Vshade

The Visual Shader Authoring Tool

by Scott Iverson

Table of Contents

what is covered in the tutorial.

Gamma Correction for Display and Textures

The Vshade **Run** menu contains two items to help convert colors between the non-linear sRGB color space commonly used to display images and the linear color space used by Air for computation.

Display: When **Display Image in sRGB** is checked, the preview image is converted from linear rgb to sRGB color space prior to display.

Textures: When **Texture Color Space Is sRGB** is checked, Air will automatically convert colors in source texture files from sRGB color space to linear rgb space. This option does not affect textures that have been converted to the Air texture format.

By default both options are enabled.

2   What's New

Vshade 14.0 (December 2014)

- Support for the new stroke shader type in Air 14.
- New Split Parameters edit option to split a parameter block into two blocks.
- Support for integer and boolean parameter types.

3   Building Shaders

To begin a new shader, choose New in the File menu and select a shader template. The template determines the shader type and in most cases provides a basis upon which to construct a certain kind of shader. For example, the Surface_Plastic template includes a block with a plastic illumination model.

A visual shader is constructed by adding blocks from the component library and connecting them together in meaningful ways. Vshade comes with a large collection of prebuilt components that are useful in building interesting shaders.

Building simple surface shaders is easy with the components in the Materials category. For example:

- Start Vshade.
- Select **New** in the **File** menu and choose the **Surface_Plastic** template.
- Select **Save** in the **File** menu and save the new shader as `PlasticWood.vsl` in a convenient directory.
- Open the *Materials* category in the component list and select the **Wood** component.
- Click in the shader construction area and place the **Wood** component to the right of the **Plastic** block.
- Click on the Color output from the **Wood** block, then on the BaseColor input to the **Plastic** block to connect the two. A connecting line should appear.

- Click the **Render** button to see the new shader.

That's how easy it is to make a simple shader. The Tutorials section contains many brief tutorials on different aspects of shader construction.

Editing is Easier than Creating

For more complex shaders it is often better to start with an existing shader that is close to the shader you want. The AIR distribution includes dozens of prebuilt shaders that can be used as starting points for your own shaders. You can find the source for shaders included with AIR in the vshaders directory of your AIR installation.

3.1   Block Types

Topics:

Input Blocks
Output Blocks
Group Blocks
Function Blocks
Annotation Blocks
Illuminate Block
Solar Block

3.1.1   Input Blocks

An input block defines input parameters for a group block or a shader.

Use an input block in a shader to define variables that the user can change. When an input block is placed in a shader, any parameters will appear in the parameter list for the shader.

Use an input block in a group block to define variables that are supplied by connections to the block.

Creating an Input Block

- Click the Input button [icon] or select **New Input** from the **Blocks** menu to display a dialog for building a list of parameters.

For a shader, input parameters appear in the parameter list for the shader. For a group block, input parameters appear as the inputs to the block to which other blocks may be linked.

Adding a Parameter

- Click **Add** to add a new parameter.

Each parameter requires a name, type, and default value. The default value can be a constant, or, for functions, a global variable. Double-clicking a color value will display a color chooser dialog. Optionally you can provide a help hint for the parameter. Within a shader or group block each parameter name must be unique.

Editing a Parameter

- Double-click a parameter to edit it.

Deleting a Parameter

- Select a row in the parameter list and click **Delete** to remove a parameter.

Changing Parameter Order

- Select a parameter and use the **Up** and **Down** buttons to change its order in the list.

Editing an Input Block

- To edit an input block, double-click in the block's title bar or select the block and choose **Modify** in the **Edit** menu.

If you are within a group block that is connected to other blocks, you will not be able to edit its input parameters. To edit the input parameters for a group block, you must first break all connections to it by closing the group block and removing any links.

Input Block Order

You can have multiple input blocks in a shader or group block. The input parameters will appear in order in each block, and the blocks will appear in the order in which they were created. To change the order of the input blocks, select the input blocks in the order in which you want the parameters to be listed and choose **Re-Order** from the **Edit** menu.

3.1.2   Output Blocks

An output block defines the outputs from a group block or shader.

Creating an Output Block

- Click the Output button or select **New Output** from the **Blocks** menu to display a dialog for building a list of parameters.

Adding and editing output parameters works exactly like adding and editing input parameters.

3.1.3   Group Blocks

A group block is a container for a linked set of blocks that define a function. Group blocks are useful for creating new functions by combining blocks you already have. Opening a group block for editing reveals the enclosed blocks. Group blocks can contain input blocks (identifying the values the block expects to receive) and output blocks (identifying values the block exports), as well as function blocks and other group blocks.

Creating a Group Block

- Click the Group Block button or select **New Group Block** from the **Blocks** menu.
- Enter a name for the new block in the dialog that appears.

Vshade presents an empty page for constructing the group of blocks. Add blocks to the group just as you would to a shader. You will need input blocks to define inputs to the group block, output blocks for the results, and function or other group blocks to perform the computations.

Closing a Group Block

- When you are done creating the group, select **Close** from the **Blocks** menu or click the Up Level

  button .

The screen will display the previous level of blocks in the shader, and the new block will appear as a

group block in the center of the screen. You can move, connect, and manipulate a group block just like an SL Function block.

Editing a Group Block

- Double-click the title bar of a group block to re-open the block for editing or select the block and choose **Modify** in the **Edit** menu.

If there are input or output links connected to the group block, you will not be able to add, to edit, or to remove input or output blocks in the group.

3.1.4   Function Blocks

A function block is a primitive function defined by shading language code. Function blocks are provided for many common computational tasks. You can easily add new functions.

Creating a Function Block

- Click the SL Function button or choose **New SL Function** from the **Blocks** menu to bring up a dialog for creating a function block.

Input and output parameters are created and manipulated just like parameters for an input block. Output parameters do not need a default value.

Function blocks are implemented as short sections of shading language source code. The source code should use the input parameters (if any) to compute the output parameters or to set global variables.

Almost any shading language function can be turned into a function block. Keep the following pointers in mind when constructing Function blocks:

1. Any global variables that are used must be declared using the extern keyword.
2. Every input parameter has a default value that is used if no link is connected to it. The function should produce a valid result for any combination of linked and default input values.
3. The code editor is fairly primitive. If you are writing a complicated function, you may wish to use an external editor and copy and paste the source code when it is finished.

Validating Function Blocks

When **OK** is clicked to close the dialog, Vshade tests to see if the function will compile. If a compiler error occurs, an error dialog will appear and the Code dialog window will not close. To exit the dialog box you must either correct the error or select **Cancel**.

3.1.5   Annotation Blocks

An annotation block is used to add notes to a shader.

Creating an Annotation Block

- Click the Annotation button or select **New Annotation** from the **Blocks** menu to display the annotation dialog. Enter the note text and click **OK**.

3.1.6 Illuminate Block

An illuminate block is used inside a light shader to define illumination issued from a point. There are two types of illuminate blocks, one for point-like light sources and one for spotlights. The two types of illuminate blocks are contained in the shader templates for a point light and a spot light respectively.

An illuminate block is a special kind of group block. You can add input and output parameters to an illuminate block, but you should not rename it or remove any of the default parameters.

Within an illuminate block, the global variable L is set to the vector from the light source to the point being shaded. Add blocks within the illuminate block to set the global variable Cl - the color and intensity of light emitted by the light source.

A light source should contain only one illuminate block or solar block.

3.1.7 Solar Block

A solar block is used within a light shader to define light emanating in a single direction.

A solar block is a special kind of group block. You can add input and output parameters to a solar block, but you should not rename it or remove any of the default parameters.

Within a solar block, the global variable L is set to the direction from which light is incident. Add blocks within the solar block to set the global variable Cl - the color and intensity of light emitted by the light source.

A light source should contain only one illuminate block or solar block.

3.2 Common Block Operations

Selecting Blocks

There are several ways to select blocks:

- Click in the title bar of a block to select it.
- Drag a rectangle from *left* to *right* to select blocks enclosed by the rectangle.
- Drag a rectangle from *right* to *left* to select blocks contained in or crossed by the rectangle.

