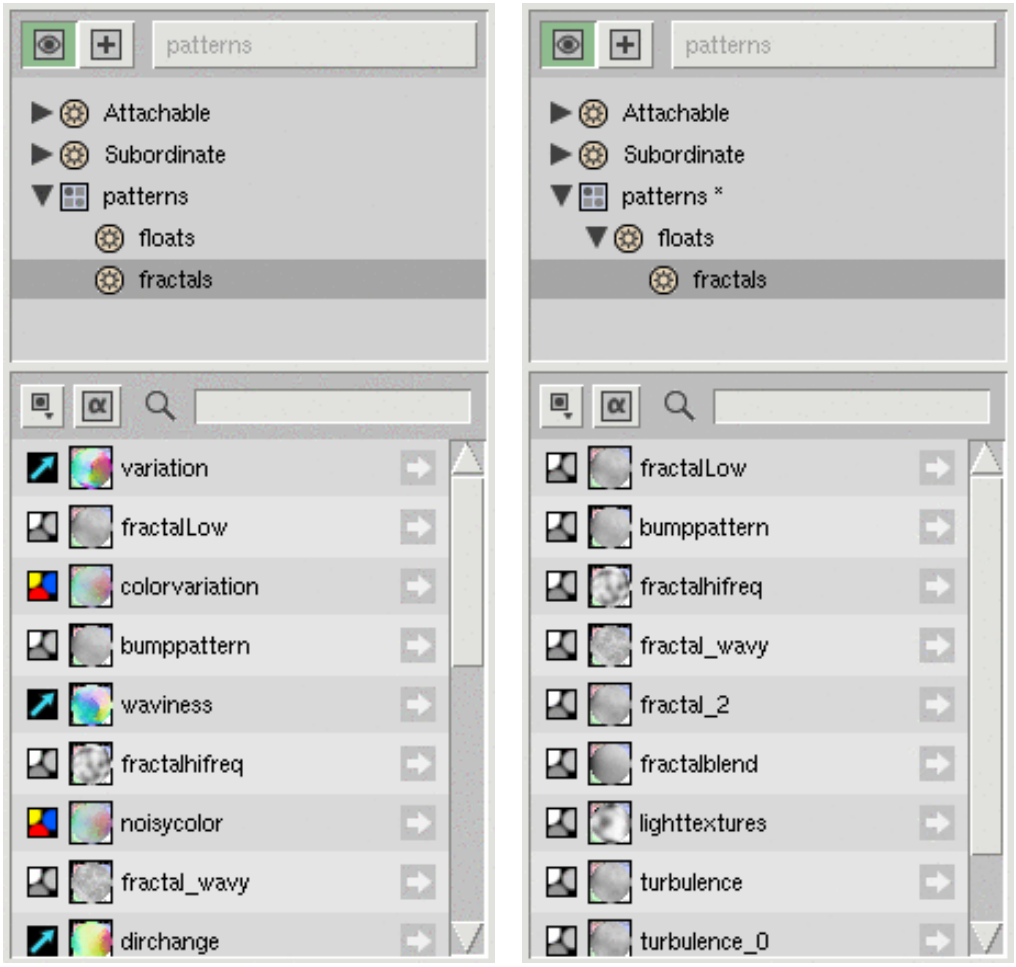
Use this editor to name your Smart Palette and define its criteria. In this example, we've defined the "Blinn" Smart Palette using a criterion that checks if the template ID for an appearance contains the word "blinn."

By default, a Smart Palette consists of one criterion. You can add more using the **+** button. With multiple criteria, you will have the option of selecting whether membership is defined by meeting **all** of the specified criteria or **any** of them.

You can edit the criteria of an existing Smart Palette by double-clicking its icon.

Nesting and Refinement

The candidates for membership in a Smart Palette are the members of its parent palette. Typically this is a palette file, but it can also be another Smart Palette. Smart Palettes nested in this fashion serve as increasing levels of refinement.



In both of the images above, the "fractals" Smart Palette is defined to contain appearances that use templates containing the word "fractal." When nested under the palette file, this displays appearances of all types that use a fractal template. When nested under the "floats" smart palette, however, the "fractals" Smart Palette only displays appearances of type float.

Global Smart Palettes

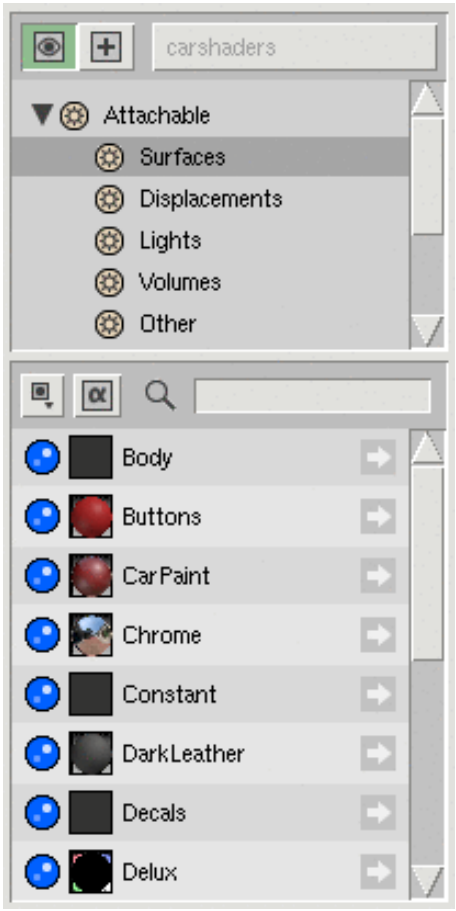
Global Smart Palettes operate on all of the appearances currently loaded. These Smart Palettes are not saved with any particular palette file; they are in fact defined and saved using preferences.

Global Smart Palettes allow you to operate on all of your current appearances, regardless of their source palette. Slim ships with a number of predefined Global Smart Palettes that allow you to quickly see just attachable appearances, just surface shaders, and so forth. You are free to redefine, reorder, and further customize these.

A second set of Global Smart Palettes is used when the Palette View is in Create Mode. These Smart Palettes provide the same refinement mechanism, but rather than showing current appearances, operate on the available templates you can use to create new appearances.

You will find that these Smart Palettes are referred to elsewhere in the system. Whether creating an appearance directly in the Network View or when creating a connection in the Appearance View, you will find that these Smart Palettes are used to organize available appearances and templates. Here, for example, is the context menu in the Network View. Note the

groups shown: Surfaces, Displacements, Lights, Volumes, Other. These are the same as those in the Palette View shown.



Smart Palette Criteria

These are the possible critera you can use to define a Smart Palette:

Attachment Type

The attachment type for the appearance. Example attachment types include `surface`, `displacement`, `light`. The "Attachable" Global Smart Palette is defined using a criterion where "Attachment Type" is not (blank) (i.e. it is defined).

Class

The class of appearance (e.g. `::Slim::Function`). See the Scripting Glossary for an index of Slim classes.

Compilation Necessary

Whether the shader for the appearance needs generation and compilation.

Light Type

Type of light shader, e.g. `point`, `distant`, `spot`, `environment`.

Modification Time

Time when the shader is last modified. Predicates for this criteria are specified relative to the current time (e.g. "less than 2 hours ago").

Name

Name of the appearance. This is most often useful using a test such as "begins with" or "contains."

Output Types

The types output by the appearance.

Selected in Client *(only when used with MTOR)*

Whether the appearance is attached to the current client (e.g. Maya) selection.

Tags

Possible appearances tags.

Template ID

The template ID for the appearance (e.g. `pixar,Blinn#2`). This is most often useful using a test such as "contains," e.g. "contains" "blinn." (Tests are case-insensitive.)

Template Obsolete

The template used by this appearance is obsolete, and can potentially be upgraded.

Prev | Next

Tags

Palettes and subpalettes allow you to organize your appearances into a hierarchy. You might also, though, like to organize your appearances by category.

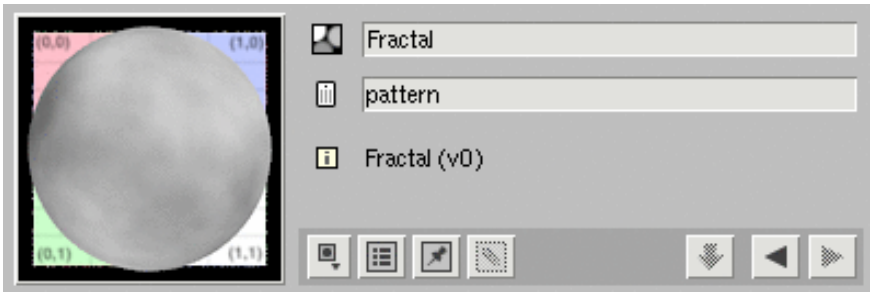
Tags allow you to do this. In Slim, every appearance contains a set of tags that you can use to categorize the appearance. These tags are first defined in templates, but can be overridden within the Appearance View. With tags set for your appearances, you can then use a Smart Palette to find appearances with specific tags.

Viewing and Editing Tags

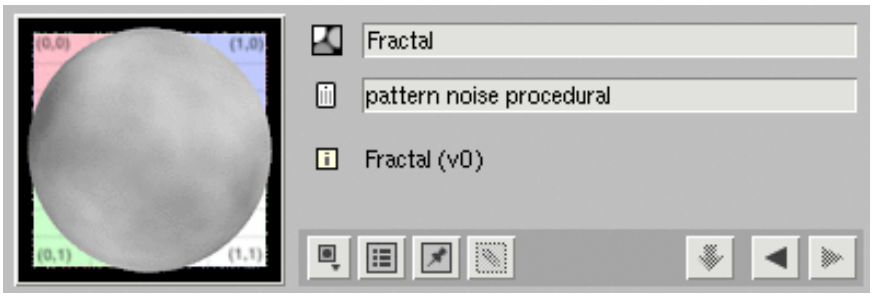
Tags can be seen within the Preview section of the Appearance View. By default, they are not visible. You can change this by selecting:

> *View → Show Appearance Tags*

Here we see the default tags for a Fractal appearance:

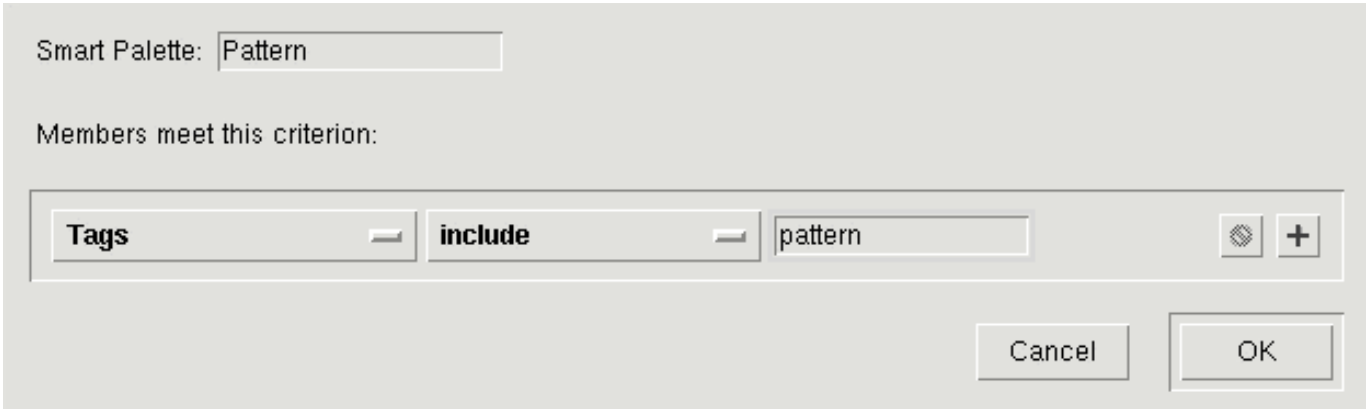


The **pattern** tag comes from the registration of the template. You are free to append or replace the default tags. Simply enter your list of tags separated by spaces. Below, we've added the tags **noise** and **procedural**:



Using Tags in a Smart Palette

With appearances tagged, you can now browse them using a Smart Palette configured to query these tags. Here is a Smart Palette that looks for the **pattern** tag:



This Smart Palette may look familiar. Slim ships with similar Global Smart Palettes, — grouped under and **Colors** — that show you the appearances and templates that have been tagged with **pattern**.

Prev | Next

Packages

You may find in your networks groups of appearances that combine to perform a single function. You can take one of these groups and present it as a single appearance by creating a *Package*.

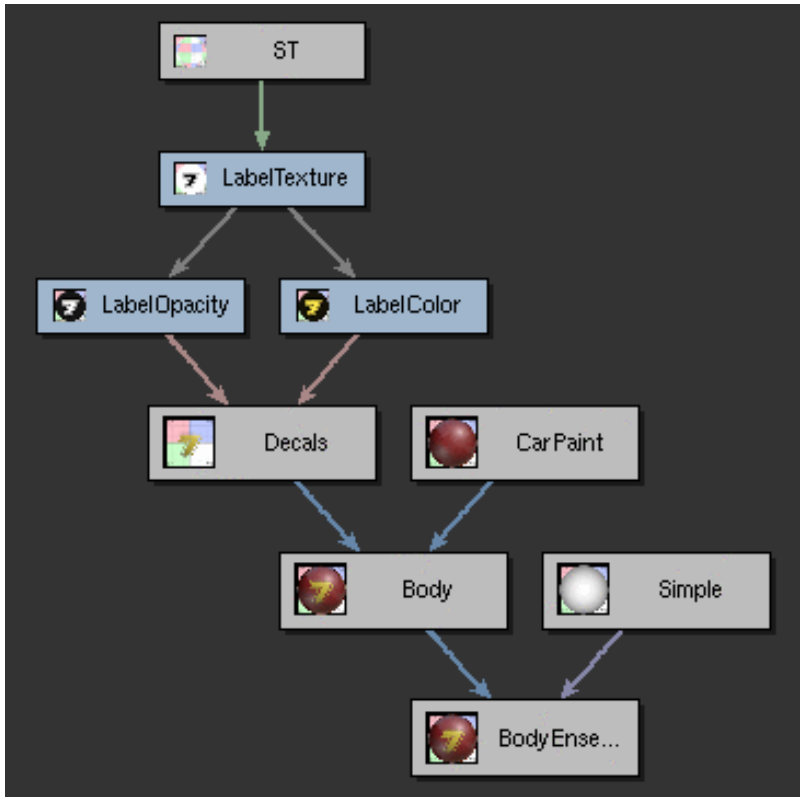
Packages serve many purposes. They make your network simpler, which may make its graph easier to navigate. They serve to compartmentalize your network into larger chunks of functionality. They also provide a way to share pieces of your network with others without these users having to understand the complexities inside.

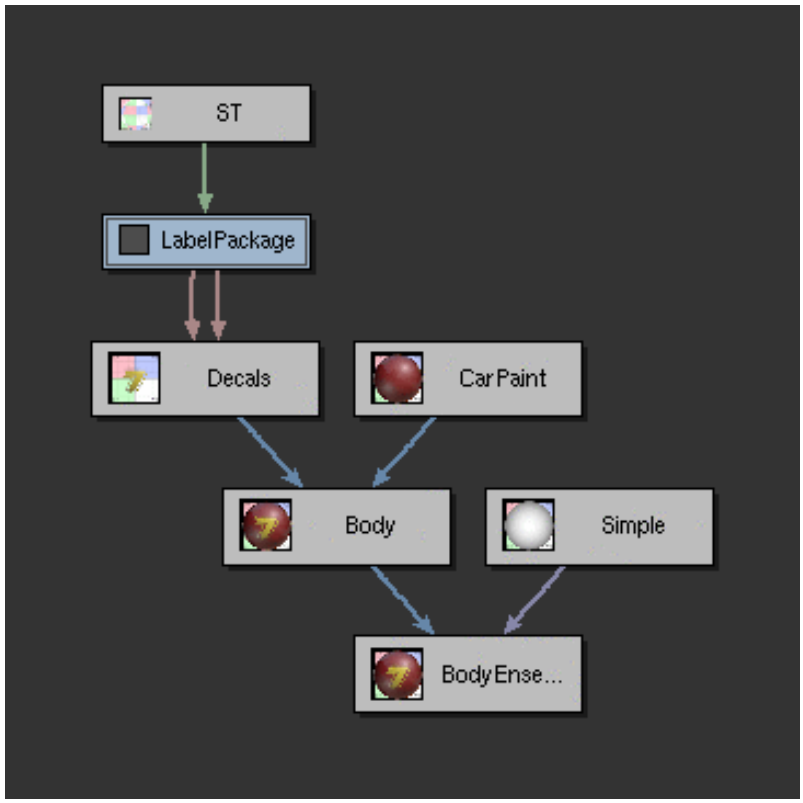
Creating a Package

To create a new Package, simply select the appearances that should be in the package, and:

Appearance → Package

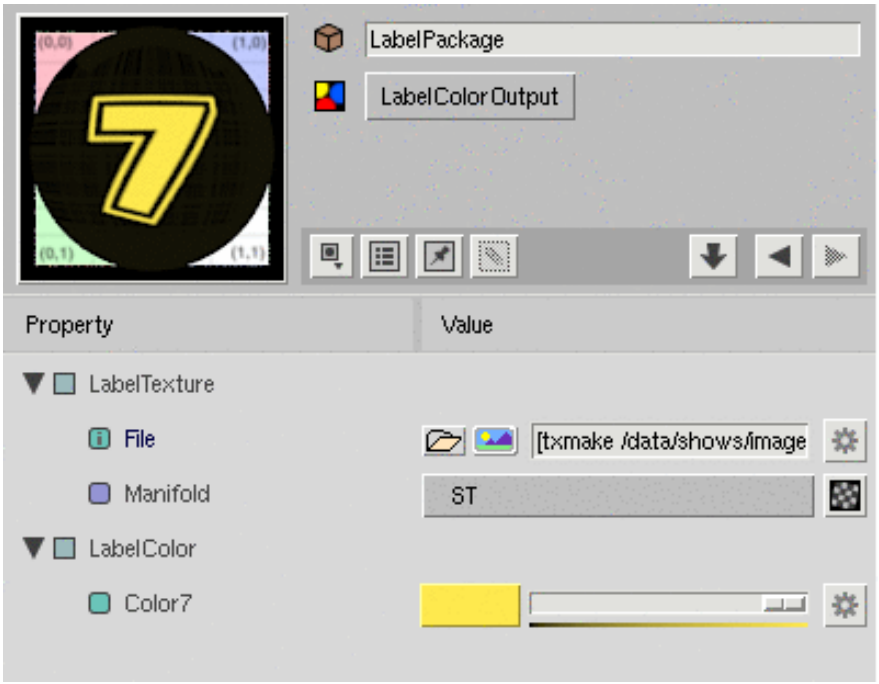
This will create a new Package node that includes the appearances you've selected. Below is a network before and after the creation of a package:





Package Parameters

Here is how the package created above appears in the Appearance View:



Package parameters are created for one of two reasons:

- Any connection to an appearance outside of the package (e.g. the connection to the "ST" appearance, not included in the package) will cause a package parameter to be created.
- Any external parameters of packaged appearances will become external parametes in the package.

You can edit the presentation of these parameters by editing its interface:

Appearance → Edit Package UI

This will place the Appearance Editor in a mode where parameters can be rearranged using drag and drop. When you're done rearranging,

Appearance → Done Editing Package UI

In addition, any unconnected parameter can be "internalized" back into the package via the Property Menu.

Packages and Multiple Outputs

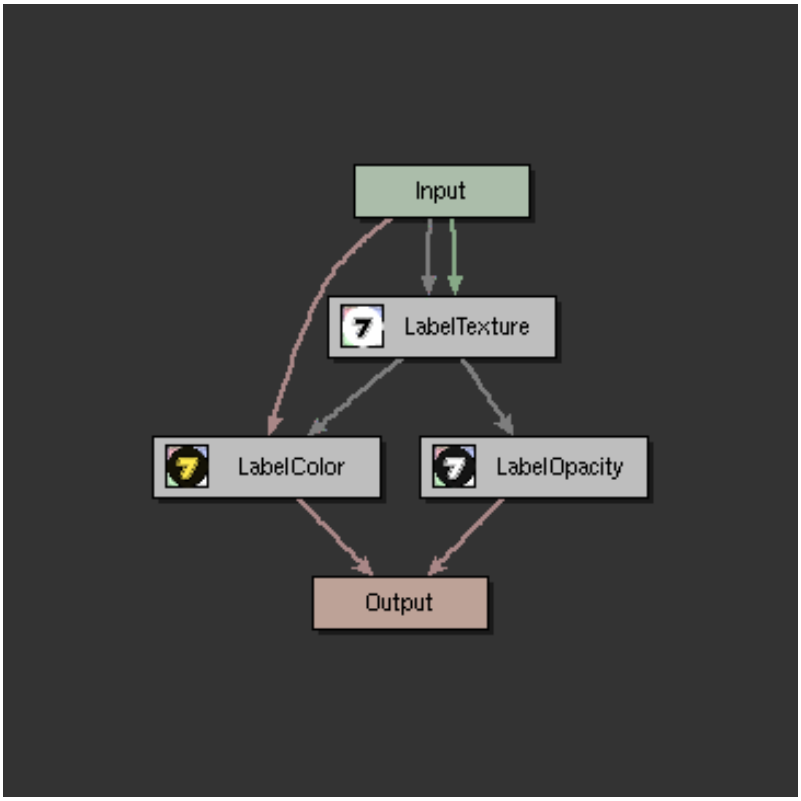
You may have noted in the examples above that the selection of appearances contained in the package actually had two different outputs: one for the LabelOpacity and one for the LabelColor. The resulting package also has two outputs. When working with packages like these, you will have to be explicit about which output you wish to connect, preview, and so forth.

Package Contents

A Package will usually be displayed as a single appearance. You can, however, see its inner-workings by revealing its contents:

Appearance → Show Package Contents

This will display the network of appearances that combine to form the Package.



In the above package, you should recognize the original appearances. In addition to these are two special nodes called *Package Terminals*. These nodes indicate how values flow into and out of your Package. The Input terminal has outputs for each of the Package's parameters shown above. The Output terminal has inputs for each of the Package's outputs.

These terminals allow you to change the "wiring" of the Package without affecting any appearances that connect to it.

To close a package back up:

Palette → Close Package

Unpackaging

Should you change your mind about the creation of a package, it can always be undone:

Appearance → Unpackage

This command will connect out-of-package appearances back to the packaged appearances and delete the Package and its terminals.

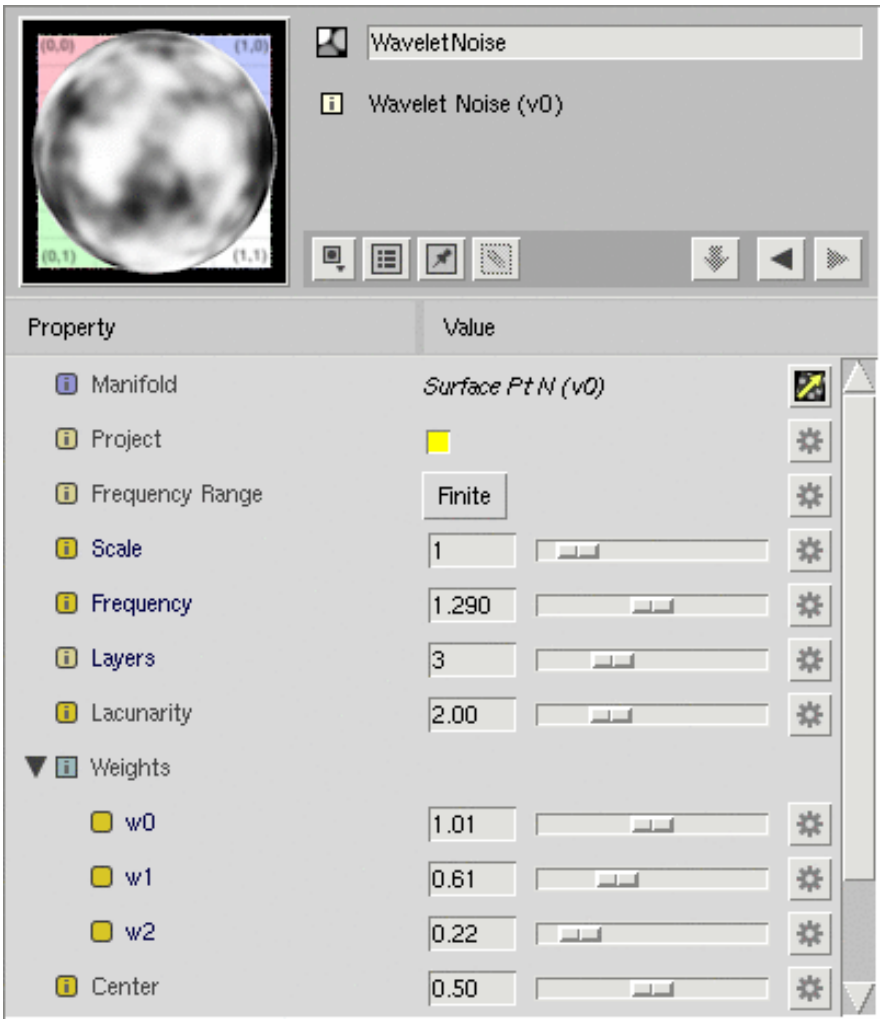
Note: Other than recreating the package, this command can not be undone. Please use it with caution!