Hold down the shift key while making a selection to add to the current selection.

De-selecting Blocks

- Click in a blank area of the construction page to de-select all blocks.

Moving Blocks

- Click in a block title bar and drag with the mouse to move all selected blocks.

Editing Blocks

- To edit a block, double-click in the title bar, or select the block and choose **Modify** from the **Edit** menu.

Cutting, Copying, Pasting, Deleting, and Renaming Blocks

The standard editing operations of cut, copy, paste, delete and rename can be performed on any selection of blocks.

Changing the Value of an Input Parameter

- Double-click on an unlinked input parameter to display a dialog for changing the default value. If the parameter is a color, you can double-click on the default value in the Edit Parameter dialog to display a color chooser dialog.

Connecting Blocks

See Connecting Blocks.

3.3   Block Parameters

Parameter Types

Each parameter in Vshade has a type which determines the kind of information it stores.

Type	Data Type	Format
float	real numbers	a single number
color	color data	r g b components
point	location data	x y z
vector	directional data	x y z
normal	orientation data	x y z
matrix	transformation	16 numbers
string	string data	text
integer	whole numbers	a single whole number
boolean	true or false value	true or false

Arrays

Parameters can also be arrays containing more than one element of any of the primitive types.

Default Values

Every input parameter must have a default value. A default value is either a constant or a global variable of the appropriate type. The table below lists the global variables and their types:

Name	Type	Description
Cs	color	color attribute input
Os	color	opacity attribute input
P	point	surface position
dPdu	vector	X tangent vector
dPdv	vector	Y tangent vector
N	normal	surface normal at P
s	float	1st texture coordinate
t	float	2nd texture coordinate
u	float	1st parametric coordinate
v	float	2nd parametric coordinate
Ci	color	color output
Oi	color	opacity output
I	vector	incident ray

3.4   Connecting Blocks

Connecting Blocks

Information is passed from one block to another by forming connections between output and input parameters.

- To connect an input parameter of one block to an output parameter of another, click on the name of one of the variables (near the edge of the block), then on the name of the other variable.

Variables can only be connected if they are compatible. Vshade will make sure that an input and an output match before allowing you to make a connection. Here are the rules for type compatibility:

Type Compatibility

1. Each variable type is compatible with itself.
2. A float variable may be used as the input to any color, point, vector, or normal variable.
3. Point, vector, and normal types are interchangable with one another.
4. If a variable is an array, it can only be connected to another array variable with the same number of elements and the same element type.

Automatically Connecting Blocks

If you have multiple connections to make between two blocks, try using the **Connect** command in the **Edit** menu. The Connect function tries to form links between compatible variables of two selected blocks. If the resulting links are incorrect, select **Undo** from the **Edit** menu to undo the connections.

Disconnecting Blocks

To remove a single connection, click on the link to highlight it, then press the delete key or select **Delete** from the **Edit** menu.

To remove all connections from a group of selected blocks, use the **Break Connections** command in the **Edit** menu or toolbar.

3.5   Compiling and Previewing

A Visual shader must be compiled before being used. The default output directory is the `usershaders` directory of your AIR installaton. The output directory can be changed using the **Output Directory** item in the **Run** menu.

The **Run** menu contains items for compiling and previewing a shader. There are also buttons for commonly used items.

- The **Preview** item or button invokes the shading compiler to compile the current shader and then renders a preview image. Use the items in the View tab to set the options for the preview image, such as the scene file and the image size.

- The **Compile Only** item or button simply compiles the shader.

Previewing in Air Show

If the **Use Air Show** item is checked in the **Run** menu, the preview image will be displayed in Air Show instead of in Vshade. When displayed in Air Show, the preview image may also include any output variables in the shader if the **Include Output Variables** item is checked in the **Run** menu.

3.6   Interactive Parameter Tweaking

Vshade 3.0 and later support interactive parameter tweaking using SiTex Graphics' TweakAIR.

To start interactive tweaking, click the IPR button in the Vshade toolbar. During an interactive session, changes made to parameter values are sent to the TweakAIR rendering process which should update the preview image very rapidly.

If you add parameters to a shader or modify the shader logic, you will need to re-start the interactive session (by clicking the IPR button again) to see the changes reflected in the preview image.

3.7   Shader Documentation

Shader Information

Select **Information** in the **File** menu to display a dialog for entering information about the shader. You can provide a synopsis, author, copyright, and description.

Creating HTML Help

Vshade can automatically create an HTML help file containing information about a shader.

- Select **Render Help Image** from the **File** menu to render an thumbnail image for the help file. The image should be placed in the same directory as the help file. Vshade will render a 128x128 preview image using the current Run Time settings.
- Select **Create HTML Help** from the **File** menu to create an HTML help file for a shader using the shader information and the hints for the input parameters.

The default location for the help file is `$AIRHOME/shaders/html`. The shader description is place directly in the help file and can include HTML formatting tags.

Creating an LIF file

Some plugins use LIF files to obtain additional information about a shader. Chosing the **Create LIF File** item in the File menu will create an LIF file for the current shader in the shader output directory.

3.8   Compiler Errors

Occasionally you may receive an error message when you attempt to compile a shader.

One common source of errors is when one more input or output parameters have the same name. Each shader parameter must have a unique name.

Any other error will appear as an error message in a pop-up dialog. This case indicates a problem with Vshade or the shading compiler: Vshade should not allow you to construct a shader that will not compile. Please send the VSL file to support@sitexgraphics.com so we can fix the problem.

3.9   Debugging a Shader

The Vshade interface prevents you from making simple programming errors. Any shader you construct should compile and work according to the shading network you have constructed. Of course what you see is not always what you expect. Vshade provides a couple tools for finding out why an errant shader is producing unexpected results.

Rendering a Link

Vshade allows you to render the value of any float, color, or point-type channel as a color applied to the

preview image. To render a link, select it and click the **Render Link** button . You cannot render a link from within a group block.

If the preview image is displayed in Vshade, data values outside the range 0 to 1 will wrap around when rendering a link. If the preview image is rendered to Air Show, the data values are not wrapped, and you can use the pixel view feature in Air Show to examine the data values at each pixel (just hold down the SHIFT key while moving the mouse pointer over the image in Air Show).

3.10   Conditional Execution

When building complicated shaders, it is common to have shader components that only need to be evaluated under certain conditions.

Execution of a group block can be made conditional on one of its inputs by providing an input variable with a prefix of `If_` or `IfNot_`. The components in the group will only be executed if the If_ input is non-zero (or not null for a string parameter). If the components of a group block are not executed, any output variables from the group block are assigned their default values.

3.11   Loops

Looping with a Group Block

Vshade 2 allows a Group block to be executed repeatedly in a loop. Vshade looks for an input parameter with a Loop_ prefix. The input value for that variable is used as a loop counter and should be an integer. If the input value for Loop_X is N, the group block will be called N times with the Loop_X variables taking on the values 0 to N-1. Loop blocks are mainly useful in instancer and

procedure shaders.

Looping with an Iterator

Vshade 5 introduces a new iterator block for simplified looping without using a group block.

You can create a simpler iterator by selecting the Iterator item from the Blocks menu. Iterators are used primarily when constructing instancer or procedure shaders.

An iterator block is simply a code block with the special name Iterator and at least one output parameter. The first output parameter, which should be a float, is used as the control variable for the loop. The first time the iterator is evaluated, the control variable will be set to 0. The iterator should set the control value to a non-zero value to execute the loop body. The iterator should set the control variable to 0 to terminate the loop. Here's sample code for a simple loop that will execute 10 times:

```
if (Control==0) {
  Control =1
} else {
  Control+=1;
  if (Control>10) Control=0;
}
```

Any blocks connected to an output parameter of the iterator (either directly or via other blocks) will be executed as part of the loop body.

4   Component Library

Vshade comes with a large library of pre-built components. A component is simply a set of blocks that have been saved together in a single file. Most components contain a single block, but some contain a linked set of related blocks. Any set of blocks can be saved as a component. The left-hand side of the Vshade window has a tree list containing the components in the library. Components are organized into categories for convenience.

The library is stored in the directory $AIRHOME/viztools/vslib. Each component is stored as a separate file. You can use standard file-manipulation tools such as Windows Explorer to change the library. If you alter the library file structure while Vshade is running, choose the **Refresh** item in the **Library** menu to update the list of components in Vshade.

Viewing Components

To view the components in a category, click the ⊞ next to the category name.

Using Components

To use a component, select the component name in the library window. Then click in the shader contruction area to place a copy of the component. The blocks in the component will appear selected. Drag the title bar of any block to re-position the set.

User Library

Vshade provides a separate user component library where you can store your own components. By default the user component library is located in your home directory in SiTex\VshadeUserLib. You can change the location of the user library in the Library menu. Categories in the user library are listed at the top of the category list with the name enclosed in brackets.

Adding a Component to the Library

- Select the blocks you wish to include in the component. Any set of blocks may be saved as a component. Links between selected blocks are saved with the component; links to unselected blocks are not saved.
- Select the category in the user library list to which you wish to add the component.
- In the **Library** menu select **Add Selection**.
- Type a name for the component and press Enter.

Creating a New Category

- From the **Library** menu select **New User Category** and type a name for the new category.

Component Library Location

The standard component library can be relocated by moving the contents of `$AIRHOME\viztools\vslib` to a new directory and defining a `VSHADELIB` environment variable to point to the new directory.

4.1   Bump Layers

The *Bump Layers* category contains components that can be layered or blended together to produce composite bump effects.

BumpBlend

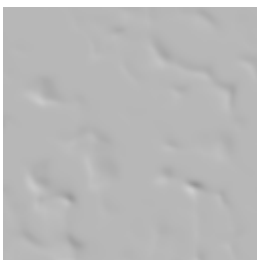   Component for blending bump values using any of the supported blend modes

BumpMapLayer

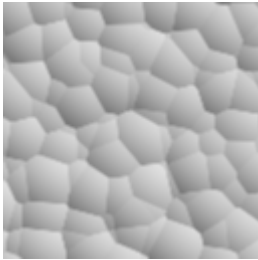   Applies a texture map as a blended layer

Dents

Nicks

4.2   Bumps

Components in the ***Bumps*** category are patterns for creating bump or displacement features using either a displacement shader or a surface shader that supports bump mapping.
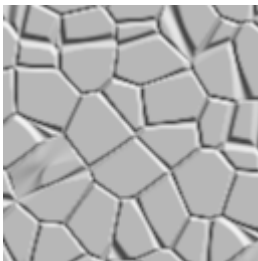
To make a simple displacement shader add any of the ***Bumps*** components to the `Displacement` shader template.

Cells

Rounded lumps based on the **Cells3D** pattern in the ***Patterns 3D*** category.
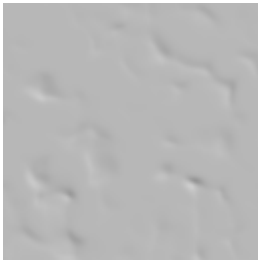
Cracked

Cracks based on the **CellBorder** pattern in the ***Patterns 3D*** category.

Dents
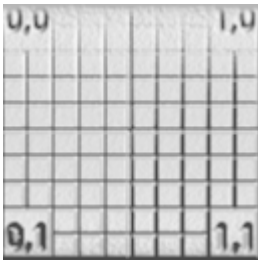

Dents based on **Turbulence** and a Power function.

Nicks

Nicks generated from a couple **Noise** functions.

Rough

Texture3DBump

TextureBump

The TextureBump component allows a texture map to be used to apply a bump pattern.

4.3   Color

The *Color* category holds components for manipulating colors.

Add

Adds colors

ColorFromRGB

Creates a color output from separate red, green, and blue input values

ColorRamp

Blends between two colors with Bias and Sharpness controls using a combination of Mix and TweakRange components.

ColorToRGB

Converts a color input to separate red, green, and blue output values

ColorTransform

Converts a color from one color space to another.

Combine

Adds two input colors with individual weights to produce an output color

Mix

Linearly blends between two colors based on an input float value

Multiply

Multiplies colors

Subtract

Subtracts colors

VaryColor

Provides separate inputs for varying the hue, saturation, or lightness of a color

VaryItemColor

Randomly varies the hue, saturation, and lightness of a color based on an item index value. This component can be used to give a slightly different color to each item in a discrete pattern like brick, tile, or cells.

4.4   DarkTree

Components for using Darkling Simulations DarkTree shader in an AIR shader. Windows only.

DarkTreeAll

Provides all the output values that can be produced by a DarkTree shader.

DarkTreeBump

Simplified DarkTree node providing only the bump elevation output.

DarkTreeColor

Simplified DarkTree node providing only the basic pattern color.

4.5   Float

The *Float* category holds basic arithmetic functions for float values.

Add

Adds numbers

Combine

Adds two input numbers with individual weights

Lerp

Linearly blends between two numbers based on an input float value

Multiply

Multiplies numbers

Subtract

Subtracts numbers

4.6   Functions 1D

The *Functions 1D* category contains a collection of standard mathematical functions.

Abs

absolute value

Ceil

smallest integer greater than or equal to X

Clamp

clamps the input to value to lie within the given interval

Exp

e raised to input power

Floor

largest integer less than or equal to X

Log

natural logarithm

Max

maximum of two values

Min

minimum of two values

Mod

Modulus function

Power

raises a number to any power

Round

the nearest integer to X

Sign

returns -1 if X is less than 0, 0 if X is 0, and 1 if X is greater than 0

Sqrt

the square root of X

4.7   Functions 2D

Functions for manipulating 2D coordinates:

Add2D

Adds a pair of 2D coordinates

Distance2D

Distance between two 2D points

Length2D

Length of a 2D vector

Multiply2D

Multiplies a pair of 2D vectors

Rotate2D

Rotates a 2D point

Subtract2D

Subtracts a pair of 2D points

XYfromPolar

Converts a coordinate pair from polar to rectilinear coordinates

XYtoPolar

Converts a coodinate pair from rectilinear to polar coordinates

4.8   Functions 3D

Functions for point, normal, and vector types.

AngleBlend

   Emits a float value based on the angle between two input vectors, optionally raised to a power

CrossProduct

   Computes the cross product of two vectors

Distance3D

   Outputs the distance between two points

DotProduct

   Returns the dot product of two vectors

FaceForward

   Flips the input normal if necessary to face towards the incoming direction

Length3D

   Returns the length of a vector

Normalize

   Normalizes the input vector

Rotate3D

   Rotates the input point about an axis

4.9   Globals

Globals components provide read and write access to global variables available to shaders.

Read Components

ColorIn	Cs	primitive Color attribute
ColorOut	Ci	output color
OpacityIn	Os	primitive Opacity attribute
OpacityOut	Oi	output opacity
Normal	N	surface normal
IncomingDir	I	incident ray from view direction
Position	P	surface location
TextureCoords	s t	standard surface texture coordinates

Write Components

SetColor   `Ci`   output color
SetOpacity   `Oi`   output opacity
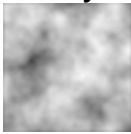SetPosition   `P`   surface location

4.10   Layers

Components that can be layered and combined with other shading patterns and each other to produce composite effects:

Blend

General component for blending two colors using any supported blend mode.

CloudLayer

Cloud pattern from a Brownian noise function

ColorLayer

Simple layer for blending a color

Decal2D

Applies a decal using standard texture coordinates on top of an underlying color or color pattern.

Decal3D

Applies a projected decal on top of an underlying color or color pattern.

StencilLayer

Applies a texture map as a colored stencil with selectable blend mode

TextureLayer

Adds a a texture map layer. The texture map is positioned using the object's standard texture coordinates.