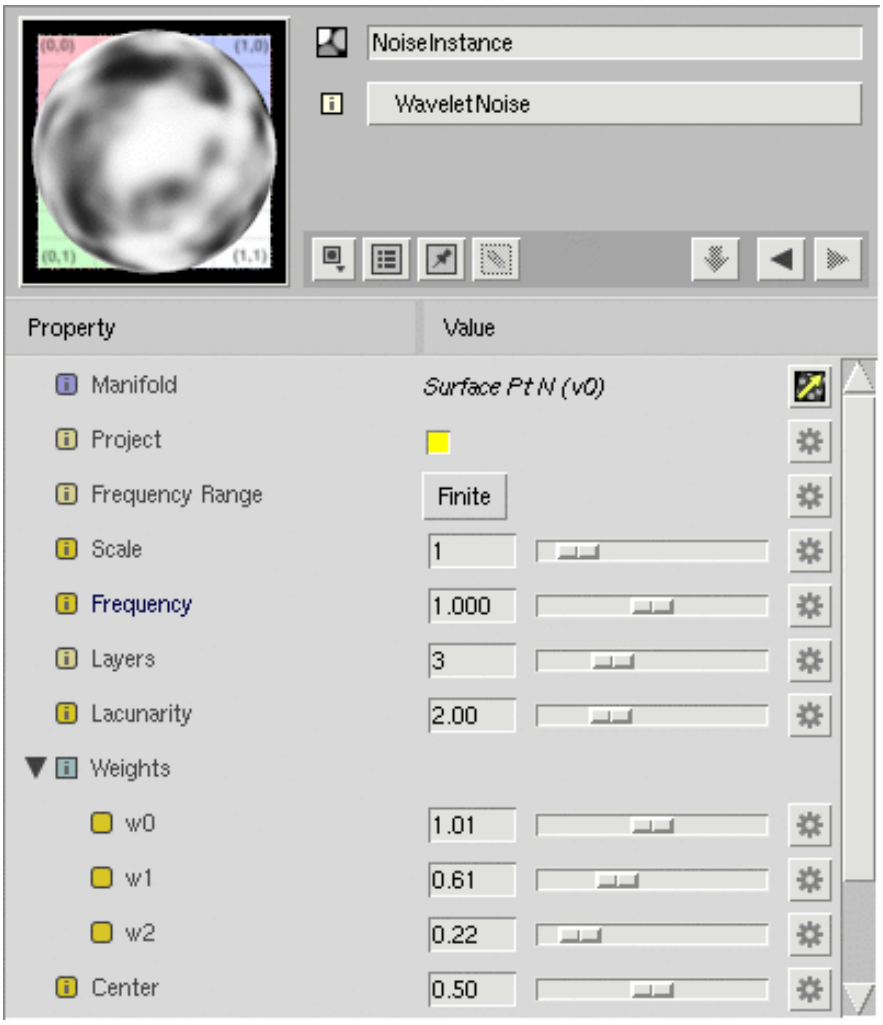
Prev | Next

Instances

When developing shaders, you will often find yourself managing appearances very similar to one another. You might, for example, have two shaders that are identical but for a few differences such as their color, or perhaps the texture map used.

Instances allow you to create these variations, explicitly setting the differences while keeping the rest in sync. Below is an appearance created using the Wavelet Noise template and an instance of that appearance:

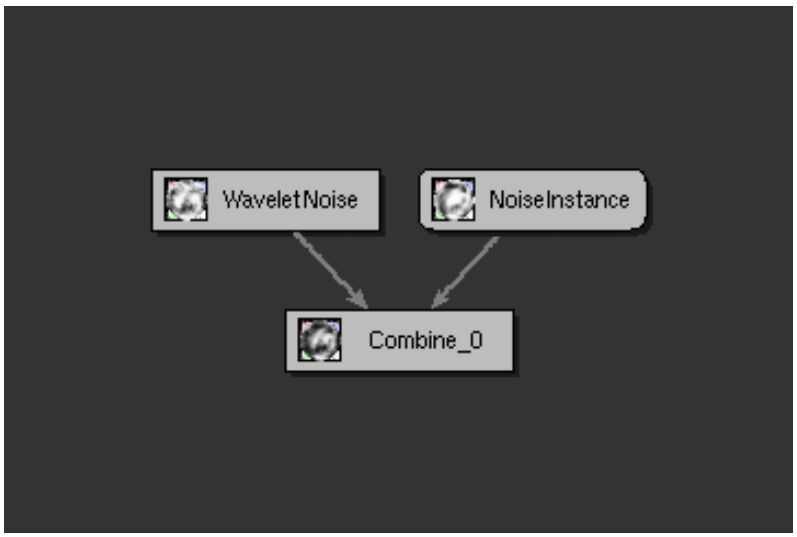




Some things to notice about the instance:

- Unless overridden, all parameter values, even when provided by a connection, will match those of the original. The original's parameters are overriden any time you initiate a change in a parameter's value undo -ing the *Set Value* will undo the override as well. In the screenshot above, only the Frequency parameter has an explicit value. The other values are inherited from the original.
- The Preview Swatch lists the original appearance. Clicking this button will take you to the original.

To differentiate themselves in the Network View, instances display with rounded corners. Below is a small network incorporating both the instance (NoiseInstance) and the original node (WaveletNoise).



Creating an Instance

To create an instance of an appearance, select the appearance and:

Appearance → Create Instance

You can create an instance of any appearance in Slim, *even another instance*. Such an instance will inherit its values from the original instance, which will inherit its values from the original appearance, which will inherit its values from a template. Instances establish a chain of inheritance, and you can make changes anywhere in that chain.

Instances and Multiple Palettes

It is possible to take an instance and drag it into a new palette. Be aware, however, that the instance relies on the original for its values. If at any time the instance is loaded and the original is not. You will receive a warning in the Message Log, and the instance will be non-functional.

When used carefully, however, this can be a powerful tool for reusing your work.

Shader Instances

The instances discussed above are of a single appearance. *Shader Instances* are instances of the entire network that forms a shader. They are created by importing the shader created by that network.

A shader instance displays all parameters in the original network that have been marked **External**. Like a regular instance, the values of these parameters will be updated as they are updated in the original (though unlike a regular instance, these updates will only occur when the shader is generated).

You can create a Shader Instance by selecting a shader-creating appearance and:

Appearance → Create Shader Instance

This will generate the shader for the selected appearance and import it into your palette. Like a regular instance, a shader instance will show a button with the original appearance in the Appearance View. Use this to navigate to and make changes to the original network.

Prev | Next

Library Palettes

Templates are the basic building blocks of shaders created in Slim, and are designed to be used and reused. As you use templates, you will find yourself creating shaders and pieces of shaders that you want to reuse. With Library Palettes, you can access and reuse these just as easily as you can templates. And unlike Slim templates, Library Palettes and their contents can be edited right within the Slim interface.

Using Library Palettes

Members of Library Palettes are shown when the Palette View is in Create Mode and (by default) are displayed along with templates in Global Smart Palettes. You can also view just the contents of Library Palette by selecting it in the Palette View.

Like a template, you can add a member of a Library Palette to your current palette by clicking on it. Depending on how that Library Palette is configured, adding will take the form of copying or instancing.

Editing Library Palettes

Fundamentally, Library Palettes are no different than ordinary palette files. In normal use, they are effectively read-only. To make changes to the contents of a Library Palette, simply tell Slim you wish to edit it. In Create Mode, select the Library Palette you wish to edit and then:

> *Palette → Edit Library Palette*

This will remove the Library Palette from Create Mode and place it in Browse Mode. You may then make changes to that palette as if it were any other palette. Inform Slim when you are done:

> *Palette → Done Editing Library Palette*

This will return the Library Palette to Create Mode.

Note: Like any other palette file, you may have to worry about write-permissions on the file that is storing your Library Palette. If after placing your Library Palette in edit mode the palette is still read-only, check that you have appropriate permissions to edit the file.

Specifying Library Palettes

Library Palettes are loaded with every Slim session, so they are specified within a Slim initialization file. By default, Slim ships with this specification for library palettes:

```
SetPref LibraryPaletteSets { personal }

SetPref LibraryPaletteDir(personal) $personalLibDir
SetPref LibraryPalettes(personal) {
    MyLibrary.splt {
        mode copy
        create 1
    }
}
```

The `LibraryPaletteSets` preference specifies the list of Library Palette *sets*. A set is a group of palettes that live in the same directory and are loaded at the same time.

The second and third preferences specify the location and contents of the `personal` set of Library Palettes. `LibraryPaletteDir` specifies the directory containing Library Palettes for the specified set. `LibraryPalettes` specifies the list of Library Palette files to find in that directory. These will be loaded in the order specified.

Each palette file listed in the `LibraryPalettes` preference is followed by a list of configuration options for the library palette:

mode (**copy** or instance)
: Action to perform when a user selects a member of the palette. By default, the appearance will be copied.

create (**0** or 1)
: Whether the palette should be created if it doesn't exist already.

For default configuration, follow the palette file with an empty list "{}".

To specify additional sets of Library Palettes, you'll need to first override the definition of LibraryPaletteSets, e.g.:

```
SetPref LibraryPaletteSets { shared personal }
```

Sets will be loaded in the order specified. An example definition for a shared set is shown below. Note there is nothing inherently special about the palette file specified. Any palette file can be a Library Palette, so long as it is specified here.

The default listing above specifies that a personal library palette should exist in the user's preference directory, and that it should be created if it doesn't exist already. The copy specification means that when the user selects an entry from the Library Palette, it will be copied into her current palette. This is the recommended action for Library Palettes such as this one which are not guaranteed to be loaded by other users. This Library Palette, then, is best used for a user's personal favorite appearances and networks.

To share a Library Palette among many users, it should exist in a shared location. If the palette is guaranteed to be loaded for all users, you can specify the action as `instance`. When a user selects an entry in such a palette, an instance of the selected node will be added to the user's current palette.

A palette specification for this sort of behavior might look something like the following:

```
SetPref LibraryPaletteDir(shared) /path/to/shared/directory
SetPref LibraryPalettes(shared) {
    CommonMaterials.splt {
        mode instance
    }
    CharacterShaders.splt {
        mode instance
    }
}
```

Using instances allows for more dynamic behavior. You can make changes to the global Library Palette and they will be automatically integrated into the palettes containing instances. This flexibility does require some care, however. For palettes with instances to function properly, the Library Palette must be loaded at all times.

Be cautious about actions such as Uniqueify, which will reassign unique identifiers and may cause instances to lose their connection to their original appearances.

As mentioned above, contents of Library Palettes are by default displayed along with templates in Global Smart Palettes. To disable this behavior, set the following preference:

```
SetPref IncludeLibraryInSmartPalettes 0
```

Prev | Next

Slim Scripting

Introduction to Slim Scripting

Background

You can write scripts to perform all the actions in Slim that you might perform interactively. You can write mel scripts to control Slim by wrapping the Slim commands in MTOR's slimcmd/slimmsg mel commands. In this way you can treat Slim as an extension of Maya and develop mel scripts that couple manipulations of geometry and Slim objects. This document serves as an introduction to scripting with Slim.

Using the Slim console you can interactively develop, debug and invoke the scripts you write. You can install developed Tcl scripts (see http://tcl.sourceforge.net/) by adding a LoadExtension directive to one of several initialization files.

Slim supports an object oriented programming paradigm. Slim commands are generally targeted to a unique object. Objects are of a particular class and all objects of the same class accept the same commands, or methods. Object handles are the means by which we identify a particular object. These handles are actually transitory Slim commands. Because they are transitory, they should not be hard coded into your scripts. Each time a script is run, it must locate object handles via appropriate commands.

The Slim Object

There is one object that you can hard code into your scripts. This object handle is **slim** and it represents Slim's application context. The slim object is the starting point for many of the scripts you will write. You can query the slim object for the active palettes or appearances. These queries will return object handles which you can use to perform further queries.

The following example code illustrates the typical workflow in a slim script. This snippet will find all parameters named "Ks" and set them to 0. It executes by querying slim for all active appearances, querying those appearances for parameters which match our criteria, and executing the `SetValue` method on the parameters that are found:

```
foreach app [slim GetAppearances] {
    foreach p [$app GetProperties -name "Ks" -access input] {
        $p SetValue 0
    }
}
```

The GetAppearances method is probably the most useful method of the slim object. Here are some of the other most useful methods of the slim object:

slim BuildShaders ?arg arg ...?

Generate and build all shaders associated with attachable functions.

slim CreatePalette ?arg arg ...?

Create or open a Slim palette.

slim GetPalettes ?arg arg ...?

Get object handles for all top-level palettes.

slim GetTemplates ?arg arg ...?

Get a list of all active templates.

The complete list of methods available for the slim object, along with their usage and arguments can be found in the Slim Scripting Glossary.

The TreeNode Class

In the above example, we are operating on three classes of objects: the slim object, appearance objects, and property objects. The slim object is unique. It is the only object of its class and no other classes inherit from it. The same is true of the workspace class (discussed below).

The rest of the classes you will work with, however, will have some things in common. All of the objects in these classes can be arranged in a tree-like structure. Palettes can be grouped within palettes (these are referred to as subpalettes). Appearances can be collected within other appearances. Properties can be grouped into a collection. Because these classes all have some common functionality, they all inherit from a common base class called **TreeNode**.

The TreeNode base class defines basic methods for organizing nodes. The names of these methods follow the metaphor of a family tree. Nodes are grouped together under a *parent*. Those nodes are their parents *children*. In the following procedure, we search for an appearance with a name specified by the name argument, and then group all similarly named appearances underneath it. TreeNode methods will be highlighted in **bold**.

```
proc findAndGroupSimilar {name} {
    # find the appearance matching "name"
    set parentApp [slim GetAppearances -name $name]

    # get the root of the appearance's tree (a palette)
    set plt [$parentApp GetRoot]

    # get the list of Chrome's current children
    set children [$parentApp GetChildren]

    # find all appearances that start with $name
    foreach app [$plt GetAppearances -name "${name}*"] {
        # skip the original
        if {$app == $parentApp} {
            continue
        }
```

```
        # check if this appearance is already a child of Chrome
        if {-1 == [lsearch $children $app]} {
            # app isn't in the list, make it a child of Chrome
            $parentApp AddChild $app
        }
    }
}
```

In the code above, we used TreeNode methods **GetRoot**, **GetChildren**, and **AddChild**. These methods are defined for any class that inherits from TreeNode. The classes we discuss below all inherit from TreeNode. A complete list of methods defined for the TreeNode class can be found in the Slim Scripting Glossary. To visualize the whole tree of classes that inherit from TreeNode, refer to the Class Hierarchy Diagram.

Palette Objects

Palettes are containers of appearances and other palettes. In Slim your can have any number of Palettes and these can be organized hierarchically. You can search palette objects for appearance objects that match various criteria. You can also encapsulate the state of a palette into an ASCII string.

The following procedure finds all of the palettes active in a scene, adds a new node specified by tmplt argument, and then saves any palette files that are external:

```
# call this template with a valid template ID, e.g. "pixar,RIBBox#0"
proc addTemplateToAllPalettes {tmplt} {
    # loop through all top-level (non sub-) palettes
    foreach pal [slim GetPalettes] {

        # check if this template exists in the palette already
        # if it does, skip
        if {[$pal GetAppearances -template $tmplt] != ""} {
            continue
        }

        # create an appearance using the template
        $pal CreateAppearance -template $tmplt

        # if this is an external palette, save it
        if {[$pal GetStorageType] == "external" && \
            [$pal GetAccessMode] == "rw"} {
            $pal Save [$pal GetFilename]
        }

        # force a redraw of the editor
        $pal UpdateEditor
    }
}
```

In the example above, we used the methods **GetAppearances**, **CreateAppearance**, **GetStorageType**, **GetAccessMode**, **Save**, **GetFilename**, and **UpdateEditor**. Here are some other useful methods defined for Palette objects:

plth DeleteDisconnected

Delete any nodes in the graph that are not *attachable* and are not connected to any other attachable nodes.

plth Edit

Raise the Palette Editor for this object.

plth SelectAppearances applist

Select the appearances specified by applist.

A complete list of methods defined for palettes can be found in the Slim Scripting Glossary.

Appearance Objects

An Appearance is a container of properties and is the base class for the various appearance types in Slim. **Functions** are a type of appearance that can participate in custom shader generation. Every function object has an associated **Template** object which itself is a type of appearance. **Instances** are a type of appearance that can't participate in custom shader generation. All of these classes of objects share a lot of behavior including the ability to search for properties that match various criteria.

As an example, the following procedure finds the appearance with name matching the passed argument. It then creates a duplicate copy of the appearance, opens the appearance in the Appearance Editor, checks its shading rate, and renders a preview.

```
proc findCopyAndEdit {name} {
    # get appearances that match string
    set apps [slim GetAppearances -name $name]

    # loop through the appearances
    foreach app $apps {
        # create copy of appearance
        # the result may be a list if $app has connected nodes
        set newApps [$app Duplicate]
        set newRoot [lindex $newApps end]

        # make sure shading rate is set to 1 (or lower)
        set curRate [$newApp GetPreviewShadingRate]
        if {$curRate > 1} {
            $newApp SetPreviewShadingRate 1
        }

        # raise in Appearance Editor
        $newApp Edit

        # render preview
        $newApp PreviewRender
    }
}
```

The procedure above made use of the **Duplicate**, **GetPreviewShadingRate**, **SetPreviewShadingRate**, **Edit**, and **PreviewRender** methods. These are some other useful methods for appearances:

apph Attach ?subtype : ""?

Attach the appearance to the current Maya selection. This is only meaningful if called when Slim is connected to MTOR.

apph Detach ?subtype : ""?

Detach the appearance to the current Maya selection. This is only meaningful if called when Slim is connected to MTOR.

apph GetProperties ?arg arg ...?

Get the list of an appearance's properties that match the search criteria. This will be covered further in the Property Objects section (below).

apph RevertValues

Revert all properties in the appearance to their respective default values.

apph UpdateEditor

Update the Appearance Editor for the appearance.

A complete list of Appearance methods, along with methods specific to Function objects, can be found in the Slim Scripting Glossary.

Property Objects

Properties are where most of the information associated with appearances are stored. As with appearances, the property class is a base class for other subclasses. Collections, Parameters, Attributes and TORAttributes are the subclasses of Property. **Collections** are used to organize properties hierarchically. **Parameters** are those properties that are actual appearance parameters. **Attributes** are those properties that have a RIB representation that isn't part of a shader instance. **SlimAttributes** can be used to control code generation. **TCLAttributes** contain tcl code that may be executed within a TCLBox. **TORAttributes** are properties that control how additional rendering passes are performed.

Properties exist inside of appearances, and are accessed using the **GetProperties** appearance method. This method can be used with a number of different switches to filter the properties returned. Given a variable apph set to the object handle of an appearance, you might use GetProperties in the following ways:

```
# get all input properties with name "Ks"
$apph GetProperties -access input -name Ks

# get all float input parameters
$apph GetProperties -type float -access input -class ::Slim::Parameter

# get all external parameters
$apph GetProperties -provider variable -access input -class ::Slim::Parameter
```

Because this method is defined for appearances, you will only be operating on the properties of one appearance at a time. If you want to operate on the properties across a palette, or an entire session, you must do so by nesting loops. The example given in our explanation of The Slim Object does just this.

The following procedure copies property settings from one node to another. Property correspondence is done by name, so the nodes do not have to be of the same template.

```
proc copySettings {fromName toName} {
```

```
        # get source appearance
        set fromApp [slim GetAppearances -name $fromName]

        # if this does not return exactly one appearance,
        # print an error and exit
        set numFrom [llength $fromApp]
        if {[llength $fromApp] != 1} {
            ::RAT::LogMsg ERROR "copySettings : fromName must matches $numFrom appearances"
            return
        }

        # loop through destination appearance(s)
        foreach toApp [slim GetAppearances -name $toName] {

            # loop through properties in fromApp
            foreach fromProp [$fromApp GetProperties -access input] {

                # find properties in toApp with the same name
                set name [$fromProp GetName]
                foreach toProp [$toApp GetProperties -access input -name $name] {
                    if {![$toProp isa ::Slim::AlterEgo::coll]} {
                        # copy value
                        $toProp SetValue [$fromProp GetValue]
                    }

                    # copy value provider
                    set provider [$fromProp GetValueProvider]
                    $toProp SetValueProvider $provider

                    # if it's a connection, set that
                    if {$provider == "connection"} {
                        $toProp SetConnection [$fromProp GetConnection]
                    }
                }

                # log copy
                ::RAT::LogMsg INFO "Copied settings from [$fromApp GetName] to [$toApp
GetName]"
            }
        }
    }
```

The example above made use of the **GetName**, **GetValue**, **SetValue**, **GetValueProvider**, and **SetValueProvider** methods for properties. These are some other useful methods for properties:

proph GetAppearance

Get the object handle for the appearance that uses this property.

proph GetConnectedFunction

Get the object handle for the function to which this property connects.

proph RevertValue

Revert the value for this property to its default.

A complete list of methods defined for Properties, especially those specific to AtomicProperties (those with a value), can be found in the Slim Scripting Glossary.

You might have noticed the `isa` query in the example above. This is a standard method available to all classes, and you can use it to check the class of an object. If you ever see the name of an object handle (e.g. coll153), you will find that the prefix of the handle is its class. A fully qualified class for a scripting object will be `::Slim::AlterEgo::`*`class`*, e.g. **::Slim::AlterEgo::coll**. You can query the class of any object using the **info** method, e.g.:

```
% coll161 info class
::Slim::AlterEgo::coll
% coll161 isa ::Slim::AlterEgo::coll
1
```

An complete index of classes can be found in the Slim Scripting Glossary.

The Workspace Object

This class is responsible for managing issues related to the current workspace. As with the application object, there should only be a single instance of this class and its handle is **workspace**. You can use the workspace object to convert path names from global to relative references using the current search paths. The workspace state is manipulated by the Workspace Editor and is managed across applications. For this reason it may not be sufficient to simply set the workspace state using the workspace object.

Methods for the workspace can be found in the Slim Scripting Glossary. For more details on working with workspaces, please refer to the workspace editor docs.

Output

You may find it useful to output information while your script is running, either to communicate to the user, or to help with debugging information.

Because Slim is designed to run in conjunction with a client application like Maya, the stdout channel is not usable for output. Instead, you can direct information to Slim's message log.

The advantage of using Slim's message log is that each message is automatically timestamped and can be coded according to' severity. You can log it using the **::RAT::LogMsg** procedure. Just send your message, preceded by its severity:

```
::RAT::LogMsg COPIOUS "Reading from testdata.tcl"
::RAT::LogMsg INFO    "Creating a new appearance"
::RAT::LogMsg NOTICE  "This appearance has been changed"
::RAT::LogMsg WARNING "This appearance contains a very low shading rate"
::RAT::LogMsg ERROR   "Cannot generate shader"
::RAT::LogMsg SEVERE  "Cannot communicate with Maya"
```

User Data

As you develop scripts, you may find it useful to associate extra information with an appearance or with a palette. The User Data mechanism allows you to associate arbitrary information with any node in Slim.

Every node in Slim has a user data *dictionary*, or data that is stored and retrieved using a *key*. Any string can be used as a key. Slim makes no assumptions about its contents. User Data is only accessible through the scripting environment. It will not be visible anywhere in the interface.