4.11   Lighting Models

This category hold components for various illumination models that determine how a given surface responds to light. You can make a simple surface shader by creating a new shader using the standard `Surface` template and adding any illumination model.

All lighting models provide extra output variables to support multipass rendering.

AddBump

The **AddBump** component can be used to add bump mapping to any *LightingModel* component: after adding the component, connect the Normal output to the Normal input of an LightingModel component. Connect the bump pattern to the Bump input of the **BumpMap** block. See the tutorial A Surface Shader with Bumps for more details.

BrushedMetal
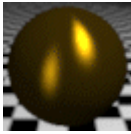
A metallic surface with fine grooves that make the specular highlight dependent on the orientation of the surface.
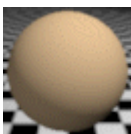
Clay
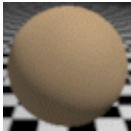
A rough surface without highlights using the DiffuseRough component from the *LightingPrimitives* category.
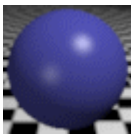
Glass
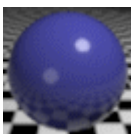
Basic glass component for refractive materials

Matte

Standard matte illumination model with Diffuse
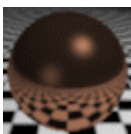
Plastic

A plastic surface without reflections using the standard Specular function.
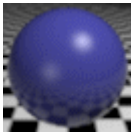
ShinyCeramic

A glossy surface with reflections.
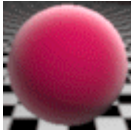
ShinyMetal

A metallic surface with reflections. The reflections and specular highlight are both modulated by the base surface color.

ShinyPlastic

A plastic surface with reflections that are more visible at glancing angles.

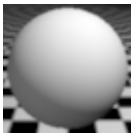**Velvet**

Simple velvet surface.

4.12   Lighting Primitives

These components implement primitive local illumination models. Combinations of primitive illumination models can be used to construct different surface appearances, such as the components in the *__LightingModels__* category.
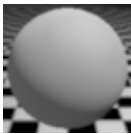
Ambient

Returns the ambient lighting component

Diffuse

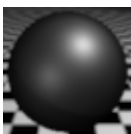standard Lambertian diffuse

DiffuseRough

diffuse reflectance for rough surfaces like clay
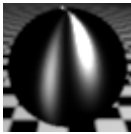
Specular

standard function for specular highlights
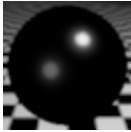
SpecularBlinn

alternate specular function due to Blinn

SpecularBrushed

specular for a surface with orientation-depended roughness like brushed metal

SpecularGlossy

specular for a glossy surface with control of the sharpness of the highlight edge

4.13   Materials

Material components contain a complete color pattern and sometimes a bump pattern. Use these as building blocks for simple surface shaders by creating a new shader with any surface template and adding a material. Materials can also be combined with other blocks to produce more complicated shaders.

Brick
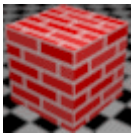
Colored brick based on the BrickPattern component in *Patterns 2D*

Granite
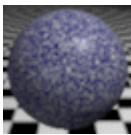
Granite made from the Turbulence component in *Noise*

Marble

Colored marble from the MarbleVeins component in *Patterns 3D*

Texture3D

Texture pattern using standard texture coordinates or a projection

TextureMap

   Texture pattern using standard texture coordinates
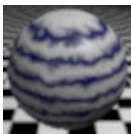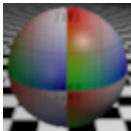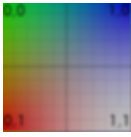
Tile
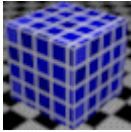
   Colored tile based on the TilePattern component in *Patterns 2D*

Wood

   Wood from the WoodRings component in *Patterns 3D*

4.14   Noise

Noise components provide stable pseudo-random numbers tied to particular input.

CellNoise

CellNoise returns a random number associated with the integer portion of the input coordinates. The output value is scaled to lie between the Zero and One inputs.

Brownian
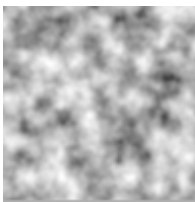
Computes a function known as fractional Brownian motion by summing several octaves of noise at different frequencies. The frequency of each sample increases by the Lacunarity parameter for each octave, while its contribution to the total sum decreases based on the Gain parameter. The output value is scaled to lie between the Zero and One inputs.

Noise

The Noise component returns a 1-dimensional noise function. The output value is scaled to lie between the Zero and One inputs.
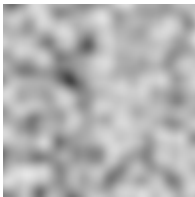
NoiseXY

The NoiseXY component returns a 2-dimensional noise function. The output value is scaled to lie between the Zero and One inputs.

NoiseXYZ
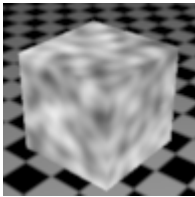
The NoiseXYZ component returns a 3-dimensional noise function. The output value is scaled to lie between the Zero and One inputs.

Turbulence
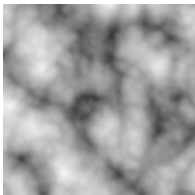
Turbulence computes a function that approximates turbulent fluid flow by summing octaves of a modified noise function. The frequency of each sample increases by the Lacunarity parameter for each octave, while its contribution to the total sum decreases based on the Gain parameter.

4.15 Patterns 1D

Patterns based on a single input signal.

PulseTrain
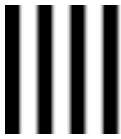
SawToothWave

SmoothPulseTrain
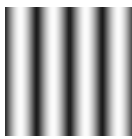
4.16   Patterns 2D

2-dimensional patterns.

BrickPattern
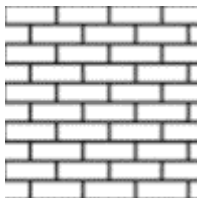
Cells2D
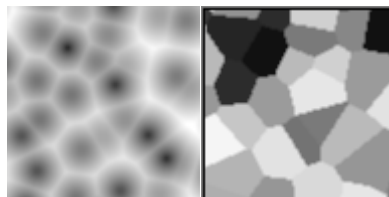
Gradient
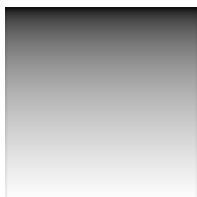
HexPattern
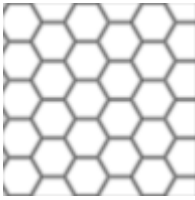
PlankPattern
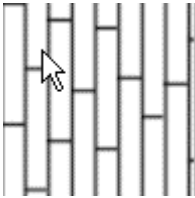
Spots2D

TilePattern
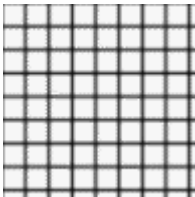
Weave
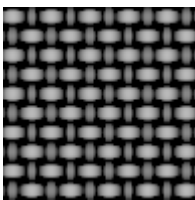
4.17   Patterns 3D

3-dimensional patterns.

CellBorder
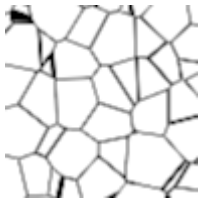
Computes the border between cells in a Voronoi-diagram-like pattern.

Cells3D
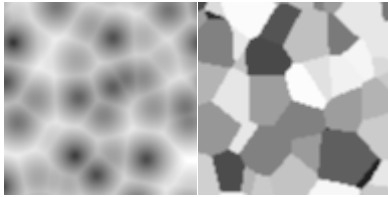
Cells3D returns the distance to the center of the nearest 3D unit cell as well as the center. The picture on the right shows the effect of using CellNoise to generate a unique value based on the nearest cell center.

MarbleVeins
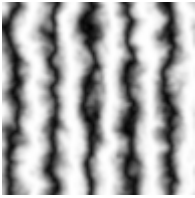
Simple marble pattern created with Turbulence.