As an example of how to use User Data, this procedure tags every active appearance with the current time, host, and user.

```
    proc storeCreationInformation {} {

        # loop through all appearances
        foreach app [slim GetAppearances] {

            # check for existing "createdBy" key
            set user [$app GetUserData createdBy]

            # if this has already been set, skip
            if {$user != ""} {
                continue
            }

            # store user name, time, and host
            $app SetUserData createdBy [GetUserName]
            $app SetUserData createdAt [clock seconds]
            $app SetUserData createdOn [GetHostName]
        }
    }
```

We can later retrieve that information using the procedure below:

```
    proc retrieveCreationInformation {} {
        # loop through all appearances
        foreach app [slim GetAppearances] {

            # check for existing "createdBy" key
            set user [$app GetUserData createdBy]

            # if this hasn't been set, skip
            if {$user == ""} {
                ::RAT::LogMsg NOTICE \
                    "Could not find creation information for [$app GetName]"
                continue
            }

            ::RAT::LogMsg INFO "Creation information for [$app GetName]"
            ::RAT::LogMsg INFO " - created by $user"

            # get time and format it
            set time [$app GetUserData createdAt]
            if {$time != ""} {
                ::RAT::LogMsg INFO " - created at [clock format $time]"
            }
```

```
        # get host
        set host [$app GetUserData createdOn]
        if {$host != ""} {
            ::RAT::LogMsg INFO " - created on $host"
        }
    }
}
```

Commands

Commands allow you to present your scripts as items in the Commands menus of the Appearance Editor and the Palette Editor. The following trivial example demonstrates how to create a Command:

Creating the .slim File

Commands, like templates, are defined in .slim files. A declaration of a custom command consists of an invocation — how the command should appear and the tcl procedure to call when it is invoked — and a body of TclTkSource — a container in which to define tcl procedures. This example custom command will be displayed in the Appearance Editor and, when invoked, will log all of the values of the current appearance to Slim's message log.

```
##
## Creating a Custom Command
##

# Tell Slim this is an extension
slim 1 extensions pixardb {
    extensions pixar pxsl {

        # Begin custom command
        cmdui printValues {

            # Place the command in the Command Menu of the Appearance Editor.
            # The first argument defines how the item will appear.
            # The second argument defines the procedure to call.
            # Note %c will automatically be substituted for the
            # handle of the current apperance or palette.

            invocation {AppearanceEditor/Resources/Print Param Values} {::Slim::printValues %c}

            TclTkSource {

                # Here's the tcl command
                # Because we used "%c" in our invocation above,
                # The current context (in this case, the current appearance)
                # will be passed as the first argument

                proc ::Slim::printValues app {

                    ::RAT::LogMsg INFO "Printing parameter values for [$app GetName]"

                    # loop through parameters, printing their values
```

```
                    foreach p [$app GetProperties -access input -class ::Slim::Parameter] {
                        ::RAT::LogMsg INFO " - [$p GetLabel] : [$p GetValue]"
                    }
                }
            }
        }
    }
}
```

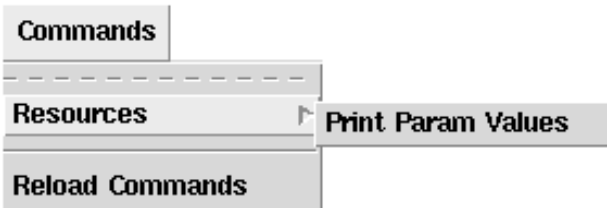
Loading the .slim File

Like templates, custom commands must be loaded in an initialization file. You use the **LoadExtension** procedure, followed by the **customcmd** class, as below:

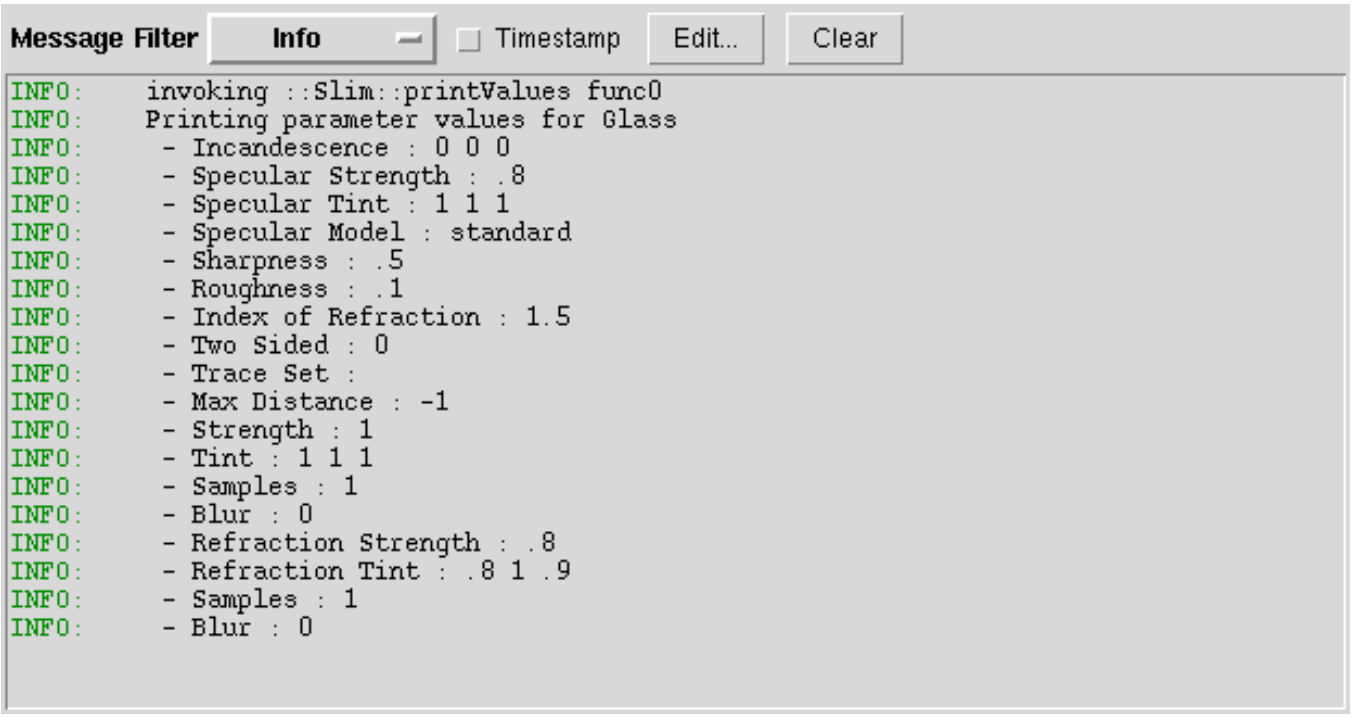
```
LoadExtension slim [file join $templates custom_commands.slim] customcmd
```

Running the Command

After restarting Slim, our command should be positioned in the Commands menu as specified by its `Invocation`:



Executing the "Print Param Values" command yields these results in the message log:



Customizing the Invocation

Optional arguments to the invocation command allow you to further customize how your Command appears in the menu:

-accelerator *binding*

Establishes a hot-key to invoke your command. The binding should be declared like a Tk binding, e.g. `<Control-m>`.

-enabled *expression*

Controls whether the menu item is enabled. The argument is a TCL expression that will be evaluated when the menu is posted. As an example, you can disable commands that may depend on Maya/MTOR using:

```
-enabled {[string match [slim GetRunMode] "server"]}
```

You can use %c in your expression, and it will be replaced by the context (that is, the appearance or palette) of the editor. Remember to enclose your expression in braces so it is evaluated when the menu is posted, and not before.

Prev | Next

Scripting Glossary

Glossary Overview

This file contains a complete glossary of the methods and procedures accessible from the scripting environment. This is a reference document. For more of an introductory environment, see the Introduction to Slim Scripting.

As mentioned in the introduction, Slim's scripting interface is also used by MTOR for communication with Slim, and by Slim itself when parsing files. It is for this reason that some methods available in the slim scripting environment may not actually be useful for scripting the behavior of Slim and are not germaine when Slim is used with RenderMan for Maya Pro.

Methods that fit this criteria are listed here but with minimal documentation and are formatted in ***italics***. For example:

objh ***GetObject***

objh ***RemoveAlias***

objh ***SetObject*** *newObject*

> These methods are used to establish the scripting environment, and while available for all objects, are not generally useful.

The Slim Object

These are the methods defined for the **slim** object:

slim ***BackupPalettes***

Create backup versions of all palettes loaded. This method is called by MTOR when it detects that Maya has crashed.

slim **BuildShaders** *?arg arg ...?*

Generate and build all shaders associated with attachable functions.

Optional arguments:

-dirty

Only act upon functions that have been changed since the shaders have last been generated

slim ***Clear***

Completely reset Slim's state. The method is called by MTOR when it opens a new scene file.

slim ***ClientCmd*** *cmd*

slim ***ClientMsg*** *msg*

Send a command or message to MTOR.

slim ***ClientSelection*** *app list ?applistcomputed?*

This method is used by MTOR to inform Slim that the selection in Maya has changed.

slim **CommitCmds**

`CommitCmds` will:

1. Flush the undo queue and commit all commands.
2. Remove all deleted Appearances from memory.
3. Remove all deleted Palettes from memory.

In the general case, the user will want to call `update` immediately before calling `slim CommitCmds`.

slim **CreatePalette** *?arg arg ...?*

Create or open a Slim palette. One of the following arguments is required:

-new

Create a new, untitled palette

-file *filename*

Open the palette specified by filename

-pickfile

Select a palette to open using the File Picker

Optional arguments:

-edit

Edit the palette that has been created or opened in the Palette Editor

slim ***Encapsulate*** *?arg arg ...?*

Provides a complete encapsulation of the current state of Slim. This is used by MTOR to save Slim data into the Maya scene. The state can be restored using the **Instantiate** method.

Optional arguments:

-full

Rather than just referencing them, fully encapsulate external palettes

-export {applist}

Only encapsulate appearances in the specified list. External palettes are fully encapsulated.

slim **FindPalette** *?name?*

Returns the object handle for the palette with the given name. If no name is provided, the first palette handle is returned. If a match cannot be found, the empty string is returned.

slim **GetAppearances** *?arg arg ...?*

Returns the list of appearances from all palettes which match the criteria specified in the arguments. The criteria can use glob style notation, which means that you can use the standard wild card characters: *, ?, etc. If no arguments are provided, the result is equivalent to the request:

```
slim GetAppearances -name * -id * -master * -template * -type * -selected
ignore -dirty ignore
```

Criteria are specified by the following optional arguments:

-name namestring

Appearances with names matching the specified string

-id idstring

Appearances with unique id matching the specified string

-master masterstring

Appearances with master matching the specified string

-template templatestring

Appearances with template matching the specified string. This is only meaningful for Functions.

-type typestring

Appearances of type matching the specified string. You can determine the list of recognized types using the `slim GetTemplates -types` command. In addition to these types, you can use any of the following *meta-types*:

_Attachable

appearances of appropriate type for attaching to objects.

_NotAttachable

appearance not appropriate for attaching to objects.

_AllSpecial

attachable appearances that follow the standard *TOR naming convention (that causes their inclusion in the RIB stream.

_Declare

appearances that obey the standard *TOR naming convention that causes their inclusion into the declaration portion of the RIB stream.

_Frame

appearances that obey the standard *TOR naming convention that causes their inclusion into the frame portion of the RIB stream.

_World

appearances that obey the standard *TOR naming convention that causes their inclusion into the world portion of the RIB stream.

_Frontplane

appearances that obey the standard *TOR naming convention that causes their inclusion

into the frontplane portion of the RIB stream.

_Backplane

appearances that obey the standard *TOR naming convention that causes their inclusion into the backplane portion of the RIB stream.

-selected (1 | 0 | ignore)

Appearances matching the specified selection state, which is one of 0 (unselected), 1 (selected) or "ignore" (either).

-dirty (1 | 0 | ignore)

Appearances matching the specified dirtiness state, which is one of 0 (clean, doesn't require regeneration), 1 (dirty, requires regeneration) or "ignore" (either). This criterion is only meaningful for appearances of class function and type _Attachable (see above).

Example uses:

```
slim GetAppearances -type float -name Fractal*
```

Returns objects handles for all float appearances with names starting with "Fractal"

```
slim GetAppearances -id 1000_TLum8400000
```

Returns the object handle for the appearance with the specified unique id

```
slim GetAppearances -template pixar*
```

Returns object handles for all appearances which use the factory templates

slim ***GetClientSelection*** *apptype*

Returns a list of names of items selected in Maya that would have resulted from an attachment of type apptype.

slim **GetDoGUI**

Returns whether Slim's user interface is visible, or whether it is being run in batch mode.

slim **GetEncapsulation** *obj*

Returns an encapsulation of the specified object. This can be used in combination with the **Instantiate** method.

slim **GetEquivalentTypes** *type*

Attachable types like Surface may have several types mapped to them. This method returns all of the types that have been defined to map to the specified type.

Example usage:

```
% slim GetEquivalentTypes surface
shadingmodel shadingmodel_aov visualizer
```

slim **GetPalettes** *?arg arg ...?*

Get object handles for all top-level palettes (i.e., subpalettes are excluded).

Optional arguments:

-visible

Only returns palettes that have not been marked as hidden

slim **GetPresentableTemplateName** *tmpltid ?arg arg ...?*

Given a template identifier, returns a version suitable for displaying to the user. This will match the string that the user sees elsewhere in the Slim interface. If the user has turned on Expert Menus, the version will be indicated by parentheses.

Optional arguments:

-showVersion (1 | 0)

Explicitly state whether the version is included

Example usage (in this example, the user has Expert Menus on):

```
% set app [slim GetAppearances -name Fractal*]
func5
% set tmpltID [$app GetTemplateID]
pixar,C_Fractal#0
% set tmpltLabel [slim GetPresentableTemplateName $tmpltID]
Fractal (v0)
```

slim ***GetRIB*** *id ?flags?*

Invokes the GetRIB method on the appearance with the specified unique ID. This is used by MTOR for rib generation.

slim **GetRunMode**

Returns whether Slim is being run in conjunction with RfM ("server") or as a standalone application ("solo").

slim ***GetSlimStatus***

Always returns the message "Slim is active." This is used by MTOR to verify that Slim is operational.

slim **GetTemplates** *?arg arg ...?*

Get a list of all active templates.

Optional arguments:

-name name

Only return templates with name matching the specified string

-type type

Only return templates of the specified type

-showhidden

Include hidden (obsolete) templates in the search. These templates are usually excluded.

-return (name | alterego) Controls whether the returned list consists of template identifier strings like pixar,ColorSpline#0 ("name", the default) or object handles like tmplt42 ("alterego").

-types

Just return a list of the types encompassed by active templates. This should be used without any other options.

slim **GetWorkspace**

Returns the workspace object.

slim ***Instantiate*** *state modes*

Instantiate Slim with the specified state string. This is how MTOR initializes Slim when opening a scene file.

slim ***InstantiateGUI***

Initialize the user interface for Slim. This creates all of the palette editors when opening a scene file.

slim **JournalCmd** *?arg arg ...?*

Execute a command and journal it. That is, make it undoable. Journaling works by specifying a command to execute, a command to execute if the user hits undo, and (optionally) a command to execute if the user hits redo.

Arguments:

-doit command

The command to execute

-undoit command

The command to execute if the user hits Undo

-redoit command

A command to execute if the user hits Redo. If this is not specified, the -doit command will be executed. This option can be useful, however, if there is a way to perform this command more efficiently after it has been undone.

-title titlestring

The string to use in the Undo/Redo menu entries

-cmdkey identifier

You will notice that every editor has its own independent Undo/Redo stack. This key, the unique ID for the palette or appearance being edited, identifies which stack to use.

Most methods invoked by the user interface (e.g. SetValue) are automatically journalled. If you find that a method used in a script isn't journalled, you can use this command to make it undoable. More commonly, though, this method is used to gather together several journalled methods into one action.

For example, the following script operates on three different parameters and sets them to zero:

```
set kd [$apph GetProperties -name Kd]
set ks [$apph GetProperties -name Ks]
set kr [$apph GetProperties -name Kr]
$kd SetValue 0
$ks SetValue 0
$kr SetValue 0
```

SetValue is automatically journalled. But if a user were to execute this script and then change his mind, he'd have to hit Redo three times. The following code groups this code into one action by disabling the journalling in SetValue and explicitly specifying undo behavior.

```
proc SetValues {p1 v1 p2 v2 p3 v3} {
    # temporarily disable journalling
    set old [::RAT::SetJournalingState 0]
    $p1 SetValue $v1
    $p2 SetValue $v2
    $p3 SetValue $v3
    # restore journalling
    ::RAT::SetJournalingState $old
}

set kd [$apph GetProperties -name Kd]
set ks [$apph GetProperties -name Ks]
set kr [$apph GetProperties -name Kr]
```

```
set curKd [$kd GetValue]
set curKs [$ks GetValue]
set curKr [$kr GetValue]

slim JournalCmd \
        -doit "SetValues $kd 0 $ks 0 $kr 0" \
        -undoit "SetValues $kd $curKd $ks $curKs $kr $curKr" \
        -title "Zero Out K Values" \
        -cmdkey [$apph GetID]
```

JournalCmd is most frequently used in CustomUIs, which must often group several low-level commands into one action. You will find several example uses of JournalCmd in the CustomUIs that ship with Slim.

slim **LoadWorkspaceFile** *dir file*

Set the workspace root to *dir* and read the specified workspace file.

slim ***MakePalettesInternal*** *names*

This method is used by MTOR when importing a scene file

slim ***ModalMsg*** *msg*

slim ***ModalStart*** *title file text ?arg arg ...?*

slim ***ModalStop*** *?arg arg ...?*

These methods are used for communication from MTOR to Slim.

slim ***NewCmdUI*** *vendor nm*

slim ***NewCustomUI*** *vendor nm*

slim ***NewExpressionUI*** *vendor type nm*

slim ***NewPalette*** *id*

slim ***NewPaletteReference*** *filename*

slim ***NewTemplate*** *type nm vendor prefix version*

These method are used by the SlimParser when reading a palette file.

slim **ReadSlimFile** *file ?context?*

Read the specified Slim file into the context given by *context*. If not provided the file is read into the global context. You can use this command to load additional Slim templates anytime Slim is running. You can also cause appearances or palettes encapsulated within a slim file to be imported into an existing palette by specifying the container palette's handle as the context.

slim **ReadSlimString** *string ?context?*

This operates like ReadSlimFile, only reads data directly from the specified string.

slim **ReloadExtensions** *?class?*

Reload extensions to Slim. Class is the class of extension, which you will find in the LoadExtension calls in slim.ini:

slim

all "slim" extension files

customcmd

all custom commands

customui

all customui definitions

With the class argument omitted, this will reload all extensions

slim **SavePrefs**

Write user preferences to disk. This is usually performed when Slim shuts down.

slim ***SetClientSelection*** *app ?type?*

This method is used by Slim for Pick Objects in Maya.

slim ***SetGuiContext***

slim ***SetProjectLocation*** *location*

slim ***SetSceneLocation*** *location*

These methods are used by MTOR to intialize Slim.

slim **Shutdown** *?force?*

Shut down Slim. This is usually used by RfM to indicate that Maya is shutting down.

slim ***Uniqueify*** *mapping*

This method is deprecated.

slim **UpdateGUI** *?arg arg ...?*

Same as the tcl/tk "update" builtin

slim **WindowCmd** *cmd win ?arg arg ...?*

Issue a command to a specified window

Example uses:

% **slim WindowCmd raise all**

Raise all slim windows

% **slim WindowCmd hide all**

Hide all slim windows

% **slim WindowCmd show Preferences**

Show the preferences window

The Workspace Object

These are the methods defined for the **workspace** object. For more background on workspaces, please refer to the Workspace Editor documentation.

workspace **AddDirMap** *fromPath toPath zone*

Add a directory mapping from *fromPath* to *toPath*. The mapping will only be applied on systems in the specified *zone*. Directory maps are also applied by **workspace ResolvePath *path***.

The following is an example of two directory mappings that would allow any paths beginning with /users to always be replaced with C:/users on windows machines, and vice versa on unix machines:

```
% workspace AddDirMap /users C:/users UNC
% workspace AddDirMap C:/users /users NFS
```

workspace **AddSearchPath** *p ?which?*

Add the path *p* to the search paths. The optional argument *which*, if provided, should be either `local` or `server`, which indicates whether to add the path to the project or server paths. If not provided, the default is to add to the project paths.

workspace **ApplyDirMaps** *path*

Apply directory mapping to the specified path. The first mapping with a *fromPath* that matches a portion of *path* will be applied. Returns the mapped path.

workspace **BuildDirs**

Create all of the directories associated with the workspace definition.

workspace **ClearDirMaps** *?arg arg ...?*

Clear directory mappings. With no additional arguments, all directory mappings are cleared.

Optional arguments:

-index i

Delete the directory mapping with the specified index in the list of mappings.

-mappattern pattern

Deletes any directory mappings which match the specified glob-style pattern. For example:

```
% workspace ClearDirMaps -mappattern {C:/usr /usr NFS}
% workspace ClearDirMaps -mappattern {* * UNC}
```

workspace **ClearSearchPath** *p ?which?*

Remove the path *p* from the searchpath list associated with *which*. The optional argument *which*, if provided, should be either `local` or `server`, which indicates whether to clear the path from the project or server paths. If not provided, the default is to clear from the project paths.

workspace ***CreateTempWSFile***

Create a temporary workspace file. This method is used for deferred ribgen

workspace **GetDir** *name ?dosubst? ?createit?*

Determine the location of the abstract directory type given by *name*. The list of known abstractions can be obtained via the **GetDirNames** method. The optional *dosubst* argument controls whether variable substitution is performed on the directory value, and defaults to 0 (do not perform substitution). The optional *createit* argument, if set to 1, causes the directory to be created; it defaults to 0.

workspace **GetDirMapZone**

Determine the zone in which the system is operating. The system's zone is specified by a preference called WSDirMapZone, which is set in the default RAT.ini. The default zoning scheme is based on filesystem type. Windows systems are in the UNC zone, while all other systems are in NFS. The list of all possible zones is specified by the pref called WSDirMapZoneList, also set in RAT.ini.

workspace **GetDirMapZoneList**

Get the list of possible zones in which directory mappings can be applied. Each directory mapping must

be associated with one zone. A list of possible zones is given by the preference called WSDirMapZoneList, which is set to {NFS UNC} in the default RAT.ini. As you may have guessed, the default zoning scheme is based on file system type. Unices would be NFS and windows would be UNC. A systems's zone is specified by the preference called WSDirMapZone, also set in RAT.ini.

workspace **GetDirMaps**

Get the list of directory mappings. Each element in the list a sub-list containing a {to from zone} triplet. Here's an example list of dirmaps (paths containing spaces are surrounded by an extra set of braces):

```
{{foo bar UNC} {bar foo NFS} {{C:/Documents And Settings} /users NFS}}
```

workspace **GetDirNames**

Get the list of names of project resource locations. Use **workspace GetDir *name*** to get the actual directory name of the resource location, relative to the project root.

workspace **GetProjectName**

Get the name of the workspace file.

workspace **GetRootDir**

Get the current root directory. This method is equivalent to: **workspace GetDir WSRoot**.

workspace **GetSceneDir**

Get the directory in which the Maya scene file resides.

workspace **GetSearchPathTypes** *?which?*

Get the list of resource types (`shader`, `texture`, etc.) for which searchpaths are defined. The *which* argument specifies whether to check `local` (the default) or `server` searchpaths.

workspace **GetSearchPaths** *which ?dosubst? ?fmt?*

Get the current project or server search paths depending on the *which* argument. To get the default project or server search paths, *which* should be `local` or `server`, respectively. Or to get the search paths used by a particular resource type, the *which* argument can be the name of the resouce type, like `shader` or `texture`. Note that if the search paths for a resource type are emtpy, the default will be used instead. The optional *dosubst* argument indicates whether to perform variable substitution when set to 1 (it defaults to 0). You can choose the format of the return value using the optional *fmt* argument which can be one of: `list` or `RIB`; the default is to return a list.

workspace **GlobalizePath** *subpath*

Prepend the current project root directory to the relative pathname given by *subpath*, and return the result.

workspace **IsLocked** *type value*

Determines whether the specified type of path is locked.

Possible values for *type*:

searchPaths

Check whether the searchpath list specified by *value* (one of `server` or `local`) is locked.

resourceLocations

Check whether the location of the resource specified by *value* is locked

workspace **LocalizePath** *path ?w?*

Convert an absolute file reference given by *path* to a relative one by stripping off known leading path components. You can choose to strip only if the path is below the current workspace root by setting the optional argument *w* to `workspace`. The default behavior is to consider all directories in the current local search paths.

workspace **LockDir** *name location*

Operates like **SetDir** method, but locks the directory to prevent future changes.

workspace **LockRootDir** *d*

Set and lock the root directory.

workspace **LockSearchPaths** *p ?which?*

Operates like **SetSearchPaths** method, but locks the searchpaths to prevent future changes.

workspace ***QuerySettings***

Encapsulate the current state of the workspace object.

workspace **ReloadProjectFile**

Reload the last workspace file associated with the current workspace root. i.e., revert to the saved version of the current workspace file.

workspace **ResolvePath** *path ?which? ?extensions?*

Determine a fully qualified pathname to the file given by *path* by traversing search paths to resolve the file. This command returns the empty string if the file can't be found. The *which* argument can be used to indicate that only the searchpaths for the given resource type should be searched. Otherwise the default `local` search paths are used. You can also provide an optional list of space separated potential extensions (in a single string) which are appended to the filename during the search; for example:

```
% workspace ResolvePath fish {} "tif tex"
```

will search for fish.tif and fish.tex in the default searchpaths. A list of the found files (full pathnames) will be returned.

```
% workspace ResolvePath fish texture "tif tex"
```

will search for fish.tif and fish.tex in the texture searchpaths.

Also note that directory mappings will be applied to the *path* before it is resolved. For example, if a dirmap exists which maps /home to C:/home in the UNC zone, then the following command on a UNC system:

```
% workspace ResolvePath /home/somefile
```

could return:

```
C:/home/somefile
```

workspace **RestoreDefaultPaths**

Restore searchpaths to application defaults.

workspace **RestoreDefaultSetupFiles**

Restore setup files to application defaults.

workspace **SetDir** *name location*

Sets the resource abstraction given by *name* to the value given by *location*. These abstractions are used to determine where rendering related resources are to be found; the list of known abstractions can be obtained via the **GetDirNames** method.

workspace **SetDirMaps** *maplist*

Set the list of directory mappings to the specified *maplist*. Each entry in the list must be a {to from zone} triplet (list). Any pathnames that contain spaces should be surrounded by an extra set of braces. For example:

```
% workspace SetDirMaps {{foo bar UNC} {bar foo NFS} {{C:/Program Files/goo} /
users/goo NFS}}
```

workspace **SetProject** *root ?projectFile?*

Set the current workspace root and initialize the workspace object's state using the contents of the last workspace file in use at that location or the file specified via the optional projectFile argument.

workspace **SetSearchPaths** *p ?which?*

Setthe search paths in the workspace object to *p*, which should be a single string (or list) containing one or several space-separated paths. Paths containing spaces should each be enclosed in an extra set of braces. The optional argument *which* can be `local`, `server`, or the name of a resource type, like `texture` or `shader`. `local` and `server` indicate that the default project or server paths should be updated. If not specified the default is to modify the local project paths.

workspace ***UpdateScriptingState***

This method is used by MTOR to update the workspace's scripting state after making changes.

workspace **WriteWSFile** *p ?which?*

Write the current state of the workspace to the current workspace file.

TreeNode Objects

The TreeNode base class implements the support for organizing objects in tree structures. Refer to the Class Hierarchy diagram to see the list of classes that inherit from TreeNode. Because the classes representing the palettes, appearances, and parameters that you manipulate all inherit from this class, the following methods are common to all of them:

objh **AddChild** *c ?front : 0?*

Re-parent the specified object under this object. By default, the new child will be last in its parent's list of children. You can make the child the first in the parent's list by passing 1 for the *front* parameter. Note that you can use this command to move appearances and sub-palettes between palettes.

objh **GetChildren**

Get a list of the handles to the children of this object.

objh **GetDescendents** *mode*

Get the list of all descendents of this object. The descendents are sorted according to the *mode* argument which can take one of these values:

depthFirst

children appear before their parents.

prefixDepthFirst

parents precede children, but siblings follow children

breadthFirst

siblings appear before children

objh **GetDrawMode**

Get the draw mode for the object. For inlined functions, this is set to "inline".

objh **GetParent**

Get the handle to this object's parent.

objh **GetRoot**

Get the handle to the object that parents all objects in this object's tree.

objh **RemoveChild** *c ?delete : 1?*

Removes the specified child. By default, the child will also be deleted.

Note: using this command to delete a node may not remove all references to the child object, which can cause problems when working with appearances. We recommend that you use the **Delete** method to delete an appearance.

objh **SetChildren** *children*

Explicitly set the list of children for the object.

PaletteEntry Objects

PaletteEntry is another base class that defines a common set of functionality for palettes and appearances. The following methods are defined for these classes:

objh **Delete** *?remove : 0?*

Deletes the entry. By default, the entry is only *scheduled* for deletion, which allows it to be easily undeleted. Setting the optional *remove argument to 1 causes the entry to be immediately deleted. Using `Delete 1` repeatedly is not recommended, as `Delete 1` can be slow if called many times in a row. The recommended method for forcing the deletion of multiple objects is:*

```
foreach object $objects {
   $object Delete 0
}
update
slim CommitCmds
```

objh ***GenerateID***

Generate a new unique ID for the entry.

objh ***GetDescription***

Get the description for the object. The description is what is displayed when the user raises the help balloon for that object.

objh ***GetDisplayLevel***

Get the display level for the entry.

Possible levels:

***""** (empty string)*

The default. Always visible.

hidden

Always hidden.

expert

Only visible when the user has "Expert Menus" turned on.

objh ***GetID***

Get the unique ID for the entry.

objh ***GetLabel***

Get the label for the object. For palette entries, this is usually identical the name.

objh ***GetName***

Get the name of the object.

objh ***GetType***

Get the type for the entry.

objh ***GetUserData*** *key*

User data is additional information associated with a PaletteEntry. It is stored as a series of name-value pairs. Other than that, the contents are completely arbitrary. You may find it to be a valuable way to store extra information for your scripts, or to communicate information from templates to CustomUIs. UserData is not visible in the Slim interface.

GetUserData accesses the user data stored with the specified key. If no user data is defined for this key, the empty string is returned.

objh ***GetUserDataArray***

Get the entire array of user data. This can be useful when copying user data from one object to another.

objh ***IsDeleted***

Because of the undoable nature of Delete, nodes that have been deleted via the interface may still be accessible from the scripting environment. This method allows you to check for this case.

objh ***IsDirty***

Returns whether the node is "dirty" and needs to be saved.

objh ***IsHidden***

This method allows you to quickly check whether a node is hidden.

objh ***SetDescription*** *desc*

Set the entry's description to the specified string.

objh ***SetDisplayLevel*** *level*

Set the display level for the entry.

objh ***SetID*** *id*

Set the unique ID for the entry to id.

objh ***SetImage*** *imgbuf*

Set the image using the specified encoded string. This method is used when reading a palette file.

objh ***SetLabel*** *label*

Set the entry's label to the specified string.

objh ***SetUserData*** *key value*

Set the user data for key to value.

objh ***SetUserDataArray*** *data*

Set the entire array of user data. This can be useful when copying user data from one object to another.

objh ***Undelete***

Recover an entry scheduled for deletion. Note that this will not work if an entry has been immediately deleted.

Palette Objects

In addition to the methods defined in the PaletteEntry and TreeNode base classes, the following methods are defined for the objects representing the palettes and subpalettes in your scene:

plth ***ChangeSaveMode*** *mode*

Change the mode in which this palette is saved:

full

All property information is saved. This mode is compatible with previous versions of Slim.

minimal

Only changes to parameters are saved. This mode can only be read by Slim versions 6.5 and greater.

Note that when converting a "full" palette to a minimal one, Slim checks each field of a property to see if it matches the template. Matching fields are considered unset and not saved.

plth ***CreateAppearance*** *?arg arg ...?*

Creates a new appearance inside this palette.

Valid options:

-template *tmpltID*

Creates an appearance using the template specified by tmpltID. You can find the names of available templates with the `slim GetTemplates` command.

-file *filename*

Creates an instance of the shader given by filename, which must be the name of a `.slo` or `.slim` file.

plth ***DeleteDisconnected***

Delete any nodes in the graph that are not attachable and are not connected to any other attachable

nodes.

plth ***DeleteEntries*** *?arg arg ...?*

Deletes the appearances represented by any object handles listed. This method acts like the Delete method in that nodes are, by default, not immediately deleted. The optional `-remove` flag specifies immediate deletion.

plth ***DeleteUnused***

Remove of all unused appearances in the palette. Appearances are deemed unused if they are not attached to any geometry in the Maya scene. This method is only meaningful if Slim is is connected to MTOR.

Because it interacts with MTOR, this method cannot be called via MEL using `slimcmd`).

plth ***Edit***

Raise the Palette Editor for this palette.

plth ***GetAccessMode***

Get the write permissions for this palette:

rw

Palette is writable

ro

Palette is read-only

plth ***GetAppearances*** *?arg arg ...?*

Get the appearances in this palette matching the specified criteria. See the entry for `slim GetAppearances` specifics on criteria.

plth ***GetFilename***

Get the filename associated with an external palette. This filename is generally derived from the palette's label. Returns the string `untitled` if the palette is internal or has never been saved.

plth ***GetGraphArea***

This method is deprecated.

plth ***GetSaveMode***

Get the save mode for this palette. Possible modes are discussed in `ChangeSaveMode`.

plth ***GetStorageType***

Determine whether palette is ***internal*** *(stored with a Maya scene) or* ***external*** *(stored as a separate file).*

plth ***ImportInstance*** *file ?arg arg ...?*

Import the compiled shader (fully) specified by file. The optional `-render` flag triggers a subsequent preview render of the shader.

Determine whether palette is ***internal*** *(stored with a Maya scene) or* ***external*** *(stored as a separate file).*

plth ***Instantiate*** *slimstr*

plth ***InstantiateApp*** *slimstr*

*Instantiate the object that has been previously encapsulated in the string slimstr. Typically you would obtain slimstr from a call to **slim Encapsulate obj** with an appearance or sub-palette handle. (Though Instantiate and InstantiateApp are equivalent, InstantiateApp is the preferred usage.)*

plth ***Internalize*** *?arg arg ...?*

*Convert an external slim palette to an internal one. Note that because the file containing the external palette will still exist, this amounts to **copying** the data inside the palette. As such, Slim will regenerate unique IDs for all elements inside the palette to prevent any possible conflicts with the palette stored in the file. RfM will update the attachment data in the Maya scene.*

plth ***NewFunction*** *type nm templ*

plth ***NewInstance*** *type nm master*

plth ***NewPalette*** *id*

These methods are used by the SlimParser when reading a palette file.

plth ***Read*** *filename*

Read the palette encapsulation in filename into the palette object.

plth ***Revert*** *filename*

Revert the palette to a checkpoint file saved in filename. The checkpointed file should be one created using Save with the `-checkpoint` option enabled, though this is not strictly enforced.

The palette stored in the checkpointed file should have a unique ID identical to that of the current palette. Reverting to a palette with a different unique ID may result in the loss of shader attachments.

plth ***Save*** *filename ?arg arg ...?*

Save the palette into the the external file filename. If the palette is currently internal, it will be converted into an external palette.

Optional arguments:

-uniqueify

New unique IDs will be generated for the palette and its appearances. So long as Slim is able to communicate with MTOR, attachments will be updated.

-checkpoint

In addition to saving to the specified filename, a checkpoint of the palette will be made. The checkpoint palette is a backup copy, saved in the same location and with the same filename plus a ".bk#" suffix, where ".bk0" is always the most recent. For example, a palette named foo.splt might have checkpoints: foo.splt.bk0, foo.splt.bk1, etc. The number of checkpointed palettes maintained by slim is determined by the `CheckpointQueueSize` preference, which can be adjusted in the Preferences Editor. The -checkpoint flag will have no effect if the queue size is set to zero, which is the default.

-keepRef

By default, when a save occurs, the palette file reference is updated. The -keepRef flag disables that behavior, for cases when the user wants to save a palette to disk but maintain the existing palette reference in slim.

plth ***SelectAppearances*** *applist*

Select the appearances specified by applist.

plth ***SetAccessMode*** *mode*

Set the write permissions for the palette.

plth ***SetSaveMode***

Set the save mode for this palette. Note that this only used by the parser and will not perform any conversion for your palette. To change the save mode via scripting, use ChangeSaveMode.

plth ***SetGUIInfo*** *geom vis*

plth ***SetGraphVisible*** *v*

plth ***SetGraphVisible*** *v*

plth ***SetPaneSize*** *p s*

plth ***SetViewStyle*** *v*

These methods are used by the SlimParser when reading a palette file.

plth ***UndeleteEntries*** *?arg arg ...?*

Undelete the appearances represented by any object handles listed. This method acts like the Undelete method in that it will only undelete appearances that are scheduled for deletion.

plth ***UniqueifyAll*** *?arg arg ...?*

Generate new unique IDs for all appearances in the palette. If MTOR is connected, it will be updated with new unique IDs to maintain appearance connections. Unless the -keepnames flag is passed, each will receive a new name.

plth ***UpdateEditor***

Update the Palette Editor for the palette. It is sometimes necessary to invoke this method after making changes to the palette via scripting. If no editor is associated with the palette, a value of 1 is returned. Otherwise a value of 0 is returned.

Appearance Objects

Appearance is a base class representing Functions (dynamic nodes that generate a shader) and Instances (which refer to an existing shader).

The following methods are defined for appearances:

apph ***Attach*** *?subtype : ""?*

Attach the appearance to the current Maya selection. This is only meaningful if called when Slim is connected to MTOR.

The optional subtype argument is used to specify the subtype for volume shaders.

Because it interacts with MTOR, this method cannot be called via MEL using `slimcmd`. Instead, you can use the following MEL:

```
mtor control attach surface `slimcmd apph GetID`;
```

apph ***Detach*** *?subtype : ""?*

Detach the appearance to the current Maya selection. This is only meaningful if called when Slim is connected to MTOR.

The optional subtype argument is used to specify the subtype for volume shaders.

Because it interacts with MTOR, this method cannot be called via MEL using `slimcmd`. Instead, you can use the following MEL:

```
mtor control detach surface `slimcmd apph GetID`;
```

apph ***Duplicate*** *?arg arg ...?*

Create a duplicate of the appearance and store it in the same palette.

Optional arguments:

-destination *dst*

Put the duplicate appearance in the palette with object handle dst.

-update *val*

Specify whether to update the editor after duplicating. By default the editor will update.

apph ***Edit*** *?arg arg ...?*

Edit the appearance in the Appearance Editor. If the appearance is already being edited, this will raise the Appearance Editor.

Optional arguments:

-newwindow

Rather than using an existing Appearance Editor window, create a new Appearance Editor for this appearance.

-slimfile

Edit this appearance as a text file using the user's preferred text editor.

apph ***EditMaster*** *op*

This method is deprecated.

apph ***Export*** *filename*

Encapsulate and export the appearance to a file specified by filename.

apph ***GetAttachmentType***

Get the type of attachment (`ensemble`, `surface`, `displacement`, etc.) the appearance makes to geometry. Some appearance types like `shadingmodel` can be mapped to an attachment type like `surface`. These mappings are defined by the `SlimAttachableMap` preference set in `slim.ini`. For appearances that cannot be attached to geometry, the empty string is returned.

apph ***GetLighttype***

For light shaders, get the type of light (e.g. `distant`).

apph ***GetMaster*** *?dosubst : 0?*

Get the master of the appearance: the base string used to refer to the shader in the RIB file. Set the optional parameter dosubst to 1 to perform variable substitution on the returned result. If not specified, dosubst defaults to 0.

apph ***GetNamespace***

Get the appearance's namespace. Usually this is the empty string, indicating the global namespace. The namespace is usually set when the enclosing palette is loaded and is useful when storing palettes within Maya scene files undergoing import or reference operations.

apph ***GetPreviewFrame***

apph ***GetPreviewObjSize***

apph ***GetPrimitive***

apph ***GetPreviewShadingRate***

These methods get the settings used for preview renders in the Appearance Editor.

apph ***GetProperties*** *?arg arg ...?*

Get the list of an appearance's properties that match the search criteria, specified using one of the following flags. The criteria can use glob style notation, which means that you can use the standard wild card characters: `*`, `?`, etc. If no arguments are provided, the result is equivalent to the request:

```
apph GetProperties -name * -access * -type * -class * -provider * -private
```

Optional filtering criteria:

-access *pattern*

Use this to limit return properties based on their access (`input` or `output`).

-class *pattern*

Use this to limit return properties based on their class. Class must be fully specified, e.g. `::Slim::Parameter`.

-name *pattern*

Limit properties to only those with a name matching pattern.

-provider *pattern*

Limit properties to only those with a value provider matching pattern.

-type *pattern*

Limit properties to only those with a type (e.g. `float`, `color`, `string`) matching pattern.

Options:

-private

When limiting properties using any of the above tests, any parameters with provider `primitive` will by default be excluded. Use this switch to include them.

-first

By default, all properties matching the specified criteria will be returned. Use this option to return only the first property that matches the criteria. Because any subsequent properties are skipped, this option can sometimes improve performance when querying an appearance with a large number of properties.

apph ***GetPropertyTree***

Get the root of the tree of properties for this appearance. All parameters, collections, and attributes in the appearance are descendents of this property. You can use this method along with the GetChildren method to descend through the property tree.

apph ***GetSubtype***

Get the subtype of this appearance. Currently, this only applies to volume appearances, which can be defined to be of subtype `atmosphere`, `interior`, or `exterior`.

apph ***GetWorklist*** *?arg arg ...?*

Get a list of actions that must be performed before rendering. This applies to nodes like ImageFile, which must sometimes convert an image to a texture before it is used.

apph ***NewCollection*** *type nm ?front?*

apph ***NewParameter*** *type nm ?front?*

apph ***NewRIBAttribute*** *type nm ?front?*

apph ***NewSlimAttribute*** *type nm ?front?*

apph ***NewTCLAttribute*** *type nm ?front?*

apph ***NewTORAttribute*** *type nm ?front?*

Though these methods are primarily used by the SlimParser, they can also be used to add parameters to a dynamic template. The method creates a property and returns a handle to it that you can then use to specify more information.

Example usage:

```
set newProp [$apph NewParameter float Kd]
$newProp SetLabel "Diffuse Component"
$newProp SetDescription "Amount of diffuse light"
$newProp SetDefaultValue 1
```

For a real example, see the source of the AddParam customui.

apph ***PickObjects*** *?modes ""?*

Pick all objects to which this appearance is attached. With the `-adapt` flag, this script acts like the ***Pick Objects (Computed)*** *command. That is, attachable shaders that use this appearance (including adaptors) will also be considered.*

Because this method interacts with MTOR, it cannot be called from a `slimcmd`.

apph ***PreserveValues***

Mark all properties in the appearance as set. This will protect the values from any changes to templates.

apph ***PreviewRender***

Initiate a render of the preview swatch for this appearance.

apph ***Reload*** *?flags ""?*

Reload the source of the appearance, whether it be a template, or a shader or slim file on disk.

For functions, the Reload method will track when the template file was loaded and when it was last changed. By default, it will only successfully reload the template, if the file on disk has changed since the palette was loaded. To force a template reload, use the `-force` flag.

apph ***RevertValues***

Unset the values for all properties in the appearance editor. These properties will revert to their default value.

apph ***SetLighttype*** *t*

Set the type of light (e.g. `distant`) for this appearance. This is only applicable to light shaders.

apph ***SetMaster*** *m*

Set the master of the appearance: the base string used to refer to the shader in the RIB file.

apph ***SetNamepsace*** *nm*

Set the appearance's namespace. The namespace is usually set when the enclosing palette is loaded and is useful when storing palettes within Maya scene files undergoing import or reference operations.

apph ***SetSubtype***

Set the subtype of this appearance. Currently, this only applies to volume appearances, which can be defined to be of subtype `atmosphere`, `interior`, or `exterior`.

apph ***UpdateEditor***

Update the Appearance Editor for the appearance. This sometimes necessary when making changes to an appearance's properties. If no editor is associated with apph a value of 1 is returned. Otherwise a value of 0 is returned.

Function Objects

Functions are nodes created from a template, and are combined to create a shader. Function is a subclass of Appearance, so Functions support all of the Appearance methods listed above.

The following methods are defined for functions:

funch ***BuildShader*** *?arg arg ...?*

Generate and compile the shader for this appearance. By default, Slim will check the dirty status of the appearance and only continue if it has changed since the shader was last generated. You can override this and force shader generation by using the `-force` flag.

funch ***CreateInstance*** *?arg arg ...?*

Create a new Instance of this shader. By default, the Palette Editor will be updated. This behavior can be overridden by passing `-update 0`.

funch ***DeleteNetwork***

Delete this function and any function in its network. Technically, these functions are only scheduled for deletion, and may be recovered with either Undelete or UndeleteNetwork.

funch ***DirtyMaster***

Indicate that the shader for the function needs to be regenerated. For most changes to functions, Slim is able to detect dirtiness. This method allows one to force dirtiness for a situation that Slim does not handle. The layering customui, for example, makes use of this method when rearranging layers.

funch ***DuplicateNetwork*** *?arg arg ...?*

Duplicate the entire network for this function and store it in the same palette.

Optional arguments:

-destination *dst*

Put the duplicate appearance in the palette with object handle dst.

-update *val*

Specify whether to update the editor after duplicating. By default the editor will update.

funch ***Flatten*** *?arg arg ...?*

Create a new appearance by flattening an active function network into a single frozen instance. The new appearance will have a name based on the name of this appearance, and will be flattened according to the Instance Structure preference.

Optional arguments:

-name *nm*

Name the new appearance nm.

-file *f*

Write the shader to a filename specified by f.

-structure *structure*

Use the defined structure. Possible options are `flat` (all parameters in one list), `grouped` (parameters grouped by appearance), and `hierarchical` (parameters arranged in a hierarchy approximating the graph).

-update *val*

Specify whether to update the editor after flattening. By default the editor will update.

funct ***GetConnectedAttachables*** *?arg arg ...?*

Get the list of downstream attachable appearances that -- directly or indirectly -- connect to this appearance. By default, the returned list will consist of object handles. To get a list of unique IDs instead, pass `-return id`.

funct ***GetConnectionClients***

Return the list of properties that connect to this function.

funct ***GetConnectedFunctions***

Get the list of functions to which properties have connected. Even if multiple properties connect to the same function. This list is guaranteed not to have any redundant listings, and is therefore useful when writing recursive procedures.

funct ***GetInstances***

Get a list of instances that have been created from this function. Note that this will only return instances that exist in the same palette as the function.

funct ***GetSLSource*** *?language : rsl?*

Get the shading language source for this function.

funct ***GetSLSourceType*** *?language : rsl?*

Determine the kind of shading language generator used by this function (e.g. `StaticFunction`, `DynamicFunction`, `GenerativeFunction`, etc.)

funct ***GetObjectID***

Get the object handle for the template that begat this function.

funct ***GetTemplateID***

Get the template identifier (e.g. `pixar,SurfacePoint#0`) for this function.

funct ***IsDownRev***

Determine whether the template used by this function is obsolete. This is often used in conjunction with UpgradeTemplate

funct ***MasterIsDirty***

Determine whether the shader for this function needs to be regenerated and recompiled before it is used for a render. Slim will automatically dirty a master when, for example, changes are made to internal parameters. The master can also be dirtied via the DirtyMaster method.

funch ***SetHasInstances***

funch ***SetModificationTime***

funch ***SetState*** *s*

funch ***SetTemplateHints*** *fileref hints*

These methods are used by the SlimParser when reading a palette file.

funch ***UndeleteNetwork***

Undo the execution of Delete Network.

funch ***UpdateInstances***

Update all instances of this function. This command regenerates the shader (if necessary), and then reloads any instances of this function that are stored in the same palette.

funch ***UpgradeTemplate***

Upgrade the template used by this function to its latest version. You can determine whether the current template is obsolete by using the IsDownRev method.

Template Objects

The Template class is a subclass of Function. As mentioned above, you can access the Template object for a function via the GetTemplate method. For your convenience, however, this is usually not necessary. Most methods that access template information like GetSLSource can be called straight from the function.

Therefore, the methods specific to the Template class are not especially useful for scripting, but are listed below:

tmplth ***AddSLDirective*** *language type dir*

tmplth ***SetSLSource*** *language type src*

tmplth ***SetAdaptor*** *src*

These methods are all used by the SlimParser when reading template and palette descriptions.

Instance Objects

Instance objects represent appearances in shader that work with an existing shader. Instance objects are used to represent proper Instances created from functions, as well as any appearances created by importing a shader. Instances do not have templates. Their source is a shader on disk, and perhaps a function in the palette. As such, the methods specific to instances are fairly limited:

insth ***GetProgenitor***

Get the file or function which begat this instance. The file is returned for instances created from imported shaders. For proper Instances, the progenitor is the unique ID for the original function.

insth ***GetProgenitorFunction***

Get the object handle for the original function. If the instance was created from an imported shader, this method returns the empty string.

insth ***SetProgenitor***

Set the progenitor for this instance.

Property Objects

Property is a base class representing the Parameters, Collections, and Attributes inside of an appearance. You will mainly deal with subclasses of Property. The only true Property objects are the roots of the Property Tree for each appearance. This root, not visible in the user interface, is accessed using the GetPropertyTree method for appearances.

The following methods are common to all subclasses of Property:

proph ***BuildDefaultWidget*** *canvas win recommendedwidth height ?arg arg ...?*

Create the widget for this property as it is seen in the Appearance Editor. This is most often used from within a CustomUI to build a Value Editor for a property. Arguments detail the specific canvas and window information for the widget, along with suggested width and height. By default, the Property Menu will appear for any property without a default connection. You can override this behavior using the `-menu` switch. See some of the factory customui scripts for example usage.

proph ***BuildIcon*** *win*

Build the icon for this property. Like BuildDefaultWidget, this method is most often used from within a CustomUI.

proph ***CreateConnection*** *templateid*

Create an instance of the specified template and connect it to this property.

proph ***GetAccess***

Determine access -- whether it as `input` or `output` -- for this property.

proph ***GetAppearance***

Get the object handle for the appearance that uses this property.

proph ***GetCommandKey***

Useful when journaling a command, this method accesses the proper command key for the property. This is useful for properties inside of appearances that may be inlined.

proph ***GetConnectedFunction***

Get the object handle for the function to which this property connects. This method will only return the connected function if it is being used. That is, if the property's ValueProvider is not set to `connection`, it will return the empty string.

proph ***GetConnection***

Return the unique ID for the function to which this property connects. Unlike GetConnectedFunction, this method will return an ID as long as it exists. The ValueProvider is not consulted. For this reason, and because the unique ID is returned, rather than the (more useful) object handle, GetConnectedFunction is the preferred method to use when scripting.

proph ***GetConnectionStyle***

Get the connection style for the property. For inline connections, this will return "inline". Otherwise, this will return an empty string.

proph ***GetDetail***

Get the detail for this property. Possible values for detail:

uniform

Property cannot be connected to a function.

varying

Property can be connected to a function.

mustvary

Property must be connected to a function.

proph ***GetLabel***

Get the label for the object. The label is primarily used for interface, and can therefore contain special characters (including spaces).

proph ***GetLabelRef***

Get a reference to the variable which stores the label. You can use the result of this as the `-textvariable` *when creating Tk widgets in a CustomUI.*

proph ***GetName***

The name will often be used within RIB or shading language and must therefore resemble a variable.

proph ***GetMsgHandler*** *msg*

Get the property's msghandler defined for the specified message.

proph ***GetMsgHandlers***

Get the entire array of msghandlers for this property.

proph ***GetSubtype***

Get subtype for this property. Subtype is most often used to define a specific editor for a property. Properties representing a texture name, for example, are of type `string`*, but of subtype* `texture`*.*

proph ***GetUserData*** *key*

User data is additionalinformation associated with a property. It is stored as a series of name-value pairs. Other than that, the contents are completely arbitrary. You may find it to be a valuable way to store extra information for your scripts, or to communicate information from templates to CustomUIs. UserData is not visible in the Slim interface.