Spots3D
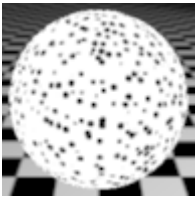
Spot pattern from a Cells3D component and a Step component.

WoodRings
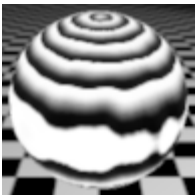
Wood ring pattern.

4.18   Point

The *Point* category contains components for basic arithmetic operations on point, vector, and normal values.

Add

   Adds two points or vectors

Combine

   Adds two input points with individual weights to produce an output point

Divide

   Divides one vector by another

Multiply

   Multiplies points or vectors

PointFromXYZ

   Creates a point from individual X, Y, and Z inputs

PointToXYZ

   Outputs the individual X, Y, and Z components of a point

Subtract

   Subtracts vectors and points

4.19   Query

Use the components in this category to query attributes, options, or dictionary entries.

AttributeColor
AttributeFloat
AttributeString

   The Attribute components return the value of the named attribute if found or the specified default value otherwise.

DictionaryColor
DictionaryFloat
DictionaryString

   The Dictionary blocks return the value of an entry in the specified dictionary file if it is found and the default value otherwise. A dictionary is organized as a simple text file of name value pairs, one entry per line.

OptionColor
OptionFloat
OptionString

The Option components return the value of the named option if found or the specified default value otherwise. For a list of currently supported options see the documentation for the option() function in the Air User Manual under

Shader Guide -> Shading Language Extensions -> option()

4.20   Reflections

Components for computing reflections.

BrushedReflections

Component for anisotropic or brushed reflections.

Environment Map

Provides a simple environment map lookup.

Reflections

This component combines ray tracing and environment-mapped reflections. If `ReflectionName` is "raytrace", ray tracing is used for reflections; otherwise `ReflectionName` is assumed to hold the name of an environment map. If the `Reflection` parameter is 0, no reflections are computed.

Trace

Simple encapsulation of the shading language `trace()` function.

4.21   RIB

Components in this directory issue RIB commands using the new ribprintf() statement in AIR 8. These components can only be used in a procedure or instancer shader.

The order in which RIB components are evaluated in the shader - and hence the order in which RIB commands are issued - is important. To provide a well-defined order, each RIB component has an Enable input parameter and a Next output parameter. Connecting the Next output from one component to the Enable input of a second component ensures that the first component is executed before the second component. For example, in the network below the Color block would be evaluated before the Sphere block:
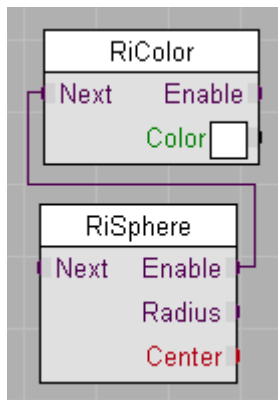
Each component disables it's output if the Enable input is 0. This behavior allows a sequence of linked RIB blocks to be turned on or off using the Enable component of the first block.

4.22   Shapes 2D

2D shape patterns with anti-aliasing

Disk

Line2D

Polygon

Rectangle

Ring

Star

SuperEllipse

4.23   Texture

Components for texture mapping.

ProjectDecal

Component for projecting a decal onto a surface. <u>DecalSpace</u> gives the <u>coordinate space</u> in which to perform the projection by applying the <u>DecalTransform</u> transformation. The <u>FrontOnly</u> parameter can be used to restrict the decal to one side of a surface.

Texture

Component for querying a texture map with parameters for scaling and translating the input texture coordinates.

Texture3D

Texture mapping component that uses the Project2D component to allow a choice of standard texture coordinate mapping or a number of <u>3D to 2D projections</u>.

4.24   Transformations

Components performing 3D transformations.

Project2D

Outputs a pair of texture coordinates based on standard texture coordinates or any of various <u>3D to 2D projections</u>.

ShadingPosition

This component converts the input surface location to a stable <u>coordinate space</u> suitable for shading such as `"shader"`, `"world"`, or `"object"` space. Use this component to generate the input point for a 3D pattern.

TransformNormal

Transforms a normal from one <u>coordinate space</u> to another.

TransformPoint

Transforms a point from one <u>coordinate space</u> to another.

TransformVector

Transforms a vector from one <u>coordinate space</u> to another.

4.25   Transitions

Components for modifying a single input signal.

Invert

Inverts the input signal between <u>Min</u> and <u>Max</u>

Pulse

Returns 1 when the signal is between <u>Start</u> and <u>End</u> and 0 elsewhere.

SmoothPulse

starts at 0, rises smoothly to 1 between <u>RiseStart</u> and <u>RiseEnd</u>, stays at 1 until <u>FallStart</u>, and then falls smoothly to 0 at <u>FallEnd</u>.

SmoothStep

Returns 0 when the signal is less than <u>RiseStart</u>, 1 when the signal is greater than <u>RiseEnd</u>, and smoothly blends between 0 and 1 for values between <u>RiseStart</u> and <u>RiseEnd</u>.

Step

Simple step function that returns 0 when the signal is less than <u>Edge</u> and 1 when the signal is above it.

TweakBump

Component for tweaking the appearance of bump mapping or displacement.

TweakRange

Function for modifying the domain and range of a signal as well as controlling the contrast (sharpness) and applying bias. See the related <u>tutorial</u> for more information.

4.26   Trigonometry

Standard trigonometric functions.

Sin, **Cos**, and **Tan** components all accept input in degrees, radians or cycles.

Similarly, the **ArcSin**, **ArcCos**, and **ArcTan** components return results in degrees, radians, and cycles.

5   Tutorials

1. <u>A First Surface Shader</u>
2. <u>A First Displacement Shader</u>
3. <u>A Surface Shader with Bumps</u>
4. <u>A Surface Shader with Texture Maps</u>
5. <u>Discrete Color Patterns: Simple Brick</u>

6. <u>Continuous Color Patterns: Simple Marble</u>
7. <u>3D to 2D Projections</u>

5.1   A First Surface Shader

This tutorial shows you how easy it is to make a simple surface shader based on the templates and
materials included with Vshade.

- Start Vshade.

Introduction to the User Interface

The Vshade window looks like this:
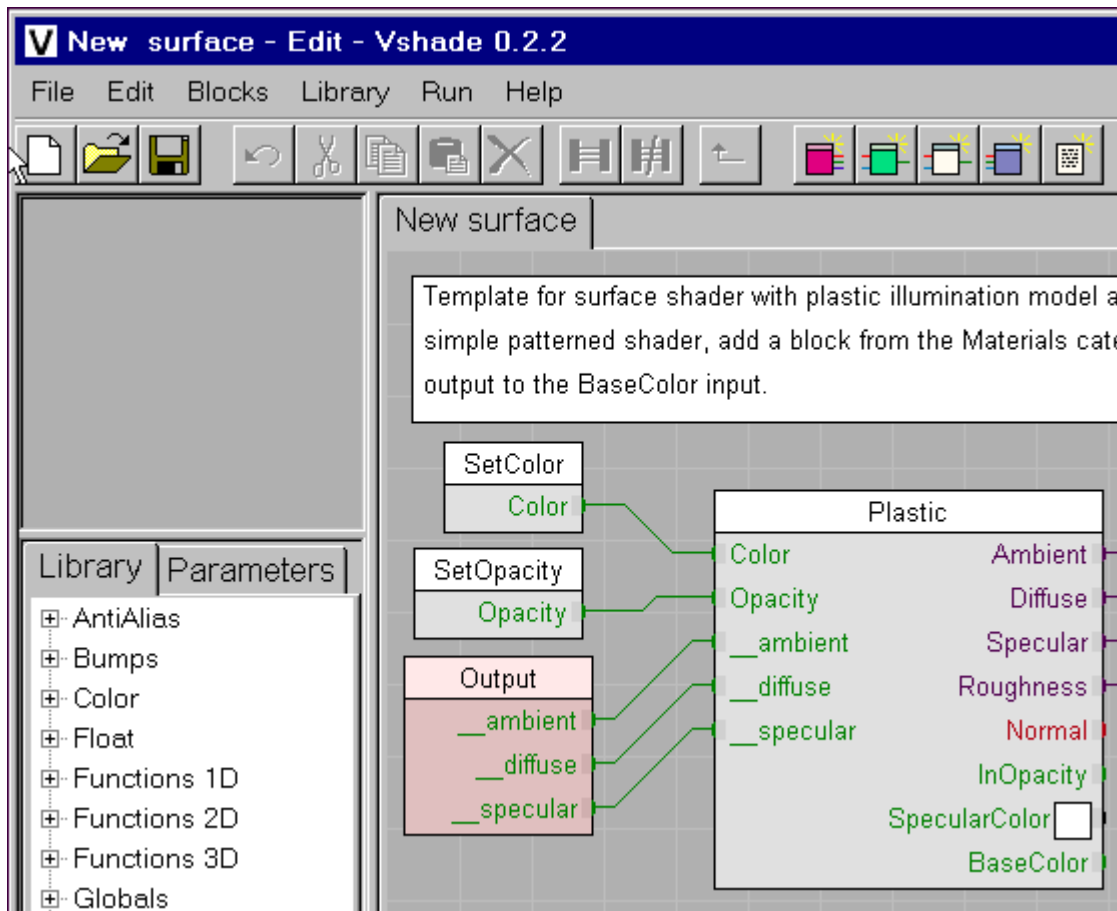
There is a toolbar along the top with buttons for common operations. If you hold the mouse pointer
over a button, a hint will appear with the name of the button's function. There are equivalent menu
items for all buttons.

In the upper left-hand corner is a blank square where a preview of the shader can be displayed.

Below the preview area is the library of blocks used to build shaders. The library is divided into
categories. To view the blocks in a category, click on the ⊞ to expand it.

The large area in the lower right is the shader construction area where you will build a shader.

Vshade automatically loads a shader template when it starts. This template is for a simple surface shader with a plastic illumination model. We could use this template to build our shader, but we will instead use a different template so we can learn how to start a new shader.

- In the **File** menu select **New** or click on the **New** button in the toolbar.

A dialog box will appear with a list of templates for common types of shaders.

- Scroll down and select the `Surface_ShinyPlastic` template. Click **OK**.

Vshade will open a new window using the selected template to begin a new shader. This template is for a surface shader with reflections. The template is a complete shader in itself.

- To see what the unmodified template shader looks like, click the Render button . A file dialog will appear prompting you to first save the shader. Save the shader as `shinygranite.vsl` in a convenient directory.

After a few seconds a picture of a sphere with our new shader should appear in the preview section of the main Vshade window. The sphere has a plastic illumination model and reflections.

Now we will turn this into a granite shader by adding a block from the _**Materials**_ category in the library.

- In the library section of the Vshade window expand the _**Materials**_ category by clicking on the ⊞ next to it.
- Select **Granite** in the _**Materials**_ category.
- Click in the middle of the shader construction area to position the _**Granite**_ block so that you can connect its Color output to the BaseColor input of the **ShinyPlastic** block. As long as the blocks are selected, you can click and drag in a block title bar to move the selected blocks.
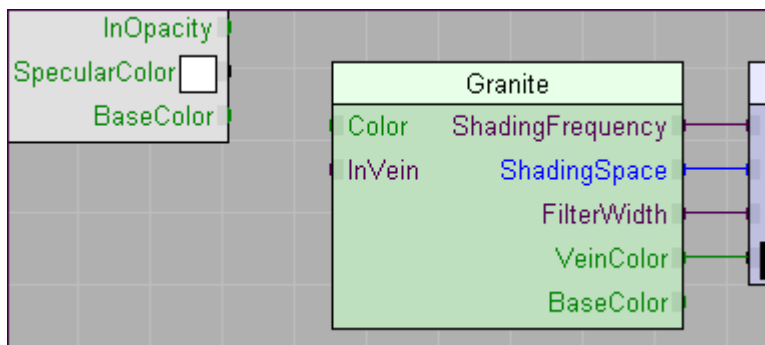
- Click on the BaseColor input to the **ShinyPlastic** block near the block edge, then on the Color output from the **Granite** block near its edge to connect to the two blocks:
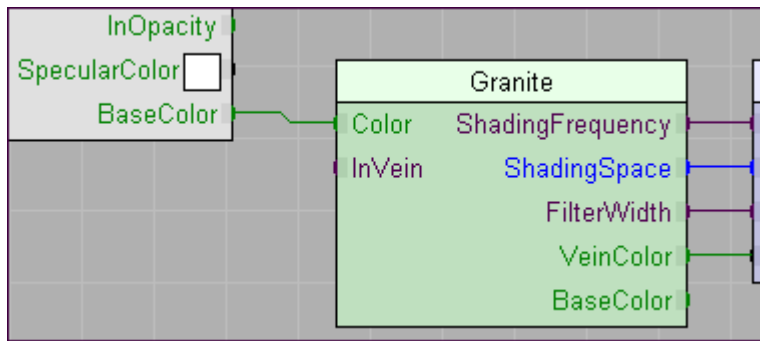
- Click the render button again to see the updated shader with a granite pattern.

You can produce other effects by connecting the Color or InVein outputs from the **Granite** block to the SpecularColor and InOpacity inputs to the **ShinyPlastic** block.

That's all there is to making a simple surface shader. Try using other Surface templates and materials in the *Materials* category to make other interesting shaders.

5.2   A First Displacement Shader

This brief tutorial shows how to make a simple displacement shader using the `Displacement` template and the blocks in the *Bumps* category.

- Start Vshade if it is not already running and select **New** from the **File** menu.
- Select the `Displacement` template to start a displacement shader.
- In the library section open the *Bumps* category and select the **Cells** block.
- Click in the shader construction area and position the **Cells** block so that you can connect the Bump output from the **Cells** block to the Bump input of the **DisplaceN** block.
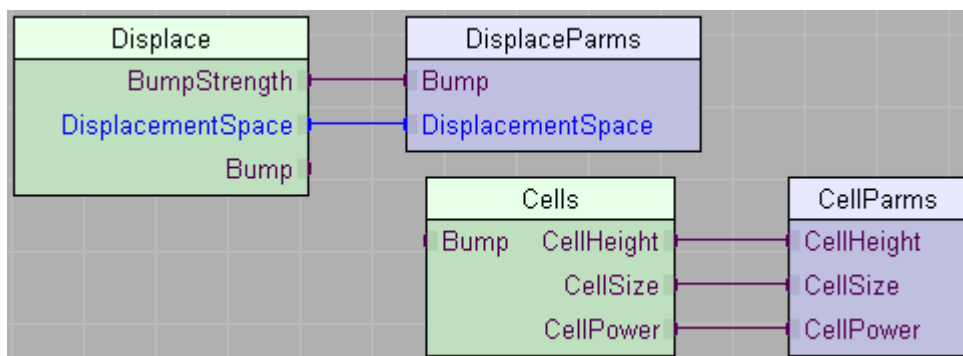
- Click on the Bump input to the **DisplaceN** block near the edge, then on the Bump output for the **Cells** block near the edge to connect the two blocks.
- Click the **Save** button and save the shader as `cellbumps.vsl` in a convenient directory.
- Click the **Render** button to see a preview of the shader.

This same procedure can be used to build a displacement shader from any of the blocks in the *Bumps* category. The blocks in the *Patterns 1D*, *Patterns 2D*, and *Patterns 3D* categories can also easily be used for displacement.

5.3   A Surface Shader with Bumps

Vshade includes many surface templates to get you started, but sooner or later you'll need to build a shader that isn't based on one of the templates. This tutorial shows how to build a surface shader using one of the illumination models provided with Vshade and how to add bump mapping to any surface shader.