GetUserData accesses the user data stored with the specified key. If no user data is defined for this key, the empty string is returned.

proph ***GetUserDataArray***

Get the entire array of user data. This can be useful when copying user data from one object to another.

proph ***GetValueProvider***

Get the source of a property's value. Valid ValueProviders depend on the specific subclass of Property. This is the complete list of possible values:

constant

Constant value. For parameters, this indicates that the value is stored within the code of the shader. This option appears as `Internal` *in the user interface.*

variable

Value is defined by a variable. For parameters, this indicates that the value is a parameter of the shader and may be overridden by the RIB file. This option appears as `External` in the user interface. For (RIB) Attributes, a value of `variable` indicates that the attribute is disabled.

connection

Value is provided by a connected function. This option is only valid for non-uniform Properties and Connections.

expression

Value is provided by a TCL expression. This option appears as `TCL Expression` in the user interface.

primitive

Value is provided by a global parameter of the shader, like a PrimVar. When the property is an output, the primitive value provider is used for arbitrary output variables.

proph ***GetValueProviders***

Get the range of valid value providers for the property. An empty list (the default) signifies no change from the default behavior.

proph ***SetAccess*** *a*

Set whether this property is an `input` or `output`.

proph ***SetConnection*** *id ?arg arg ...?*

Connect the property to the function with the specified unique ID. When using this method, you must also make sure that the property's ValueProvider is set to connection. To disconnect the property from a function, pass the empty string as id. The optional args list is used by Slim for more customized undo behavior.

proph ***SetDetail*** *d ?default : ""? ?style : ""?*

Set the detail for the property, as defined in GetDetail. For properties with detail `mustvary`, the optional default parameter can be used to specify the default template to make a connection to. For an inline connection, pass `inline` for style.

proph ***SetDrawMode*** *m*

Set the Draw Mode for a property. The default Draw Mode is all. When a Collection has a Draw Mode of `children`, it will not be drawn, but its children will.

proph ***SetLabel*** *label*

Set the entry's label to the specified string.

proph ***SetMsgHandler*** *msg cmd*

Set the property's msghandler for msg to be cmd.

proph ***SetState*** *s*

Set the state of the property. Most often, this is used with Collections, which can either be open or `closed`. Properties without any children have a `barren` state.

proph ***SetUserData*** *key value*

Set the user data for key to value.

proph ***SetUserDataArray*** *data*

Set the entire array of user data. This can be useful when copying user data from one object to another.

proph ***SetValueProvider*** *p*

Set the ValueProvider for the property to p. Valid options for ValueProvider are discussed in the GetValueProvider method.

proph ***SetValueProviderRange*** *range*

Set the range of valid ValueProviders to the provided list, e.g. `{constant connection}`. You can use this to, for example, prevent a property from being marked external. Valid entries for the range are discussed in the GetValueProvider method.

Collection Objects

Collection objects are used to group Parameters and Attributes. Since they do not have a value, there is but a small number of methods specific to Collection objects:

collh ***GetCustomUIName***

collh ***GetCustomUIVendor***

collh ***GetCustomUIModes***

CustomUIs are defined on Collection properties. These methods access the CustomUI information for a collection.

collh ***NewCollection*** *type nm*

collh ***NewParameter*** *type nm*

collh ***NewRIBAttribute*** *type nm*

collh ***NewSlimAttribute*** *type nm*

collh ***NewTCLAttribute*** *type nm*

collh ***NewTORAttribute*** *type nm*

Like their counterpart methods defined for the Appearance class, these methods are used both for parsing a palette file, and for creating new properties in a dynamic template. When using these methods, any properties will be created as children of the collection.

collh ***SetCustomUI*** *vendor nm ?modes : ""?*

Set the customui information for the collection.

AtomicProperty Objects

*The AtomicProperty base class represents all properties which have a value. Because of this, the majority of the methods listed below relate to the property's value. Parameter and all of the *Attribute classes are all subclasses of AtomicProperty, and from a scripting point of view, are relatively similar.*

proph ***GetArrayIndex***

If the property is an element of an array, get its index. If the property is not an element of an array, this method will return -1.

proph ***GetDefaultValue***

Get the default value for this property.

proph ***GetExpression***

Get the TCL expression that is set for this property. Note that this may return an expression, even if the

property's ValueProvider is not set to `expression`.

proph ***GetValue***

Get the current value of the property.

proph ***GetMax***

proph ***GetMin***

proph ***GetPrecision***

proph ***GetRange***

Access the range information for this property. As you can see in a palette or template file, the range is stored as a pair consisting of `{min max}` or as a triple consisting of `{min max precision}`. ***GetRange*** *accesses this list directly.* ***GetMin*** *and* ***GetMax*** *are provided as convenience functions. The* ***GetPrecision*** *method accesses the third element of range, if present. If it is not present, this method returns the `DefaultSliderPrecision` preference (defined in slim.ini with a default of `0.01`).*

proph ***GetTCLContext***

Get the context in which a TCL expression will be evaluated. By default, TCL expressions are evaluated in the `mtor` context, allowing them to make use of information such as the object name, frame, etc. TCL expressions that wish to evaluate before rib generation and/or make use of the Slim scripting environment will be evaluated in the `slim` context.

proph ***GetUserRange***

Get the user range for the property, which is set when the user enters a value beyond the template-specified range (including precision).

proph ***RevertValue***

Revert the value for this property to its default.

proph ***SetArrayIndex*** *i*

Set the index for a property that is an element of an array. An index of -1 indicates that this property is not an element in an array.

proph ***SetDefaultValue*** *v*

Set the default value for this property to be v.

proph ***SetExpression*** *expr*

Set the TCL expression for this property to be the specified string. To use this effectively, you will want to make sure that the property's ValueProvider is set to `expression`.

proph ***SetRange*** *r*

Set range to r, where r is a list containing `{min max}` or `{min max precision}`.

proph ***SetTCLContext*** *c*

Set the context in which a TCL expression is evaluated.

proph ***SetUserRange*** *r*

Set the user-overridden range for a property. This is mainly used by the SlimParser.

proph ***SetValue*** *v ?dynamic : 0?*

Set the current value of the property. The dynamic flag is used to indicate that the value is being set interactively. This will, among other things, keep the value from making its way onto the Undo stack. A slider, for example, would have dynamic set to 1 when the user is dragging the slider back and forth and set to 0 when the user releases the slider.

proph ***ValidateTCLExpression*** *str*

Evaluate the expression specified by str. Results of the validation will be sent to the Message Log. If the evaluation generates an error, it will be returned. A successful evaluation will return an empty string.

proph ***ValidateTCLString***

Validate a property that has a TCL expression for its value. This is useful for properties that, for example, contain a `txmake` expression.

Parameter Objects

What sets the Parameter class apart from other subclasses is its existence inside of the shader. It is for this reason that the methods specific to Parameter are oriented toward its presencing in shading language.

parmh ***GetNameSL*** *uniqueify*

Get the name of the parameter as it exists in shading language. The uniqueify parameter specifies whether to guarantee the the parameter has a unique name by prepending it with the name of the appearance. This is necessary for non root-level Functions.

parmh ***GetValueSL***

Get the parameter's value in a format suitable for shading language. For example:

```
% $parmh GetType
color
% $parmh GetValue
{1 0 0}
% $parmh GetValueSL
color(1,0,0)
```

Attribute Objects

Attribute objects represent RIB attributes associated with your shader. This is the only method unique to the Attribute class:

attrh ***IsEnabled***

It is briefly mentioned in the listing for GetValueProvider that for an Attribute, a value provider of `variable` indicates that the Attribute is disabled. This is admittedly not very intuitive, so the ***IsEnabled*** *method is provided as a convenience.*

SlimAttribute, TCLAttribute, and TORAttribute Objects

Though internally these classes differ from the AtomicProperty base class, from a scripting perspective they are the same. There are no methods specific to these classes.

Class Index

This index lists all of the classes of objects with which you will interact. These classes all derive from the TreeNode class, as demonstrated by the Class Hierarchy diagram.

This index lists the common name (how we refer to the class in documentation), the internal name (how the class is used within Slim internals), and the scripting name (how the class is used within the Slim scripting environment). Base classes like TreeNode do not have a scripting representation.

Common Name	Internal Name	Scripting Name
TreeNode	::Slim::TreeNode	
PaletteEntry	::Slim::PaletteEntry	
Palette	::Slim::Palette	::Slim::AlterEgo::plt
Appearance	::Slim::Appearance	
Function	::Slim::Function	::Slim::AlterEgo::func
Template	::Slim::Template	::Slim::AlterEgo::tmplt
Instance	::Slim::Instance	::Slim::AlterEgo::inst
Property	::Slim::Property	::Slim::AlterEgo::prop
Collection	::Slim::Collection	::Slim::AlterEgo::coll
AtomicProperty	::Slim::AtomicProperty	
Parameter	::Slim::Parameter	::Slim::AlterEgo::parm
Attribute	::Slim::Attribute	::Slim::AlterEgo::attr
SlimAttribute	::Slim::SlimAttribute	::Slim::AlterEgo::slimattr
TCLAttribute	::Slim::TCLAttribute	::Slim::AlterEgo::tclattr
TORAttribute	::Slim::TORAttribute	::Slim::AlterEgo::torattr

Procedures

The following procedures are defined in the Slim scripting environment:

GetAppName

This procedure returns the name of the current application. (Hint: it's "slim".)

GetEnv *var ?default?*

Get the value of the specified environment variable. You can use the optional default argument to specify a value to return if the environment variable is not defined.

GetHostName

Get the name of the host machine that is running Slim.

GetPlatform

Get a name-value list of platform information, suitable for setting as an array, e.g.:

```
% array set platform [GetPlatform]
% set platform(os)
Linux
```

tcl veterans will recognize this as the ***tcl_platform*** *global.*

GetPref *var ?default?*

Get the value of the specified preference. You can use the optional default argument to specify a value to return if the preference is not set. Note that querying an unset preference will have the side effect of setting that preference to the specified default.

GetUserName

Get the name of the user that is running Slim.

GetVersion *variation:numeric*

Get the current version of the application. The optional variation parameter dictates how much version information to print. By default, the `numeric` *variation is used:*

numeric

The simplest form of the version string, e.g. `6.5`*.*

RAT

Include the build suffix e.g. `6.5b3`*.*

short

Return the RAT version plus build date.

long

The most verbose version information, including build date and copyright information. This is the string returned when running `slim -version`*.*

::RAT::GetUniqueID

Generate a new unique identifier.

::RAT::LogMsg *level msg*

Send the message msg to the Slim Message Log. Valid values for level are: `COPIOUS`*,* `INFO`*,* `NOTICE`*,* `WARNING`*,* `ERROR`*,* `SEVERE`*, in order of rising priority.*

::RAT::PickFile *?arg arg ...?*

Invoke the File Picker. Arguments to this command specify how to configure the File Picker and the command to execute when the user has selected a file:

-cmd *command*

The command to execute when the user has a selected a file. The pattern `%v` *is substituted with the selected pathname.*

-filetypes *types*

Types of files to show in the File Picker. Sample types include `All`*,* `Image`*,* `Texture`*, etc. A complete list of definitions of types and how they map to file extensions can be found in* `RAT.ini`

-initialfile *path*

The initial path to show in the File Picker.

-mode *mode*

The File Picker's mode of operation. Possible modes include openone (the default), `openmulti`, `opendir`, and save.

-title *title*

Title for the file picker window. By default, this is "Open File..." when in open mode; "Save File..." when in save mode.

SetPref *var value*

Set the specified preference to the specifed value.

These procedures are defined via the Custom Commands that ship with Slim. You may find them useful to use in your own scripts.

::Slim::ResourceReport *?arg arg ...?*

Enumerate external resource usage and point out unresolved resources. The report is printed to the Slim Message Log. By default the report is verbose. The only valid arg is ***-short****, which trims the report down to only workspace information and warnings about unresolved resources. Source can be found in the file `customcmd.slim`.*

::Slim::SearchReplace *pattern replace ?arg arg ...?*

Searches for pattern in string entry fields of appearances and replaces with replace. Information about each match replacement that occurs is printed to Slim's Message Log. Note that expression fields (ie. fields with the orange tcl button to their right) are ***not*** *included in this search/replace. You may also provide one or more of the following optional arguments:*

-scope *objh*

Restrict searches to the fields in the specified object handle, which may point to an appearance or to a palette. The default scope of `all` applies the search to all appearances in all palettes.

-filter *flt*

Restrict searches using the specified filter:

all

Apply to all string entry fields (the default).

appnames

Only apply to names of appearances.

masters

Only apply to the master fields of appearances.

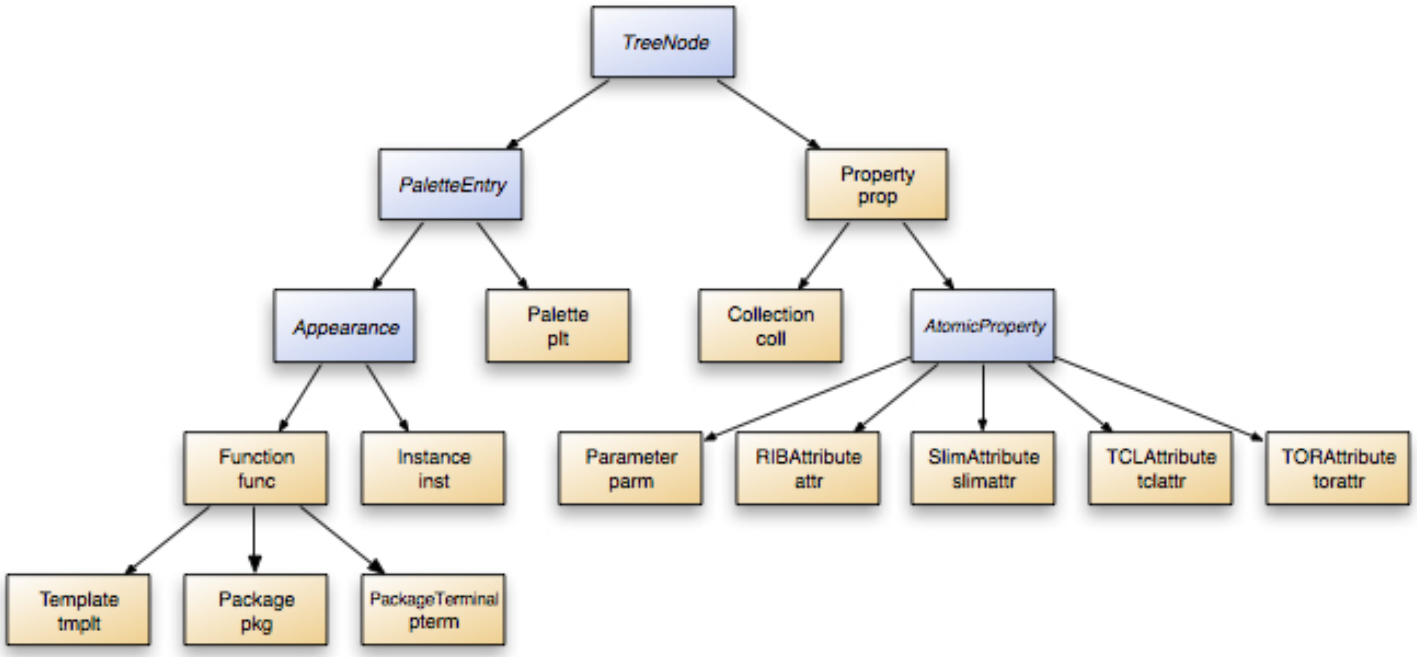
maps

Only apply to properties that contain a file reference. This does not include entries for an appearance's name or master.

nomaps

Apply to everything ***except*** *properties that contain a file reference.*

Scripting Class Hierarchy



Slim Templates

Writing Slim Templates

Introduction

Slim uses *templates* to define the functions that are combined and manipulated in order to create custom shaders. Templates have access to all of the power of RenderMan Shading Language, but can present high-level, modular, reusable units that are accessible to programmers and artists alike. Everything from simple patterns and filters to complicated illumination models can be represented using Slim templates.

Templates are only used for custom shader generation. If you have an existing compiled shader and you'd like more control over its interface in Slim, you can define its interface using a companion .slim file to provide parameter descriptions and ranges.

Slim ships with a large number of templates for creating custom shaders, but should you find something missing, you can develop your own templates and add them to the system. It is even possible to take the entire standard set of Slim templates and replace them with a set of your own.

Templates are stored in .slim files. The list of templates loaded into Slim and the specifics of how they are presented in Create Appearance menus are defined inside the initialization files that are read at startup. Templates consist of Tcl commands describing an appearance's user interface and a block of RenderMan Shading Language (RSL) which describes its contribution to the shader.

We won't discuss specifics of RSL or Tcl here. Suffice it to say that working knowledge of RSL is required to create useful Slim templates. As you will see, the amount of Tcl knowledge required depends on the complexity of the template being created.

Writing a Simple Template

To demonstrate how to write a template, we're going to create a simple template for performing a `smoothstep` operation. For the purposes of illustration, we're actually going to go *inside-out*. We'll start with the core of a template, the RSL code, and introduce each of the layers that wrap this code into a template, and then into a `.slim` file.

The RSL Function

First, here is the RSL portion of our `smoothstep` template:

```
void
pxslSmoothstep(float input, minVal, maxVal, scl;
               output float result;)
{
    /* calculate the standard smoothstep */
    result = scl * smoothstep(minVal, maxVal, input);
}
```

Here are some key things to notice:

- This is standard RSL syntax, comments and all. There are no special characters or strings, just a definition of a function.
- The function does not return the result of the smoothstep, but instead sets an output variable. This will always be the case, especially when setting multiple result variables.
- The name of the RSL function contains a special prefix: `pxsl`. We'll return to how this prefix is used later on.

Parameter Definitions

Next, we'll list the parameters of this function and provide more information about them for the sake of user interface. These parameters are defined using *Tcl* syntax:

```
parameter float Input {
    description {
        Pattern to feed through smoothstep
    }
    detail mustvary
    default s
}
parameter float MinVal {
    label "Min Value"
    description {
        Threshold below which the function will return 0
    }
    default .2
    range {0 1 .001}
}
parameter float MaxVal {
    label "Max Value"
    description {
        Threshold above which the function will return 1
    }

    default .8
    range {0 1 .001}
}
parameter float Scale {
```

```
    description {
        Scale to apply to the result
    }
    detail varying
    default 1
}
parameter float result {
    access output
    display hidden
}
```

Before analyzing the specifics of each parameter's declaration, some comments about the parameter declaration:

- These parameters are used when calling your RSL function. You must declare a parameter for every parameter of your function, and the order of these parameters must correspond to the order of parameters in your RSL function.
- The names of these parameters must resemble variable names (i.e. no spaces, punctuation, etc.). To define a more user-friendly string, you can use the `label` keyword. Note that in this case, it is **not** necessary for the names declared here to match the names of parameters in your RSL function (though it is certainly recommended).

The parameter statements are actually a series of calls to the proc called `parameter`, which takes the form:

parameter *type name body*

`type` and `name` ("float" and "Input" in the first parameter) are the most important arguments. What follows is a list of key/value pairs with specific details about the parameters. Because this is Tcl, it is common to list these details on consecutive lines (as the {} syntax allows one to do). The key-value pairs can be listed in any order, though as with anything else, staying consistent is appreciated by your fellow developers.

Let's examine some of the keywords listed above:

access *acc*

Access refers to whether the parameter is an input or output parameter of the RSL function. Input is the default, so this is only necessary when declaring an output parameter. Note that even though the output parameter result is not visible by the user, it still needs to be declared. Neglecting to include output parameter(s) or to properly label their access can result in problems with insufficient variables in the resulting shader.

default *def*

The default value for the parameter. Remember that this is Tcl syntax, so when providing default values for a color, use a list, e.g. `default {1 1 1}`.

description *desc*

This information will be used to create the info "balloon" that the user sees when pressing the **i** button. You can use curly braces to make your description span severl lines. Don't worry about whitespace; it will be compressed.

detail *d*

By default parameters are uniform, which means their value cannot be provided by another appearance. You can use **detail** to specify a parameter as **varying**, and thus connectable. In the case above, we've specified that Input is detail **mustvary**, meaning the value can only be provided by another appearance.

display *d*

The visibility of the parameter. You can use `display hidden` to make a parameter hidden, which is usually necessary for output parameters.

range *r*

The range specifies the minimum and maximum values of the parameter, and optionally, the precision. The default range is {0 1 .01}. You can also specify a parameter as unbound, e.g. {0 unbound .0001}.

Template Declaration

The appearance parameters and shading language code are declared inside of a template. Our template declaration will look something like the following. Note that for the sake of brevity in this demonstration, we have skipped the parameter definitions and use a placeholder.

```
template float Smoothstep {
    description {
        0 if input < minVal; 1 if input >= maxVal; otherwise, a
        smooth Hermite interpolation between 0 and 1.
    }

    template parameters

    RSLSource StaticFunction {
        void
        pxslSmoothstep(float input, minVal, maxVal, scl;
                       output float result;)
        {
            /* calculate the standard smoothstep */
            result = scl * smoothstep(minVal, maxVal, input);
        }
    }
}
```

template is a proc which takes the form:

`template` *`type name body`*

We've named this template "Smoothstep." The name used here must match the suffix of the function declared within the RSL source. We've also added a description. This will be visible to users inside of the Appearance Editor.

Our shading language is enclosed within a call to **RSLSource**, and we've declared our code to be a StaticFunction. This is the simplest type of RSLSource, as its contents are simply a block of RSL code.

Template Wrapper

The outer-most piece to your template file wraps your templates up by author and vendor information. You can use this to distinguish between the factory templates that ship with Slim and the templates you develop. The wrapper looks something like the following:

```
slim 1 extensions author {
    extensions vendor prefix {

        template(s)
```

```
    }
}
```

The fields are defined as follows:

author
: The author of the template. This is basically for documentation purposes, and is not used at all by Slim.

vendor
: The vendor is used to define a namespace for your template in Slim, the full description of which is *vendor*,*template-name*#*template-version*.

prefix
: The prefix acts as a namespace for the shading language functions that are generated by your template, the names of which are the simple concatenation of the prefix and the template name.

Below is an example of the standard template wrapper found in factory templates:

```
slim 1 extensions pixardb {
    extensions pixar pxsl {


    }
}
```

Were we to define our Smoothstep template, within this wrapper, its full identifier would be:

```
pixar,Smoothstep#0
```

whereas the shading language function generated by this template would be:

```
pxslSmoothstep
```

Note that this is the name that we used in our RSLSource block above.

The Complete Template File

Below is the complete template file. You can also find this file here.

```
##
##  Smoothstep.slim
##  Implements the shading language smoothstep function
##

slim 1 extensions pixartt {
    extensions pixar pxsl {
        template float Smoothstep {
            description {
                0 if input < minVal; 1 if input >= maxVal; otherwise, a
                smooth Hermite interpolattion between 0 and 1.
            }
            parameter float Input {
                detail mustvary
```

```
                description {
                    Pattern to feed through smoothstep
                }
                default s
            }
            parameter float MinVal {
                label "Min Value"
                description {
                    Threshold below which the function will return 0
                }
                default .2
                range {0 1 .001}
                subtype slider
            }
            parameter float MaxVal {
                label "Max Value"
                description {
                    Threshold above which the function will return 1
                }
                default .8
                range {0 1 .001}
                subtype slider
            }
            parameter float Scale {
                description {
                    Scale to apply to the result
                }
                detail varying
                default 1
            }
            parameter float result {
                access output
                display hidden
            }
            RSLSource StaticFunction {
                void
                pxslSmoothstep(float input, minVal, maxVal, scl;
                               output float result;)
                {
                    /* calculate the standard smoothstep */
                    result = scl * smoothstep(minVal, maxVal, input);
                }
            }
        }
    }
}
```

Using Your Template

Loading and Testing

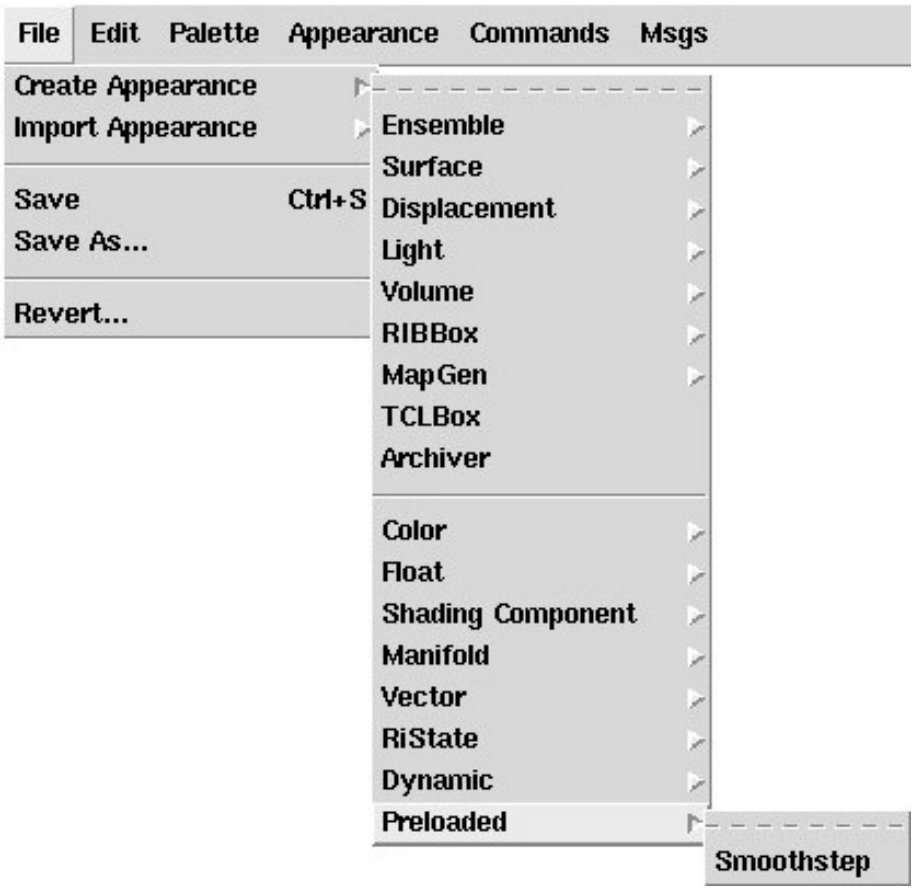
The quickest way to load your template into Slim for the sake of debugging is to use Slim's scripting environment. Assuming you've saved `smoothstep.slim` in the current working directory, you can read it into

slim with the following command:

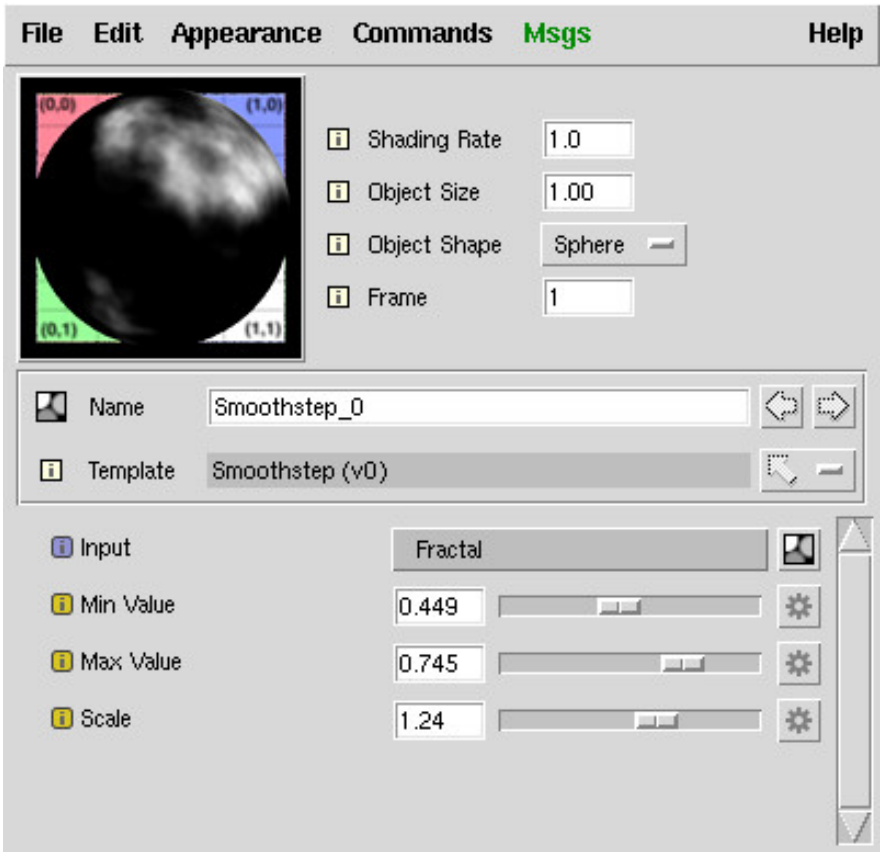
```
slim ReadSlimFile smoothstep.slim
```

This reads the commands in the specified slim file, which in this case houses your template. Any errors encountered when parsing the .slim file will be directed to the Message Log.

You can create an appearance based on your template by finding it in the Preloaded menu. This is where Slim places any templates that have been loaded, but not *registered* (more on that later). Note that this menu is only present with Expert Menus turned on.



Create an appearance using your Smoothstep template, then bring it up in the Appearance Editor. It should look something like the following:



In this example, we've adjusted the sliders to make sure they function properly, and connected a Fractal as input to test how our template can be used to filter a pattern. As you can see from the preview swatch, the template works fine.

If you aren't as lucky, you will find errors from the shader compiler in the Message Log. Change your template to fix the errors, and use the **File : Reload Template** command to reload your template and try again.

You can also use the Reload Template command to update your appearance with any changes that you make to the template. Reloading a template will update the settings for parameters in an appearance, though parameter values that have been set by the user will be preserved (you can update these too by reverting parameters to their default values).

Loading Your Template at Startup

Obviously, entering the Slim console to read your Slim file is not something you or your users will want to do when they want to use your template. Instead, you'll want to tell Slim to load your template at startup and to place it in its Create Appearance menus.

You'll do this by editing one of Slim's initialization files. For the sake of testing, the easiest ini file to work with is your personal slim.ini file, located at `$HOME/.pixarPrefs/RenderMan_Studio/<Version>/slim.ini`. If this file is not already present, go ahead and create it.

You'll first want to tell Slim the directory, or search path, that contains your template. If, for example, you've stored your smoothstep.slim file in a subdirectory off of your home directory called `slimdev`, you will want to add this directory to the template search path by adding the following to your slim.ini:

```
set tSearchPath [file join [GetEnv HOME] slimdev]

set paths [GetPref TemplateSearchPaths]
lappend paths $tSearchPath
SetPref TemplateSearchPaths $paths
```

Next, you'll want to tell Slim to load your template. The best way to load your template is to take advantage of Slim's lazy template loading mechanism. This system allows templates to be loaded only as they are needed, which means Slim is quick to start regardless of the number of templates that you add. With lazy template loading, you *register* your template by giving Slim enough information to place it in its Create Appearance menus and to find the actual template file when its needed.

The factory slim.ini file ships with a procedure to help you with this step, as demonstrated below:

```
::Slim::RegisterLazyTemplates {
    smoothstep.slim {
        { pixar,Smoothstep#0  float  Smoothstep  /Float/Utility }
    }
}
```

RegisterLazyTemplates expects a list of slim files containing templates and details about the templates inside. The details consist of four vital pieces of information:

- The template identifier (pixar,Smoothstep#0)
- The return type of the template (float)
- The way the template should be presented in the Slim UI (Smoothstep)
- Where the template should be placed in the Create Appearance menus (/Float/Utility)

With lines like the above in your .ini file, your template should be visible in Slim's Create Appearance menus. Assuming you've turned on Expert Menus, you should see it in **Create Appearance : Float : Utility**.

The lazy-loading is the best method for loading a Slim template. If for some reason, however, it is necessary to load your template immediately at startup, you can do so using the `LoadExtension` initialization command directly. This is common for files containing customui information, tcl macros, or templates not normally seen by the user. For example:

```
LoadExtension slim [file join [GetEnv HOME] slimdev conversion.slim] slim
```

Note that in this case, we're required to fully specify the path to the .slim file (which we're doing via the `file join` command).

Installing Your Template

Once a template is polished and ready for general consumption, you'll likely want to install it someplace other than your home directory.

Though it might be tempting, we recommend that you **not** place it inside of the slim installation location or register it using the factory slim.ini. Doing so will make the process of installing new versions of Slim much more complicated, and you risk accidentally deleting your template.

Instead we recommend a common directory for all of the extensions you add to Slim. You can tell Slim (and other RenderMan Studio applications) about this directory using the $RMS_SCRIPT_PATHS environment variable. For example, you might store all of your extensions in a directory like:

```
/share/pixar/extensions
```

To tell Slim (and MTOR, Alfred, etc) about this directory, you would:

```
% setenv RMS_SCRIPT_PATHS /share/pixar/extensions
```

Slim will look for a file called slim.ini file in this directory. Use this file to list any extensions (scripts, templates, etc.) you wish to be loaded by Slim. You might want to place your templates in a subdirectory, like:

```
/share/pixar/extensions/slim/templates
```

Transfer your template to the subdirectory, and then copy the code from your personal .ini file to the shared .ini file. Note that the only change required is the declaration of the template search path:

```
set tSearchPath /share/pixar/extensions/slim/templates

set paths [GetPref TemplateSearchPaths]
lappend paths $tSearchPath
SetPref TemplateSearchPaths $paths

::Slim::RegisterLazyTemplates {
    smoothstep.slim {
        { pixar,Smoothstep#0  float  Smoothstep  /Float/Utility }
    }
}
```

With the template and .ini files in place. Any user that sets RMS_SCRIPT_PATHS above will load the template you've installed. More information about initialization files is available here.

Dynamic Templates

For most applications, declaring the RSL for a template as a *static* block will suffice. As you have seen, it is relatively straightforward to do. There are situations, however, where you will demand more power and flexibility. For these situations, you can take a more active role in code generation with the various flavors of *dynamic* templates.

Dynamic Functions

The Smoothstep template above is actually quite specific. It might make a more useful template if the user were able to select what operation were performed on its three input arguments via a property. Here is the declaration for a *slimattribute* that would provide the user with a choice of functions:

```
slimattribute string Operation {
    description "Operation to perform on input"
    subtype selector
    range {
        Clamp clamp
        Remap remap
        Smoothstep smoothstep
    }
    default smoothstep
}
```

Before we discuss how to use the new slimattribute, here are some definitions for what you may not recognize:

slimattribute

A slimattribute is different from a parameter. Unlike a parameter it cannot be set to

internal or external, and it cannot be connected to another function. It isn't used to communicate with MTOR (as a torattribute is) or influence RIB generation (as a rib attribute is). It is only useful from the context of the appearance. This makes it useful for controlling code generation.

subtype selector

Without this declaration, the user would be presented with a text entry field. By declaring a subtype, we give Slim more information about how to present the property to the user. In this case, we've specified **selector**, which means the user will select from a predefined list of options.

range ...

We've used range before to specify the minimum and maximum values for sliders. Here we're using it to specify the options to present to the user. The range consist of a list of pairs of labels (which are presented to the user) and values (which are used internally).

To use this slimattribute, we will have to change how we declare the RSL portion of our template. Rather than declaring it as a static block, we will use Tcl to output each line of our function. In doing so, our **StaticFunction** will become a **DynamicFunction**

DynamicFunction templates are specified by declaring two local procedures that will be executed when Slim generates your shader. The basic structure looks like:

```
RSLSource DynamicFunction {
    proc primvars {} {
        ...
    }
    proc function {} {
        ...
    }
}
```

The first procedure, primvars, will be executed within the parameter block of the shader. The second procedure, function, will be executed when declaring the function in the shader. Inside of each statements you will write a series of output statements containing your shading language. Were we to rewrite our Smoothstep template using a DynamicFunction, the result might look something like:

```
RSLSource DynamicFunction {
    proc primvars {} {
    }
    proc function {} {
        generateBody {
            output "/* calculate the standard smoothstep */"
            output "result = Scale * smoothstep(MinVal, MaxVal, Input);"
        }
    }
}
```

The **generateBody** procedure declares the function for you, so all you have to do is compose the body of the function. Note that when using generateBody, your parameter names must **exactly match**.

So far, this isn't very interesting, because we're not doing anything very dynamic with our DynamicFunction. So let's incorporate the Operation slimattribute:

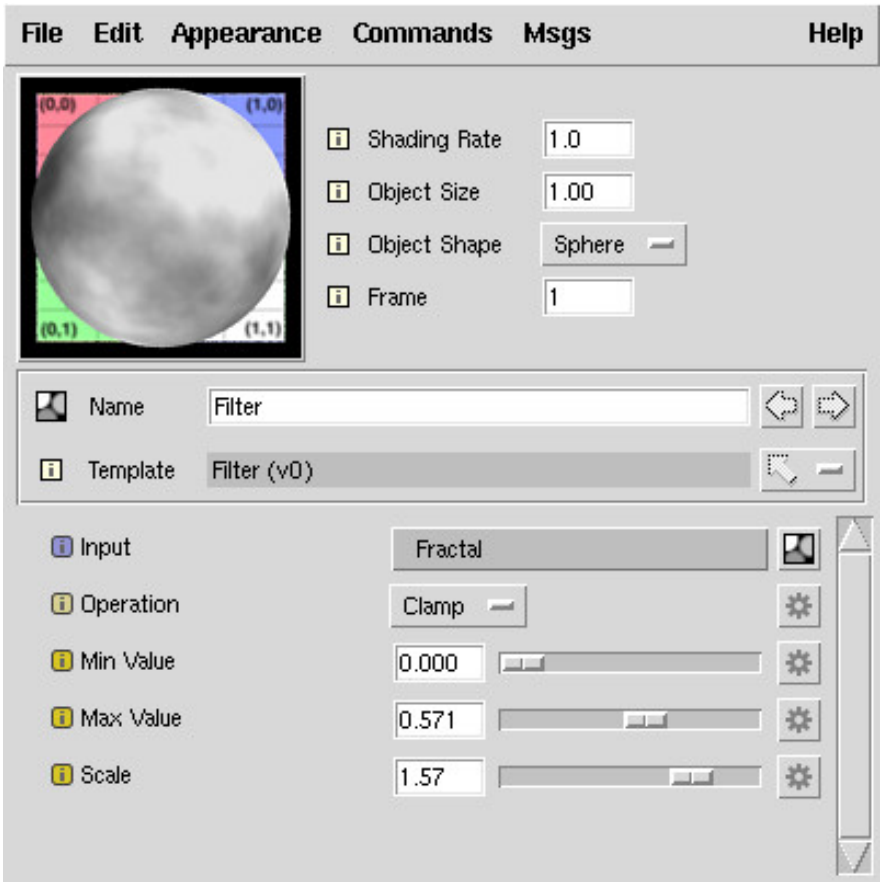
```
RSLSource DynamicFunction {
```

```
    proc primvars {} {
    }
    proc function {} {
        generateBody {
            switch [getval Operation] {
                clamp {
                    output "/* clamp */"
                    output "result = Scale * clamp(Input, MinVal, MaxVal);"
                }
                remap {
                    output "/* remap */"
                    output "result = Scale * (MinVal + Input * (MaxVal-MinVal));"
                }
                smoothstep {
                    output "/* smoothstep */"
                    output "result = Scale * smoothstep(MinVal, MaxVal, Input);"
                }
                default {
                    output "/* unrecognized Operation: [getval Operation] */"
                    output "result = 0;"
                }
            }
        }
    }
}
```

Here we use the **getval** procedure to query the value of our slimattribute. Then, we output the proper bit of RSL based on the value of Operation. This is why we've used a slimattribute rather than a parameter. By using a slimattribute, we guarantee that the value is constant, so it's suitable for directing our code generation.

By using a DynamicFunction, we've made our Smoothstep template a lot more useful. And because we're making decisions about which function to execute at code-generation time (versus render time), the resulting shader remains efficient.

Here is the complete file containing the new template, which we've christened with the more general name of "Filter." If you'd like, you can copy the file, read it into Slim, and try it out. Appearances created using this template should appear something like the following:



To see the effect of using a DynamicFunction, change the value of Operation, and view the resulting shader using **File : View SL Source** (be sure Expert Menus are turned on). You'll see that everytime you change the Operation, the body of your function will change.

Accessing Primitive Variables

The example above had an empty delcaration of the primvars procedure. This procedure is useful when you want to declare additional parameters of your shader that are set by the geometry of your scene. Here is a template that allows you to declare an additional shader parameter from within a function:

```
template float PrimVarFloat {
    label PrimVar
    slimattribute string name {
        label "Primitive Variable"
        description {
            Enter the name of your float primitive variable here.
            Note that this cannot be an expression.
        }
        default {}
    }
    parameter float Multiplier {
        description {
            A multiplier for the underlying vertex variable.
        }
        default 1
        detail varying
    }
    parameter float result {
        detail varying
        access output
        display hidden
```

```
    }
    RSLSource DynamicFunction {
        # in primvars, we declare our primitive
        # variable based on the name entered by the user.
        proc primvars {} {
            set nm [getval name]
            if {$nm != ""} {
                declare output varying float $nm 0
            }
        }
        # in function, we generate the code.
        proc function {} {
            generateBody {
                set varnm [getval name]
                if {$varnm == {}} {
                    output "/* no name entered for primitive variable */"
                    output "result = Multiplier;"
                } else {
                    # be sure to extern the variable
                    output "extern float $varnm;"
                    output "result = Multiplier*$varnm;"
                }
            }
        }
    }
}
```

Note that we've declared our primvar using the **declare** keyword.

Note that it is possible to access a *defined* primitive variable from a StaticFunction by declaring a parameter to have provider **primitive**. This template provides access to the AgeNormPP variable, a variable automatically attached to Maya particle systems by MTOR:

```
template float AgeNormPP {
    description {
        Access to the vertex variable associated
        with Maya particle systems. This value ranges from
        0 to 1.
    }
    parameter float multiplier {
        description {
            A multiplier for the underlying vertex
            variable.
        }
        default 1
    }
    parameter float ageNormPP {
        display hidden
        provider primitive
        detail varying
        default 0
    }
    parameter float result {
        access output
        display hidden
```

```
    }
    RSLSource StaticFunction {
        void
        pxslAgeNormPP(float mult; output float result;)
        {
            extern float ageNormPP;
            result = mult * ageNormPP;
        }
    }
}
```

Dynamic Shaders

The example templates above both produce RSL functions that are inserted into the shader. If you've ever written an RSL shader, though, you know that at some point, the shader will need to set **Ci** (or **Cl**). DynamicShader templates provide the main body of the shader that triggers all of the function calls and sets what variables are necessary for the shader to take effect. It is for this reason that DynamicShaders are often declared as RSLMain.

As an example of an extremely simple DynamicShader template, here is the source for an Ambient light template:

```
template light Ambient {
    lighttype ambient
    parameter color AmbientColor {
        description {
            The color of your the light. You
            can connect a pattern generator here or
            simply use a constant color.  Two common choices
            are an image map or a spline which varies with
            distance.
        }
        detail varying
        default {1 1 1}
    }
    RSLSource DynamicShader {
        output "extern point Ps;"
        output "L = vector(0,0,0);"
        generate
        output "Cl = [getvar AmbientColor];"
    }
}
```

Some things to notice about this template:

- The type of template is **light**. DynamicShaders can only be declared for the four canonical types of RenderMan shaders: surface, displacement, light, and volume.
- A **lighttype** has been declared. This is necessary for light templates. Valid lighttypes are ambient, point, distant, spot, and environment.
- There is no generateBody call. This code will be in the main body of the shader, not in a function, so a function wrapper is not necessary.
- There is a **generate** call. This triggers the execution of functions in the function graph.
- The **getvar** procedure is used. Since this code will not be inside of a function call, we don't know what the name of the variable containing the value of AmbientColor will be. Most likely, it will be a temporary

variable used for the result of the function connected to AmbientColor. You must always use **getvar** to refer to your parameters in a DynamicShader.

Generative Functions

View the source of a simple shader generated by Slim and you will see that there is a relatively simple flow of data. The flow is actually a graph, not unlike what is visible in Slim's graph editor. Leaf appearances like manifolds are the first functions to be executed. The results of these functions are used by pattern generators. These results, in turn, are used by illumination functions, which eventually flow to the root function of the graph.

Most of the time, this sort of data flow works very well. It means that no function is ever needlessly executed multiple times. If multiple appearances connect to the same function, that function will only be executed once. There are times, however, when you may find you need more control over the execution of functions. The following RSLSource from a template that switches between two inputs provides us with an example situation:

```
RSLSource StaticFunction {
    void
    pxslSwitch( float input0, input1, which;
                output float result; )
    {
        if (which > 0.5)
            result = input1;
        else
            result = input0;
    }
}
```

This template takes two input parameters, and based on the **which** parameter, selects one of them. This might allow a user to choose between two different patterns that are fed into the function.

When considering the way that data flows in a slim shader, however, you may come to this realization: you will often be calculating something that will never be used. If input0 is connected to a function that ray-traces, but which is set to 1, all of that computation has been wasted.

In these situations, you would like more control over the generation of the shader. Rather than relying on the input functions being computed before our function is called, we wish to choose which function is called. It is in these situations that we can use a **GenerativeFunction**.

GenerativeFunctions are like DynamicFunctions, except inside them you can use the generate command on a specific parameter. This, for example is a version of the Switch template written using a GenerativeFunction:

```
RSLSource GenerativeFunction {
    proc primvars {} {}
    proc function {} {
        generateBody {
            output "if (Which > 0.5)"
            output "{"
            indent
            set var1 [generate Input1]
            output "result = $var1;"
            exdent
            output "}"
            output "else"
            output "{"
```

```
            indent
            set var0 [generate Input0]
            output "result = $var0;"
            exdent
            output "}"
        }
    }
}
```

When the shader is generated, the generate commands will expand into calls to the functions connected to Input0 and Input1. To see how this works, you can test the simple Switch template in switch.slim. Connect functions to each input and then view the resulting shader. You'll see that the function calls are actually quite complicated. Slim is keeping track of which functions are called so that if the result of any function is used by more than one node, the function doesn't get executed twice.

To see a more complicated example of a GenerativeFunction in action, view the source for the LayerUltimo (aka "Layer (fast)") template.

Code Generation Commands

connected *paramname*

useful to determine whether a parameter obtains its results from another function. returns 1 if *paramname* is connected, 0 otherwise.

declare *access detail type name value*

allows you to declare primitive variables.

only valid within the primvars procedure of Dynamic and Generative functions

define *expression*

inserts a #define in the shader with the given *expression*

exdent

decreases the indentation level for output commands (it's the opposite of **indent**).

generate *args*

invokes Slim's code generator which causes the creation of temporary variables and the invocation of all connected functions. When specified with no arguments, the entire call graph will be emitted. When specified with a list of parameter names, only those portions of the call graph associated with the named parameters will be emmited. Note that *generate* supports the use of wild cards (glob expressions) as filters in the parameter list, with the following three exceptions:

- -- all the subsequent arguments are treated literally and exact match is forced.
- - is used to specify arguments to explicitly exclude.
- ****:* (with the optional leading "-") the argument is interpreted as a regular expression and regexp matching is used instead of wild cards (glob).

only valid within a GenerativeFunction or DynamicShader

generateBody *body*

used within the function procedure of a DynamicFunction or GenerativeFunction, generates

the function parameter list. Enclose your code generation logic within the body.

getFunctionName

used within the function procedure of a DynamicFunction, returns a unique name for a DynamicFunction. only necessary if declaring the function parameter list manually.

getproperties *args*

returns a list of property **names** that match the criteria set forth in the option list. Most commonly used as

`getproperties -name` *`pattern`*

Other criteria correspond to those used in the GetProperties method.

getpropertyinfo *paramname args*

returns information about *paramname* as specified by additional arguments:

default

The default value of *paramname*

detail

The detail of *paramname*

label

The label of *paramname*

provider

The value provider for *paramname*. Return value is one of: constant (internal), variable (external), expression, or connection.

type

The type of *paramname*

userdata *key*

User data for *paramname* specified by *key*

getvar *paramname*

returns and RSL representation of *paramname*. Depending on the connectivity of the parameter, this may result in a reference to a temporary variable, a local constant or the name of a formal shader parameter.

getval *paramname*

returns the current value of the *paramname*. Note: this only is accurate for properties with constant values. Parameters with values provided by an external value or by a connection may have a different value at run time than they do at code generation time.

indent

increases the indentation level for output commands.

include *file*

adds *file* for inclusion (via #include) at compile time.

output *str*

places *str* into the output stream of the code generator.

safeeval *expression*

Code generation executes in a limited tcl interpreter that is separate from the safe interpreter used by the Slim scripting environment. You can use **safeeval** to execute a command in the main safe interpreter. Use %c in your expression and it will be replaced by the current function, e.g.

```
set functionName [safeeval %c GetLabel]
```

Prev | Next

Templates: Advanced Topics

Visualizers

When viewing the source of the Slim-generated shader, you may have wondered how the return value of a function is turned into the final color, **Ci**. Visualizers are simply a special DynamicShader template. Here, for example, is a visualizer template for the float type:

```
template visualizer float {
    parameter float floatFunc {
        detail mustvary
        default 1
    }
    RSLSource DynamicShader {
        generate
        output "Ci = [getvar floatFunc];"
        output "Oi = color(1,1,1);"
    }
}
```

As you can see, other than the specialized declaration of the template name and type (visualizer is the type of template; float is the name), this is just another template. Slim automatically connects the floatFunc parameter to the result of the appearance being visualized, and the result is a usable shader.