- In Vshade select **New** from the **File** menu and choose `Surface` as the template to load. This loads a blank surface shader.
- Click the **Save** button and save the shader as `GlossyCells.vsl` in a convenient location.
- In the component library window expand the *Lighting Models* category and select the **ShinyCeramic** component.
- Click in the shader construction area and place the **ShinyCeramic** component as illustrated:
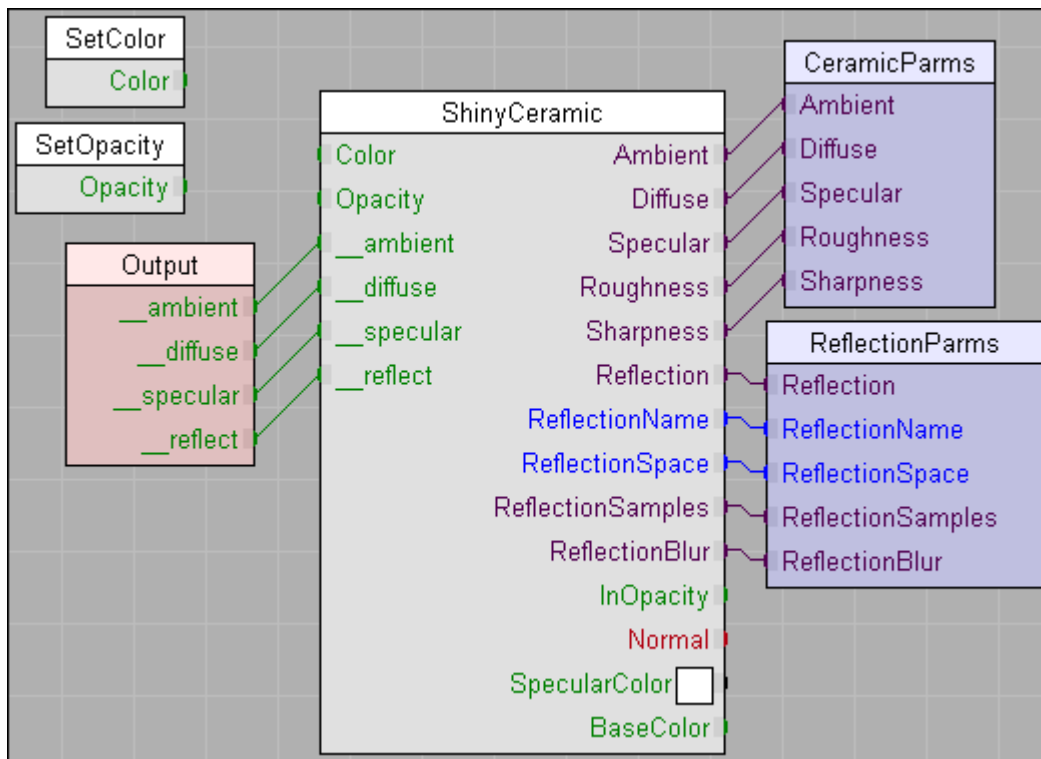
Next, we'll connect the color and opacity outputs from the **ShinyCeramic** block to the blocks that set the opacity and color for the surface shader.

- Connect the Color output from the **ShinyCeramic** block to the Color input of the **SetColor** block.
- Connect the Opacity output from the **ShinyCeramic** block to the Opacity input of the **SetOpacity** block.
- Click the **Render** button to see the basic shiny ceramic shader.

Now we'll add some cells to the shader.

- Select the **AddBump** component in the *Lighting Models* category.
- Place the **AddBump** component just to the right of the Normal input to the **ShinyCeramic** block.
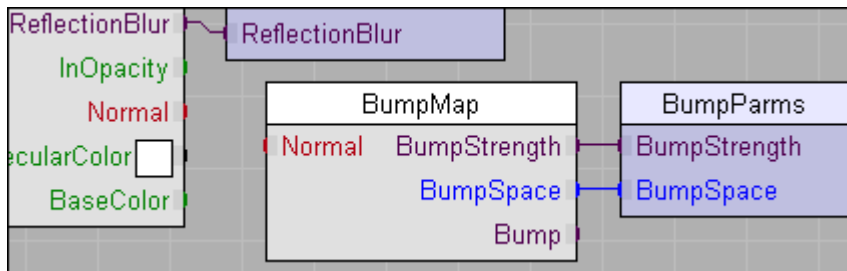
- Connect the Normal output from the **BumpMap** block to the Normal input to the **ShinyCeramic** block. Now the **ShinyCeramic** block will use the bump-mapped normal for its shading calculations.

FInally, we need to add a cells pattern to use as the bump map.

- Expand the *Bumps* category and select the **Cells** component.
- Place the **Cells** component to the right and below the **BumpMap** block.
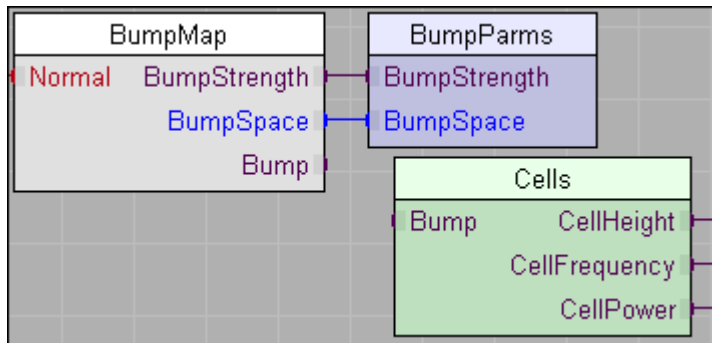
- Connect the Bump output from the **Cells** block to the Bump input of the **BumpMap** block.
- Click the **Render** button to see the final shader.

You can use this same procedure to combine any of the illumination models provided with Vshade with any of the Bump patterns to form new shaders.

5.4   A Surface Shader with Texture Maps

Texture maps are a common means of adding pattern and variation to an object. With *AIR*'s programmable shaders texture maps can be used to control almost any aspect of a surface's appearance. This tutorial shows how to build a surface shader with texture maps.

- Start Vshade.
- Click the **New** button to begin a new shader and choose the `Surface_Plastic` template.
- Click the **Save** button and save the shader as `MyTexturedPlastic.vsl` in a convenient directory.
- Click the **Render** button to see a preview of the base shader. You should see a shiny blue sphere.

Texture Map for Base Color

We will now add a texture map to control the base object color.

- In the component library open the *Materials* category and select the **TextureMap** component.
- Place the **TextureMap** component in the shader construction area and position it to right and slightly below the **Plastic** block.
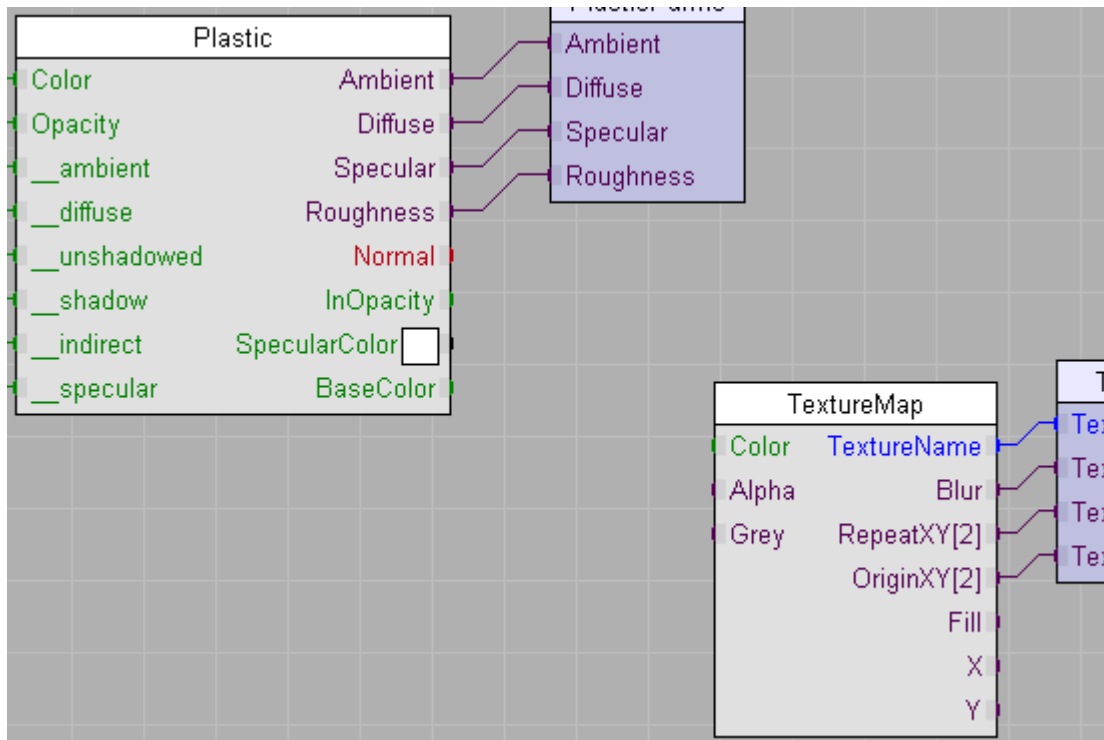
- Connect the <u>Color</u> output of the **TextureMap** block to the <u>BaseColor</u> input of the **Plastic** block.
- Render a preview image again to see the new shader.