Collections

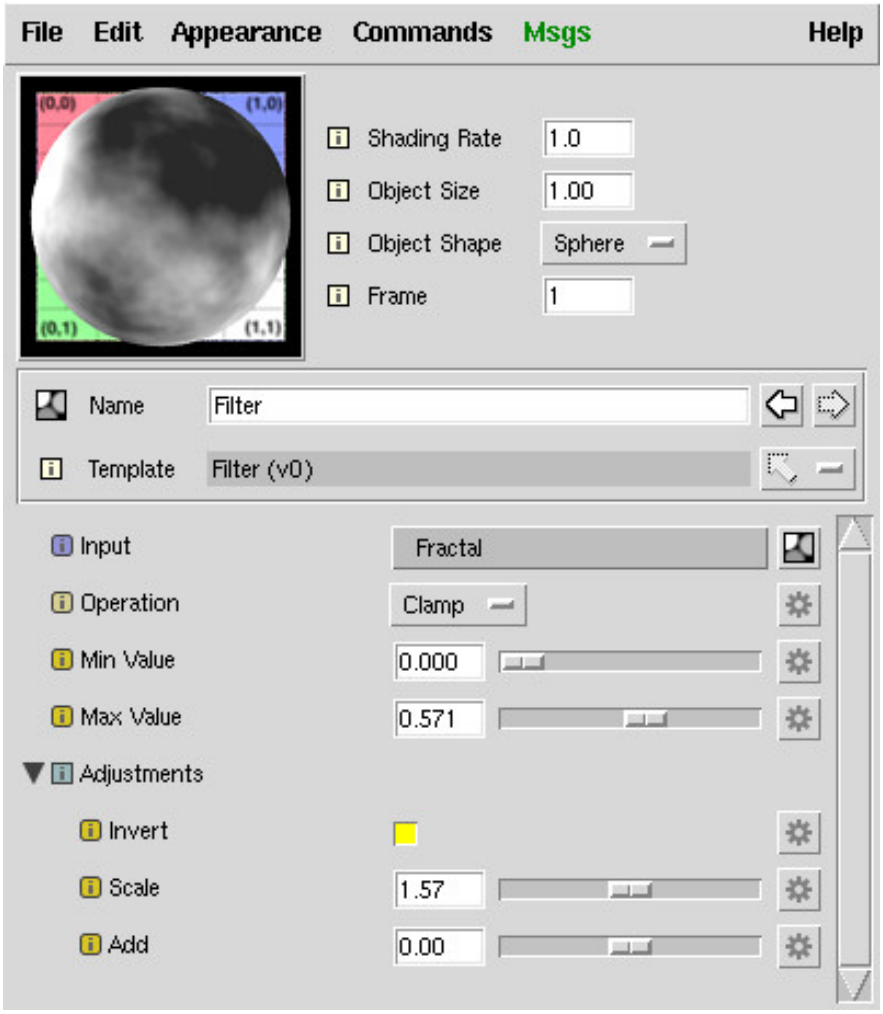
Collections, as their name would suggest, collect a group of properties. Collections are just another class of property, so collections can group other collections. There are several ways to use collections.

The most common use of collections is to group a set of controls in a template. This is achieved by declaring a collection of type `void`. This declaration of parameters in an upgraded Filter template groups several similar controls together:

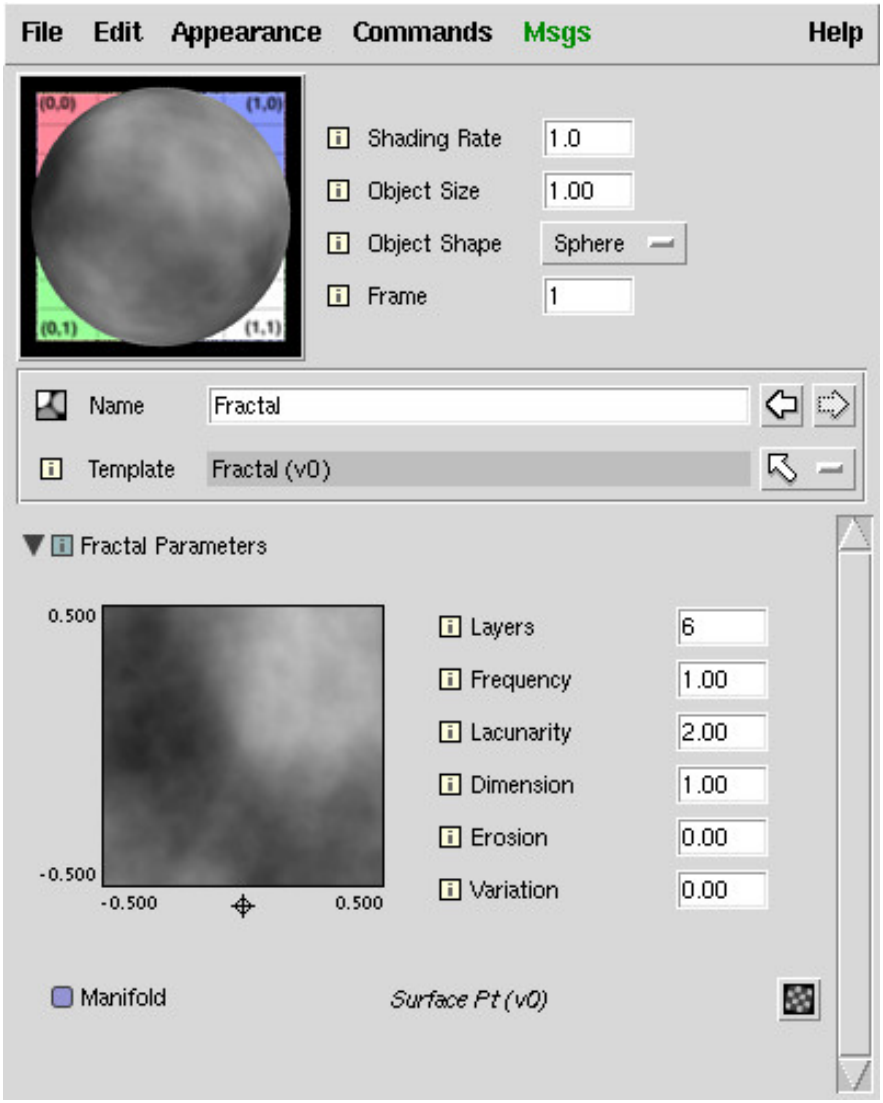
```
collection void Adjustments {
    state open
    description {
        Post-op adjustments
```

```
    }
    parameter float Invert {
        description {
            Invert result
        }
        subtype switch
        default 0
    }
    parameter float Scale {
        description {
            Scale to apply to the result
        }
        detail varying
        default 1
    }
    parameter float Add {
        description {
            Additive adjustment to apply to the result
        }
        detail varying
        default 0
    }
}
```

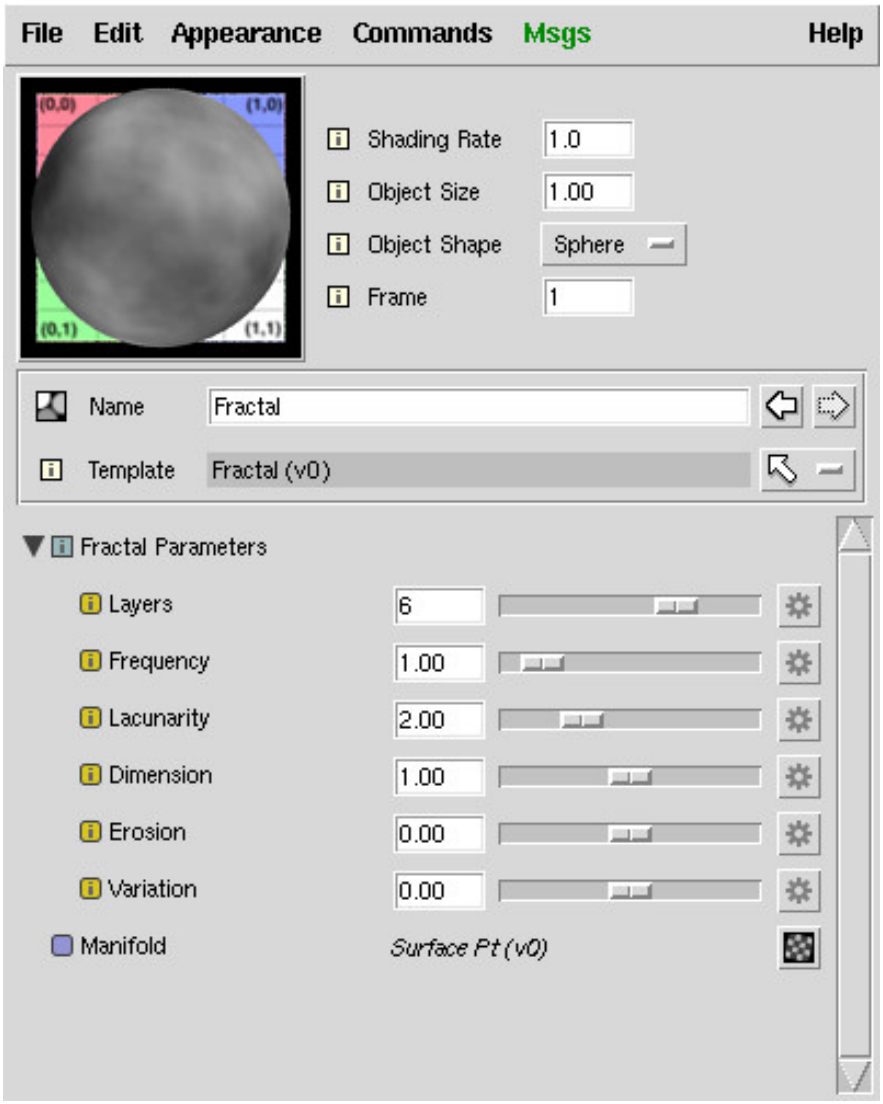
With these additional parameters grouped in a collection, our Filter template now produces the following appearance:



Collections are also used to group a collection of properties and present them using an alternative, custom user interface (or **customui**). The float Fractal templates use a customui to display a realtime preview of the fractal pattern. Here is how a Fractal appearance is normally displayed in the Appearance Editor:



Inside the Fractal Parameters collection, though, are just normal parameters. If we close the collection and reopen it while holding the Control key, the collection will display its parameters in the standard way:



Typed Collections

Collections can also be used to represent a *compound type*. You may have noticed that illumination templates are of type **shadingmodel**. Shadingmodel templates return information about both color and opacity. But there is no shadingmodel type in RSL, though, so how does this work? As you can see in this snippet from a shadingmodel template, a shadingmodel is just a collection containing two color parameters. Here is a declaration of the output parameters for a shadingmodel template:

```
collection shadingmodel result {
    access output
    display hidden
    parameter color ResultColor {
        access output
    }
    parameter color ResultOpacity {
        access output
    }
}
```

And here is a declaration of parameters for a template that takes a shadingmodel as input:

```
collection shadingmodel Layer1 {
    description {
```

```
        Connect this to a shadingmodel
    }
    detail mustvary
    state locked
    parameter color CLayer1 {
        detail mustvary
        default {1 .25 0}
    }
    parameter color OLayer1 {
        detail mustvary
        default {1 1 1}
    }
}
```

The **detail mustvary** line forces a connection. The **state locked** means that this collection should be represented as a single entity, and users shouldn't be able to open it and set CLayer1 and OLayer1 directly.

Slim will handle any compound type you wish to define. To use a compound type, you must:

1. Declare the template with your compound type
2. Declare output parameters grouped in a collection of the compound type
3. Declare input parameters in client templates in a collection of the compound type
4. Create a visualizer for your type. This allows users to preview the appearance.

That's it. There is no central definition for the contents of a compound type. When making connections, Slim just assumes that the number of input parameters matches the number of output parameters. If they don't match, an error will be generated.

Type Conversion

Slim features *automatic type conversion*, which allows a user to connect a color appearance to a float parameter without any intermediate nodes.

The details of how to convert from one type to another are specified using special conversion templates. These templates are not visible to the user, and have some limitations, but are otherwise no different than any other template.

Here, for example, is the template that defines how to convert a color to a float:

```
template color ConvertColorToFloat {
    display hidden
    description {
        Template for automatic type conversion from color to float
        Takes luminance of color
    }
    parameter color in {
        provider connection
    }
    parameter float out {
        access output
    }
    RSLInclude "pxslUtil.h"
```

```
        RSLSource StaticFunction {
            void pxslConvertColorToFloat(
                color in;
                output float out;
            )
            {
                /* calculate luminance */
                out = pxslLuminance(in);
            }
        }
    }
```

As you can see, it's pretty simple. It's just a template that has a color for an input, a float for an output, and conversion code inside.

Slim knows to use this template when performing this conversion because of the following preference set in slim.ini:

```
SetPref TypeConversionTemplate(color,float) \
        pixar,ConvertColorToFloat
```

This preference tells Slim that a user is free to connect a color appearance to a float parameter, and also the name of the template to use when generating the shader.

You can convert between your own custom types by writing conversion templates and setting preferences like the one above.

Inline Connections

When developing templates, you may encounter cases where, for the sake of modularity, you split your functionality among two or more templates. It would be useful, though, if that fact were transparent to the user.

This hidden modularity can be accomplished using **inline** connections. With an inline connection, the two (or more) connected nodes appear as one in the Palette and Appearance Editors.

You can specify an inline connection on the detail line, e.g.:

```
detail mustvary pixar,MicroManifold inline
```

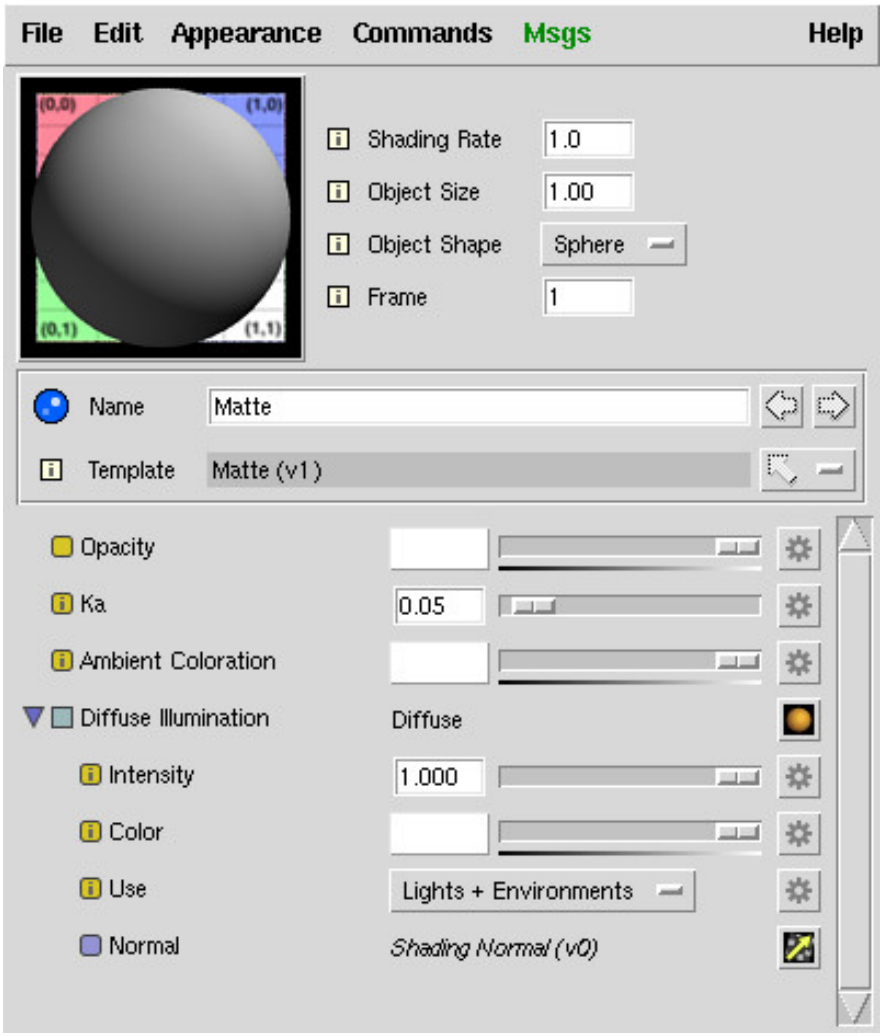
The connection to this template will resemble a collection. Opening the collection instantiates the template and presents the parameters of the template for editing. Because the node will not be represented in the Palette Editor, it cannot be disconnected. It can only be deleted.

You may wish to specify that a template is always instantiated with an inline connection. To do this, add:

```
drawmode inline
```

to the properties of your template.

As an example, the Matte template shown below makes an inline connection to a Diffuse template:



As you can see, but for the purple disclosure widget, this appears just like a collection. But because it is a connection to another template, we have a common Diffuse template that can be used by multiple shadingmodels. Plastic, for example, contains an inline connection to the same Diffuse template.

Shading Components and AOVs

The Diffuse template referred to above is a **shadingcomponent** template. Shadingcomponent is a compound type which contains several parameters describing the illumination contributions of the shadingcomponent. The first element in a shadingcomponent is the overall color contribution. What follows is a list of parameters containing the contribution to each of many different illumination styles.

This set of contributions flows through the entire system of combining and layering. A shader fitted to handle this data will know, in addition to Ci, the total contribution from ambient lighting, diffuse lighting, specular lighting, and can output this information as Arbitrary Output Variables (AOVs).

This is the current list of contributions that are tracked in a shadingcomponent:

- **surfacecolor** (the unlit surface color)
- **incandescence**
- **ambient**
- **diffuse**
- **thintranslucence**
- **subsurfacescattering**

- **backscattering**
- **specular**
- **rim**
- **reflection**
- **refraction**

Every factory shadingcomponent template will contribute to one or more of these styles of illumination. Here, for example, is the template for an Ambient shadingcomponent:

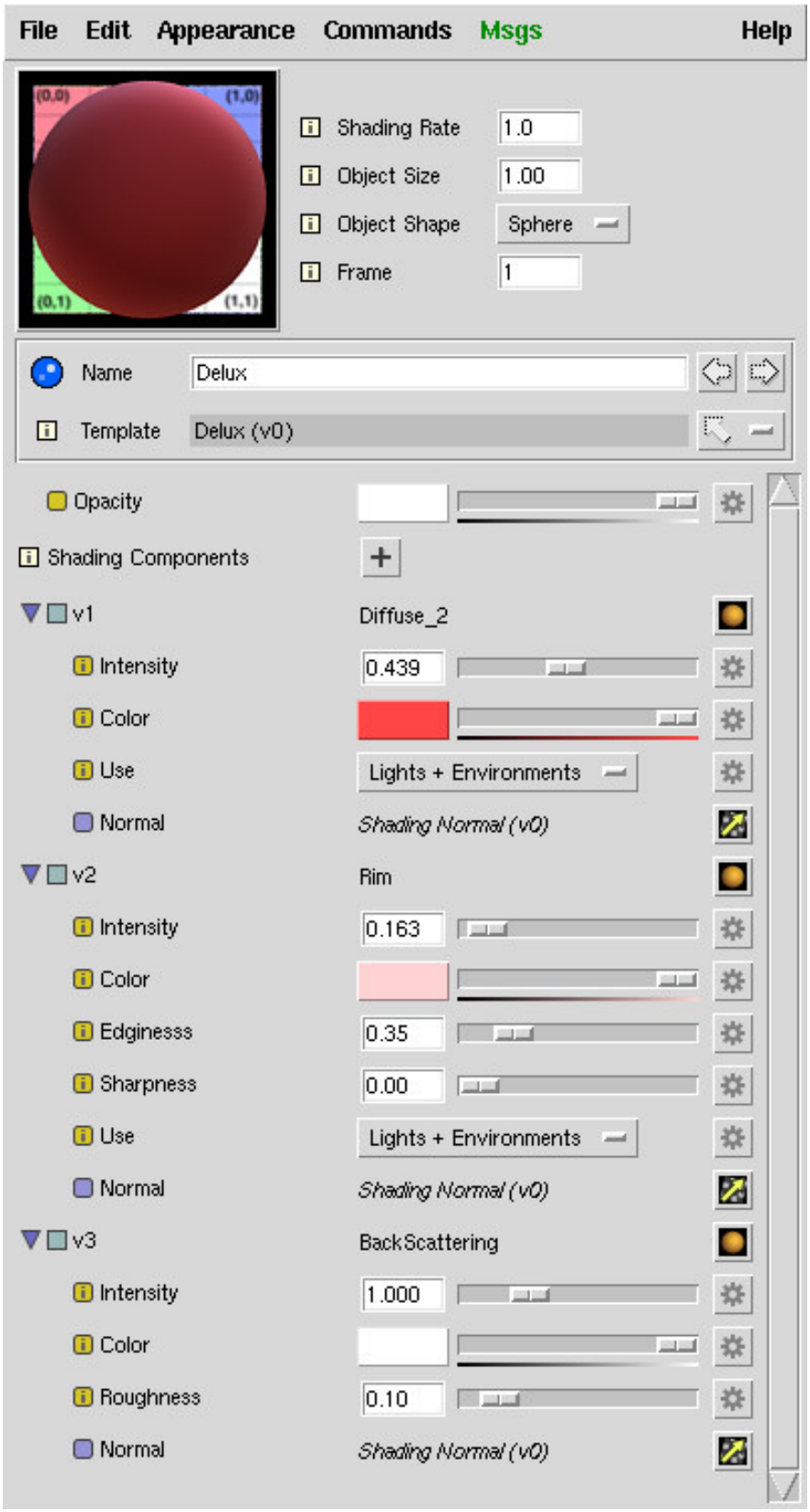
```
template shadingcomponent AmbientSC {
    label Ambient
    description {
        Ambient illumination
    }
    parameter float Intensity {
        description "Intensity of ambient component"
        range {0 1 .001}
        default 1
        detail varying
    }
    parameter color Color {
        description "Color of ambient component"
        detail varying
        default {1 1 1}
    }
    collection shadingcomponent result {
        access output
        display hidden
        parameter color col {
            access output
        }
        eval [::aovOutputParameterMacro]
    }
    RSLInclude "pxslAOV.h"
    RSLSource StaticFunction {
        void pxslAmbientSC(
            float Ka;
            color coloration;
            output color col;
            DECLARE_AOV_OUTPUT_PARAMS
        )
        {
            INIT_AOV_OUTPUT_PARAMS

            col = 0;
#if SLIM_SHADERTYPEID != SLIM_TYPEID_light
            col = coloration * Ka * ambient();
#endif
            c_ambient = col;
        }
    }
}
```

As you can see, macros (both in Tcl and in RSL) have been defined to assist in the delcaration and

initialization of the contribution parameters. The important step is the last line of the Ambient function, where it sets the c_ambient parameter (delcared via the DECLARE_AOV_OUTPUT_PARAMS macro) to the amount of ambient light. This value will flow through illumination functions and layering functions and eventually be written to an AOV. Automatically.

The most powerful use of shadingcomponents is via the Delux shader, which combines an arbitrary set of shadingcomponents. Here we have used Delux to combine Diffuse, Rim, and BackScattering to create soft, velvetine shader:



Delux is a shadingmodel_aov template, which means it will accept illumination contributions from

shadingcomponents and set them to Arbitrary Output Variables. As you can see, it does little more than allow the user to add new shadingcomponents which are editable using inline connections. This means that you can make Delux even more powerful by creating new shadingcomponent templates. This allows you to concentrate on a specific effect (CD-like diffraction? jewel-like glimmer?) without worrying about reimplementing things like diffuse or ambient. It also means your users are easily able to combine effects in ways you might not have imagined.

Message Handlers

You might like to reconfigure an appearance based on the value of a parameter, or more generally, execute a script whenever the value is changed. Message handlers allow you to trace the execution of methods like SetValue and execute a script to handle it.

Here is an example of using a msghandler to show or hide one property based on the value of another. In this example, we have a torattribute
called "BakeStyle" which can be set to one of three different values: ptcToTex (texturemap), ptcToBkm (brickmap), or ptcToPtc (pointcloud). The second torattribute, "TextureResolution", only applies when baking a texture, so we'd like to when BakeStyle not set to ptcToTex.

The msghandler below is executed everytime the **SetValue** method is called, which is what the Appearance Editor calls when a user changes the value. Via substitution, it is given the handle of the BakeStyle property. With the use of some property methods, the msghandler queries the current value, determines the appearance in which the property resides, and then finds the TextureResolution property that exists in that appearance. Then, based on the current value of BakeStyle, it either shows or hides the TextureResolution property and updates the editor accordingly:

```
torattribute string BakeStyle {
    default ptcToBkm
    detail uniform
    subtype selector
    range {
        texturemap ptcToTex
        brickmap ptcToBkm
        pointcloud ptcToPtc
    }
    msghandler {
        SetValue {
            # run this everytime the "SetValue" method is called

            # obj will be replaced by the handle of this property
            set prop %obj

            # get current value, and the appearance of this property
            set val [$prop GetValue]
            set app [$prop GetAppearance]

            # find the property that we are showing/hiding
            set resProp [$app GetProperties -name TextureResolution]
            set curLevel [$resProp GetDisplayLevel]

            if {$val == "ptcToTex"} {
                # BakeStyle is texturemap; make texture res visible
                set newLevel ""
```

```
            } else {
                # it's something else; make texture res hidden
                set newLevel hidden
            }

            # update display level, editor
            if {$newLevel != $curLevel} {
                $resProp SetDisplayLevel $newLevel
                $app UpdateEditor
            }
        }
    }
}
torattribute float TextureResolution {
    default 512
    display hidden
    subtype selector
    range { 64 64 256 256  512 512 1024 1024 2048 2048 4096 4096 }
}
```

A few notes of interest:

- Though torattributes are used in this example, the same msghandler could be used for parameters or any flavor of attribute.
- Note that TextureResolution is declared as hidden in template. This is because BakeStyle's default value is ptcToBkm.
- Because the entire body of a msghandler will be copied to any appearance that is created from the template, you may find it useful to place your msghandler code in a common proc that is referenced by the template. Not only will this keep palettes smaller; it allows you to change the code without having to reload the template for existing appearances.

  The Bakeable template from which this example derives does this very thing. Rather than using the code above, it calls a common proc that is used by several templates. Here is the msghandler used by the Bakeable template:

```
    msghandler {
        SetValue {
            ::Slim::CustomUI::pixar::Vis::UpdateVisibility %obj {
                ptcToTex {
                    "" TextureResolution
                }
                * {
                    hidden TextureResolution
                }
            }
        }
    }
```

  Don't be frightened by the "CustomUI" in the proc name. It's simply part of the namespace. UpdateVisibility is a general proc which takes the handle to the property and a description of the parameters to show and hide based on the value of the property. You will also find this used by the EnvMap template and the Combine template. Feel free to use it in your own templates.

Message Handlers may defined for these methods of AtomicProperties:

- CreateConnection
- SetConnection
- SetValue
- ScrubValue (not an actual method, this traces when the user drags a slider)
- SetValueProvider

Versioning

Though you will undoubtedly spend much time fretting over the design of your template before releasing it to users, you will often find that you will need to make a change after it has been deployed. For this situation, Slim has a template versioning mechanism which you can use to push out changes without worrying about breaking existing work.

If you look at how a palette is saved you will notice that all of the parameters in an appearance is saved, but the RSL Code is not. This keeps the heart of a template (the RSL) central, which means small fixes can be easily deployed.

The downside of this is that if the definition for an RSL function changes, a palette may become out-of-sync with the template. If, for example, a palette is created using a template, and then a new parameter is added to that template, the appearance in the palette will have less parameters than the template. This will often result in errors when compiling the shader. These errors will persist until the user reloads the template.

There are two ways to prevent this from happening. First, Slim 6.5 has an Auto Template Reloading feature. Whenever the current version of a template is requested by a palette, Slim checks to see if they have the appearance and the template have the same number of parameters. If they don't, the template is reloaded.

You can take matters into your own hands, though, by creating a new version of the template. This new version will be used for any newly-created appearances, but the old one will remain present for any existing palettes.

A template's version is declared on the first line of the template, using the templateV construct:

```
templateV float Filter 1 {
    ...
}
```

This declares the template as version 1. Using the versionless `template` construct is equivalent to declaring version 0 of a template.

To make a new version of a template you must:

1. Make a copy of the current template. You'll need this to support existing palettes.
2. Create the new version of the template by editing the template and bumping the version.
3. Take the old version of the template and place it in a `legacy` subdirectory.
4. Update slim.ini. The new template should be registered using the new version number, but the old template should be registered as well. By declaring that the old version is in a legacy directory, Slim will know not to place that version in the menus.

You should verify that you have proceeded correctly by loading a palette using the old version of the template. If the old version has been successfully stored as a legacy template, the appearance should continue to render, and its legacy status should be indicated via a red template name:

More Examples

Looking for more examples of templates? Every appearance that you can create in Slim is based on a template, and all of the templates that ship with Slim are plain text files that you can use as reference. Slim's templates are located in `RATTREE/lib/slim`, RATTREE

Prev | Next

Slim Files

Slim File Format

Background

We use the Slim file format to encapsulate the state of Slim and to represent Slim extensions. The format is syntactically very simple and is based on the TCL programming language (see **http://tcl.sourceforge.net/**). A TCL based parser can be built up quite trivially given the knowledge that each context, described below, supports a predefined set of legal commands. This notion that a only a certain set of keywords is meaningful at a given point in the parsing process is fundamental to all file formats and can be represented in modern TCL (post 8.0) using TCL's namespace facility. To write a TCL parser for the .slim file format, one could simply construct a number of namespaces each of which represents a parsing context and is comprised of a set of procedures which embody the set of keywords available in that context. Each keyword is implemented as a procedure whose arguments can either be fixed or variable in number according to standard TCL practice. Now, in order to parse a file, one must simply source it into the appropriate TCL context. The implementations of the keyword procedure must simply pull the arguments from their argument list and do something with them. Where keywords are used to represent hierarchical structure, the implementation must also make recursive calls to TCL's namespace eval command.

But, hey, we've already written such a parser so why should you care about the details? Suffice it to say that, well, people ask. In any event, the syntactic structure of .slim files is so simple that it may not be necessary to actually parse them. Simple sed or perl scripts could also be used to process them in interesting and automated ways. In any event we've included a stripped down version of a .slim file parser here for your perusal. Note that to put this parser to useful purpose you'll need to implement objects that accept all the messages and build up appropriate state.

But back to the task at hand. What exactly is the .slim file format? The first thing to keep in mind is that its actually just a TCL script. As such it abides by all the standard TCL conventions for commenting, new lines, braces and quotes. A .slim file is parsed by simply sourcing it into an appropriately prepared TCL interpreter.

Global Context

At the outermost scope, a Slim file must contain one or more slim commands. All slim data is nested within in the body of the slim command. Here's the format of the slim command:

```
slim version type creator body
```

Here, type can be one of: extensions, palette or appearance.

And here's a skeleton of a .slim file containing an encapsulated palette:

```
slim 1 palette slim {
    palette plt1 {
```

```
        function shadingmodel "Blinn_80" "pixar,Blinn" {
            identity 0ZgCEW39ZIB00000
            description {Mimics the Maya Blinn shader.}
            master {$torShaders/$INSTANCENAME}
            previewinfo {
               shadingrate 5
               objectsize 1
               objectshape Sphere
               frame 1
            }
            parameter color SurfaceColor {
              default {0 .25 1}
              provider connection
              detail varying {}
              label {Surface Color}
              connection 0ZgCEW39Z-F00000
            }
           ... more parameters...
        }
        ... more functions and instances ...
    }
    ... more palettes ...
}
```

As you can glean from this example context is everything. For example, the identity command is only meaningful in the context of an object that has an identity. Palette commands are only meaningful in places where palettes are expected and not, for example, where parameters are expected.

The remainder of this document details the various parsing contexts and the commands, or keywords, that are meaningful in these contexts. Keep in mind that, while the .slim file format is rather general, it is used for three primary purposes:

1. To store Slim extensions - templates, custom UIs
2. To encapsulate the current Slim state
3. To encapsulate a single Slim appearance.

But the best place to learn about the .slim file format is to simply peruse existing files. Fortunately there's no shortage of these. See, for example:

- the output of the **toslim** utility program. When you import a .slo file, Slim uses the toslim utility to generate temporary .slim files.
- the templates that ship with the system. These can be found at $RATTREE/lib/slim.
- the results of saving one of your palettes.

Context Summary

context	description
global	The outermost scope of a TCL file.

slim	The slim context is the container for all Slim data. A small set of keywords are legal within the slim context and depend on the type field of the slim command.
extensions	The extensions context is valid only within the slim context and is used to collect any number of customui, expressionui and templates.
palette	The palette context is valid within the slim context and the palette context. Palettes are containers of appearances.
appearance - instance, template, function	The appearance context is an abstract context representing the similarity between instances, templates and functions. Appearances are containers of properties.
instance	The instance context is valid within palette contexts. An instance is an appearance that's based on an externally defined shader. Instances are containers of properties.
template	The template context is valid within extensions contexts. A template is an appearance that's used to generate functions and is fundamental to Slim's shader creation features. Templates are containers of properties.
function	The function context is valid within palette and function contexts. A function is an appearance that's based on a template.
property - parameter, collection, attribute, torattribute	The property context is valid within appearance contexts and is an abstraction encompassing the parameter, collection, attribute and torattribute contexts.
parameter	The parameter context is valid within the appearance and collection contexts. Parameters are used to collect all information associated with a single parameter of an appearance.
collection	The collection context is valid within the appearance and collection contexts. Collections are containers for parameters and can have a custom UI. Collections are also used to represent higher order datatypes like shadingmodels and arrays.
attribute	The attribute context is valid within appearance and collection contexts. Attributes are used to collect all information associated with a single RenderMan attribute. Attributes differ from parameters only in that they are not formal parameters to an appearance.
torattribute	The torattribute context is valid within appearance and collection contexts. TORAttributes are used to collect all information associated with a single TOR attribute. TORAttributes differ from parameters and attributes only in that they are not formal parameters to an appearance and have no one-to-one RIB representation.
previewinfo	The previewinfo context is valid only within appearance context. It's used to collect all information related to preview rendering.
cmdui	The cmdui context is valid only within the extensions context. It is used to collect all information associated with a custom command user interface.

customui	The customui context is valid only within extensions contexts. It's used to collect all information associated with a custom user interface.
expressionui	The expressionui context is valid only within the extensions contexts. It is used to collect all information associated with a custom expression user interface.

Keyword Summary

keyword	description	context
access a	sets a parameter's access mode. Value values are input (default) and output.	parameter
attribute type nm body	creates a new attribute in the current appearance. type is one of the primitive types: float, color, string, point, vector, normal, matrix. Nm is the name of the attribute. Additional characteristics of the attribute can be nested within the body.	appearance, collection
cmdui nm body	registers a new cmdui under the current vendor namespace name *nm*. The body argument is used to nest the TclTkSource and invocation keywords. A cmdui appears within Slim as a menu command which, when invoked, executes associated TCL extensions.	extensions
collection type nm body	creates a new collection property and adds it to the current appearance. type is the data type of the collection. nm is the collection's name and all other characteristics can be nested within the parameter. Collections are used to group parameters and attributes as well as to construct higher order data types. They should also be used to represent RenderMan array parameters.	appearance, collection.
connection id	establishes that a parameter is connected to the function whose id is *id*.	collection, parameter
customui nm body	registers a new custom ui under the current vendor namespace and named *nm*. The body argument is used to nest the TclTkSource keyword which contains the source for the customui widget.	extensions
default d	establishes a default value for a parameter. Non scalar data types should be embedded within braces or quotes according to standard TCL syntax.	parameter, attribute, torattribute
description d	sets the description for the current context.	palette, appearance, property

detail d default flags	sets the detail field for the current parameter. The detail is used to determine where a parameter can be connected to other functions. Valid values are: uniform, varying and mustvary. The default field can be used in the mustvary case to establish a default connection. In this case the value should be the template ID of the default connection. You can optionally use the "inline" flag to specify that the connection should be an inline connection.	parameter
display d	sets the display mode field for the current property. Useful to hide parameters from the user. Valid values: hidden, visible.	property
drawmode m	sets the drawmode field for the current collection. Used to disable the drawing of the collection widget. Valid values: all, children.	collection
expressionui type nm body	registers a new expression ui. *type* is the data type that the expressionui can be used for, *nm* is a descriptive label for the expression ui. *body* is used to nest the definition of the expressionui and must contain a LaunchExpression keyword.	extensions
frame f	sets the frame for preview rendering purposes.	previewinfo
function type nm template body	creates a new function object in the current context. type is the function's return type, nm is function's descriptive name, template is the template ID of the template that the function is based on. Body is used to nest all the properties of the function.	palette, function
icon i	sets the icon for an appearance. icons are represented in a special compressed ASCII form that can be easily parsed.	appearance
identity id	sets the identity for the current appearance. This identity is used to establish connections between functions within Slim and to establish a relationship between a client's geometric objects.	appearance
index i	sets the array index of a parameter. Parameters are deemed array elements when their index is not -1 and they are members of a collection.	parameter
instance type nm master body	creates a new instance object and adds it to the current palette. *type* is the instance's type - surface, displacement, volume, light. *nm* is the descriptive label for the appearance and *master* is a string representing the instance's master. It is this string that's used to invoke the shader via the RIB file. *body* is used to nest additional properties of the instance.	palette
invocation menupath cmd	registers an invocation for the current cmdui. You can have any number of invocations for a given cmdui and these will appear in Slim menus as provided by the menupath argument. Currently two root menus are supported: PaletteEditor and AppearanceEditor.	cmdui
label l	sets the label for the current object. Labels are used only for display purposes. Property names are not editable and must uniquely represent a property within the context of an appearance.	property

LaunchExpression l	sets the launch expression for a custom expression ui. Slim automatically substitutes %f with the name of the temp file used to transmit the expression.	expressionui
lighttype t	sets the lighttype for a lightsource appearance. Valid values are: distance, spot, point and environment.	appearance
master m	sets the master reference for functions. This is used to both generate shaders as well as to reference the shaders in the RIB file.	function
modified time	sets the time when the appearance was last modified. This field is used to maintain shader dirty status between slim sessions. See this note for more details.	function
msghandler body	sets the list of handler actions for messages of a parameter. Body consists of messages (SetValueProvider, SetConnection, SetValue) and the handler code to be executed. Slim substitutes %obj for the parameter receiving the message.	parameter
node *id x y*	specifies the location of an appearance node within a graphArea.	graphArea
objectshape s	sets the object shape for preview rendering. Valid values are: Sphere, Cylinder, Torus, Cube, Teapot, Plane,	previewinfo
objectsize s	sets the object size for preview rendering.	previewinfo
offset *x y*	sets the x,y offset of the current graphArea	graphArea
palette nm body	creates a new palette object named nm and adds it to the current context.	slim, palette
palettereference file	causes the palette described in *file* to be loaded. The palette is marked external.	slim
parameter type nm body	creates a new parameter and adds it to the current context. type is the data type of the parameter and should be one of the RenderMan primitive data types: float, point, vector, normal, color, string, matrix. *nm* is the parameter's name and must be unique within the appearance. *body* is used to nest additional information about the parameter.	appearance, collection
previewinfo body	collects all information related to preview rendering for an appearance.	appearance
provider p	sets the provider for a property. The provider determines where the value of a parameter is obtained from. Valid values are: constant, variable, expression, connection.	collection, parameter
range r	sets the parameter's range for GUI purposes. The format of range depends on the subtype of the parameter. For scalar value the range is a vector comprised of a min, max and optional resolution field: `{0 1 .01}`. For selector subtypes, the range is a list of pairs representing the label and value for a menu of options: `{XYZ 0 X 1 Y 2 Z 3}`	parameter, attribute, torattribute.

RSLDefine *expr*	registers *expr* as a preprocessor definition (#define) to be output in RSL code generation process. This is generally useful for templates of type StaticFunction only. For DynamicFunctions and DynamicShaders, we recommend the use of the new include & define keywords.	template
RSLFunction body	sets the RenderMan Shading Language source code for the current template.	template
RSLInclude *file*	registers file as a preprocessor include (#include) to be output in RSL code generation process. This is generally useful for templates of type StaticFunction only. For DynamicFunctions and DynamicShaders, we recommend the use of the new include & define keywords.	template
RSLMain body	sets the RenderMan Shading Language source code generator for the current template.	template
RSLSource type body	sets the RenderMan Shading Language source code for the current template. *type* determines whether the source is quoted verbatim or used to generate RSL. Valid values for *type* are: • StaticFunction • DynamicFunction • DynamicShader	template
shadingrate s	sets the shadingrate for preview rendering purposes.	previewinfo
slim version type creator body	initializes the slim parsing context. *version* is used to control parsing behavior and should be set to 1. *type* is the file's type and should be set to one of: extensions, appearance or palette.	global
state s	sets the state of collection widgets. Valid values are: open, closed, locked.	collection
subtype s	sets the subtype for the current property. Valid values are: slider, vslider, switch, selector, bigstring, environment, reflection, shadow, depth, texture.	property
userdata data	sets user data for a property, appearance or palette. userdata consists of a list of name/value pairs as set by the user.	property, palette, appearance
userrange range	sets the range for a parameter with a value or precision that has gone beyond the recommended range as specified in a template.	parameter
TclTkSource body	sets the TCL/Tk source for the current custom ui or cmdui.	customui, cmdui
torattribute type nm body	creates a new torattribute and adds it to the current appearance. *type* is one of the primitive RenderMan data types. nm is the identifier for the torattribute and must be unique within the appearance. *body* is used to nest additional information about the torattribute.	appearance, collection
value v	sets the value of the current parameter. Non scalar data types should be embedded within braces or quotes according to standard TCL syntax.	parameter, attribute, torattribute

Tracking Modifications Between Sessions

When you are interactively working with a palette, Slim keeps track of changes that will require a regeneration and recompilation of the shader (or *master*) in order to take effect. Every time you render your scene, Slim checks which appearances have been modified since their shaders were last generated and regenerates/ recompiles what is necessary.

For interactive work, this is a relatively simple process: every attachable appearance contains a flag indicating whether it is "dirty" or "clean." When an appearance is modified in a manner which will require changes to the shader (e.g. the value of an internal parameter is changed), it is marked as **dirty**. When the shader is regenerated, it is marked as **clean**.

Tracking this state becomes more complicated between sessions of Slim. If, when loading a palette, Slim were to assume that any existing shaders were invalid, you would always have to wait for those shaders to be regenerated. On the other hand, if Slim were to assume that any existing shaders were valid, you might unknowingly find yourself using old or invalid versions of the shader.

To maintain a consistent dirty state between sessions, Slim not only tracks **that** an appearance is modified, it keeps track of **when** an appearance is modified. This modification time (measured in POSIX fashion) is stored within the body of the shader as a comment:

```
/* shader modification timestamp: 1109976261 */
```

This same number is stored with the appearance when it is saved:

```
function shadingmodel_aov "Glass" "pixar,Glass#1" {
    ...
    modified 1109976261
    ...
}
```

When this appearance is next loaded, Slim compares the `modified` value in the palette to the timestamp stored within the shader:

- If the numbers are **equal**, Slim knows that the last modification to the appearance as stored in the palette has been propagated to the shader on disk. In this case, the shader is determined to be **clean** and is **not** regenerated for the next render.
- If the modified value stored in the palette is **greater** than the value recorded in the shader, there are changes that have been made to the appearance that have not been propagated to the shader. In this case, the shader is determined to be **dirty** and is regenerated for the next render.
- If the modified value stored in the palette is **less** than the value recorded in the shader, there are changes that have been propagated to the shader that were not stored in the palette. This may have occurred if a user made changes to an appearance, generated the shader, but abandoned those changes by not saving the palette. In this case, the shader is determined to be **dirty** and is regenerated for the next render.

The `modified` mechanism is new for Slim 6.5. Slim 6.0 employed a similar mechanism known as `mastermtime`. This mechanism differed from the one described above in that, rather than storing the time that an appearance was modified, it stored the time that the shader was generated (more precisely, the `mtime` of the file). The `mastermtime` mechanism had two main problems:

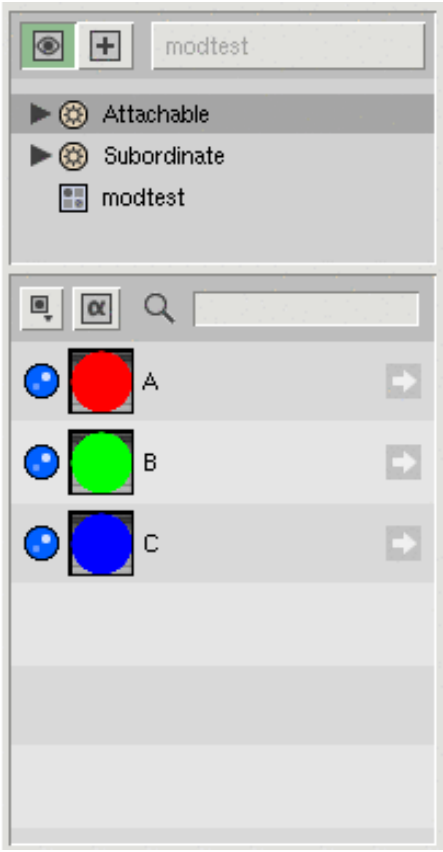
- Slim recorded the modification time of the file on disk for later comparisons. Unfortunately, because the `mtime` is updated when a file is copied, Slim would consider dirty any shader that had been copied to a new location for the sake of rendering. This resulted in shaders often being unnecessarily regenerated for rendering.
- This mechanism did not work well if multiple jobs were simultaneously working with the shaders in a common directory. Because Slim was tracking and checking when the shader was generated, the shader generated by one job was deemed to be dirty by the next job to come along. This meant that jobs would often "stomp on" one another.

These problems have been addressed by the `modified` mechanism:

- By recording the time within the body of the shader itself, the shader can be moved or copied and the time will not be affected.
- By recording the time that the appearance is modified (rather than the time that the shader is created), jobs running simultaneously will all agree on the modification time, so long as they don't themselves perform any modifications to the appearance.

To understand the benefits of the `modified` mechanism, you may wish to perform this experiment:

1. Check that your shader directory is clean of existing shaders
2. Open the Slim Message Log, set the Filter to Info, and make sure the Timestamp box is checked.
3. Create a new palette consisting of three Constant shaders and name them A, B, C.
4. Change A's Surface Color to red and perform a preview render.
5. Change B's Surface Color to yellow and perform a preview render.
6. Change B's Surface Color to green, but do **not** perform a preview render.
7. Save the palette.
8. Change C's Surface Color to purple and perform a preview render.
9. Close the palette without saving it.



Now reopen the palette and render icons for the three shaders. A, B, and C should be represented as red, green, and blue respectively, as seen here:

Now open the Message Log to see which shaders were regenerated/recompiled.

- Because all modifications saved for A (the change of Surface Color to red) were represented in its shader, A should not have been regenerated.
- Because B had a modification that was saved in the palette (the change of Surface Color from yellow to green) that was not represented in the shader, B should have been regenerated.
- Because C had a modification that was represented in the shader (the change of Surface Color from blue to purple) but not in the palette, C should have been regenerated.

Now close the palette without saving, reopen it, and render all of the icons. You should find that, because all of the shaders are now in sync with what had been stored in the palette, none of them were regenerated.

Converting Legacy Palettes

The format of palette files written by Slim 7 differs from that of previous versions. For the most part, Slim 7 will be able to read palettes written by previous versions of Slim. These attributes, however, will be lost in the translation:

- **Icons** : Appearance icons (i.e. the rendered preview swatches) have a different resolution in 7. Icons in legacy palettes are discarded when loaded into Slim 7.
- **Graph Work Areas** : The Network View is completely new in Slim 7, so data regarding Graph Work areas are discarded.
- **Window Information** : Palettes previously stored window geometry. Because of the one-window nature of Slim 7, this information is ignored.
- **Collected Appearances** : With the advent of Packages and Smart Palettes, appearances are no longer collected within one another. Any collected appearances in legacy palettes will be "exploded."
- **Bundles** : Bundles have been replaced by Packages in Slim 7. As above, bundles in legacy palettes will be exploded.

Palettes saved by Slim 7 can **not** be read by previous versions of Slim. When saving a palette created by a previous version, Slim will ask if you'd like to first make a backup.

##
Smoothstep.slim
Implements the shading language smoothstep function
##

```
slim 1 extensions pixartt {
   extensions pixar pxsl {
      template float Smoothstep {
         description {
            0 if input < minVal; 1 if input >= maxVal; otherwise, a
            smooth Hermite interpolattion between 0 and 1.
         }
         parameter float Input {
            detail mustvary
            description {
               Pattern to feed through smoothstep
            }
            default s
         }
         parameter float MinVal {
            label "Min Value"
            description {
               Threshold below which the function will return 0
            }
```

```
        default .2
        range {0 1 .001}
        subtype slider
      }
      parameter float MaxVal {
        label "Max Value"
        description {
          Threshold above which the function will return 1
        }
        default .8
        range {0 1 .001}
        subtype slider
      }
      parameter float Scale {
        description {
          Scale to apply to the result
        }
        detail varying
        default 1
      }
      parameter float result {
        access output
        display hidden
      }
      RSLSource StaticFunction {
        void
        pxslSmoothstep(float input, minVal, maxVal, scl;
                       output float result;)
        {
          /* calculate the standard smoothstep */
          result = scl * smoothstep(minVal, maxVal, input);
        }
      }
    }
  }
}
```

##
Filter.slim
A simple filtering funtion
##

```
slim 1 extensions pixartt {
   extensions pixar pxsl {
      template float Filter {
         description {
            Perform the selected filtering operation
            on the input.
         }
         parameter float Input {
            detail mustvary
            description {
               Pattern to feed through smoothstep
            }
            default s
         }
         slimattribute string Operation {
            description "Operation to perform on input"
            subtype selector
            range {
               Clamp clamp
```

```
        Remap remap
        Smoothstep smoothstep
    }
    default smoothstep
}
parameter float MinVal {
    label "Min Value"
    description {
        Threshold below which the function will return 0
    }
    default .2
    range {0 1 .001}
    subtype slider
}
parameter float MaxVal {
    label "Max Value"
    description {
        Threshold above which the function will return 1
    }
    default .8
    range {0 1 .001}
    subtype slider
}
parameter float Scale {
    description {
        Scale to apply to the result
    }
    detail varying
    default 1
}
parameter float result {
    access output
    display hidden
}
RSLSource DynamicFunction {
    proc primvars {} {
    }
    proc function {} {
        generateBody {
            switch [getval Operation] {
                clamp {
                    output "/* clamp */"
                    output "result = Scale * clamp(Input, MinVal, MaxVal);"
                }
                remap {
                    output "/* remap */"
```

```
                output "result = Scale * (MinVal + Input * (MaxVal-MinVal));"
            }
            smoothstep {
                output "/* smoothstep */"
                output "result = Scale * smoothstep(MinVal, MaxVal, Input);"
            }
            default {
                output "/* unrecognized Operation: [getval Operation] */"
                output "result = 0;"
            }
        }
    }
  }
 }
}
```

##
Switch
A simple generative function that switches between two inputs
##

```
slim 1 extensions pixartt {
   extensions pixar pxsl {
      template float Switch {
         description {
            Switch between two inputs
         }
         parameter float Input0 {
            label "Input 0"
            detail varying
            default 0
         }
         parameter float Input1 {
            label "Input 1"
            detail varying
            default 0
         }
         parameter float Which {
            label "Which Input"
            detail uniform
```

```
            range {0 1 1}
            default 0
        }
        parameter float result {
            access output
            display hidden
        }
        RSLSource GenerativeFunction {
            proc primvars {} {}
            proc function {} {
                generateBody {
                    output "if (Which > 0.5)"
                    output "{"
                    indent
                    set var1 [generate Input1]
                    output "result = $var1;"
                    exdent
                    output "}"
                    output "else"
                    output "{"
                    indent
                    set var0 [generate Input0]
                    output "result = $var0;"
                    exdent
                    output "}"
                }
            }
        }
    }
}
```