The formerly blue sphere now appears white. Here's what happened: by default the <u>BaseColor</u> is taken from the object's color attribute, which is set to blue by default in Vshade. After we connected the **TextureMap** block, the <u>BaseColor</u> came from the texture map. Since we specified no texture map, the **TextureMap** block returns the <u>Fill</u> value, which is 1, producing white.

If no texture is provided, we would like the shader to have the same appearance it had before the **TextureMap** block was attached. We can accomplish this by multiplying the <u>Color</u> output of the **TextureMap** block by the object's color value.

- Open the *Globals* category in the component library and select the **ColorIn** component.
- Place a <u>ColorIn</u> component above the **TextureMap** block in the shader construction area.
- Open the *Color* category and select the **Multiply** block.
- Place a **Multiply** block to the left of the **TextureMap** block.
- Connect the **Multiply** block as depicted in the following snapshot:
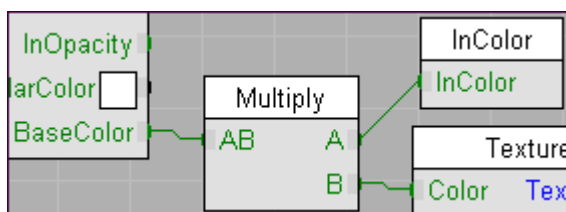
- Render the preview image again. You should once again see a blue sphere.

- To see what the shader looks like with a texture map, double-click the <u>TextureName</u> input parameter to display the **Edit Parameter** dialog.

- Change the **Value** to `grid.tx`. (`grid.tx` is a texture map included with **_AIR_**).
- Render a preview again to see the texture-mapped sphere.

- Note that it has a blue-ish tinge. That's because the texture map result is multiplied by the blue object color. To remove the blue tint, select **Settings** in the **Run** menu to display the **Preview Settings** dialog. Change the **Color** value to `1 1 1` (white). Close the dialog and re-render.

Texture Map for Specular

Now we will add an additional texture map that controls the specular color.

We will need a little more room near the **Plastic** block, so move the **Multiply**, **InColor**, **TextureMap**, and **TextureParms** blocks a couple inches to the right and down. (Tip: You can easily select all 4 blocks at once by dragging a rectangle from right to left that touches or encloses all 4 blocks.)

- In the **_Materials_** category select the **TextureMap** component.
- Place a new **TextureMap** component above the **InColor** block. Your shader should look something like this:
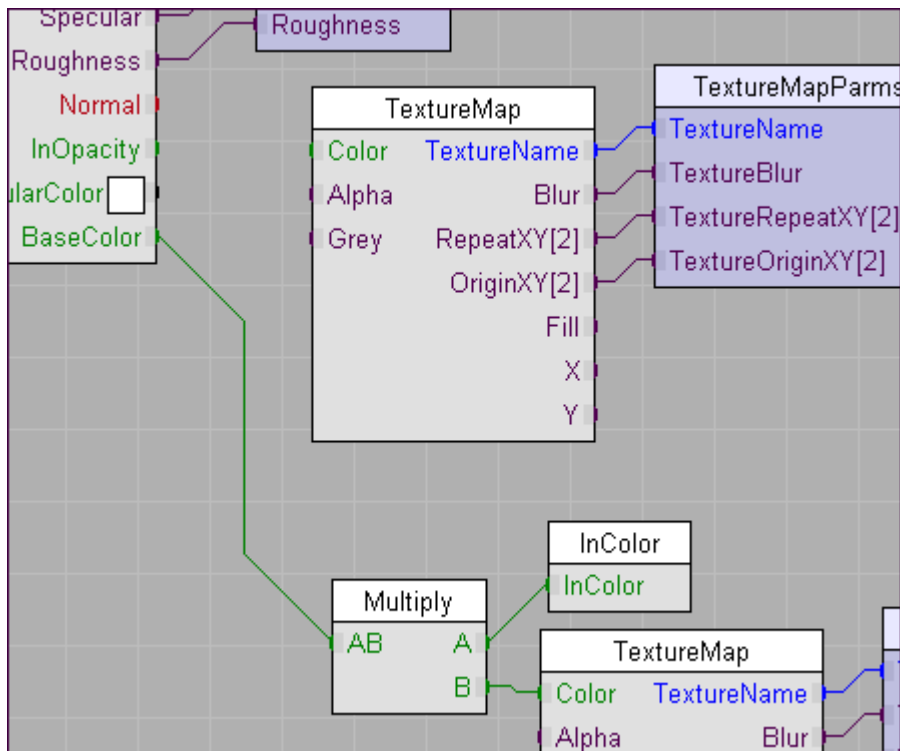
- Connect the Color output from the new **TextureMap** block to the SpecularColor input to the **Plastic** block.
- Click the **Render** button.

Vshade displays a warning message that there is a duplicate parameter in the shader: TextureName. Every shader parameter must have a unique name. Vshade provides an easy means of renaming all the parameters in an input or output block.

- Select the top **TextureParms** block (the one connected to the **TextureMap** block connected to SpecularColor).
- In the **Edit** menu select **Prefix**. This allows us to change the prefix assigned to these parameters

from `Texture` to something else. Change the prefix to `Specular` and click **OK**.
- Now select the other **TextureParms** block. Choose **Prefix** in the **Edit** menu and change this prefix to `Color`. Click **OK**.
- Click the **Render** button to preview the shader.

- To see the effect of a specular map, double-click the <u>SpecularName</u> parameter and set its value to `grid.tx`.
- Render the preview again to see the preview image with a specular map.

Using these same techniques you can add texture maps to control almost any aspect of a surface appearance. E.g., you could add another map to control the opacity. For templates that support bump mapping, another map can be used for the bump value.

5.5   Discrete Color Patterns: Simple Brick

Vshade contains many components that generate patterns as floating-point values that vary between 0 and 1. Those patterns can be used with the color **Mix** component to make simple color patterns. This tutorial shows how to make a simple colored brick and mortar pattern and how to vary the color of each brick.

- Start a new shader and select the **Surface_Matte** template.
- Click the **Save** button and save the shader as `SimpleBrick.vsl` in a convenient directory.
- Open the *Color* category in the component library and select the **Mix** component.
- Place the **Mix** component to the right of the **Matte** block.
- Connect the <u>Mix</u> output to the <u>BaseColor</u> input of the **Matte** block.
- Open the *Patterns 2D* component category and select the **BrickPattern** component.
- Place the **BrickPattern** component to the right and below the **Mix** block.
- Connect the <u>InBrick</u> output of the **BrickPattern** block to the <u>Fraction</u> input of the **MixColors** block.
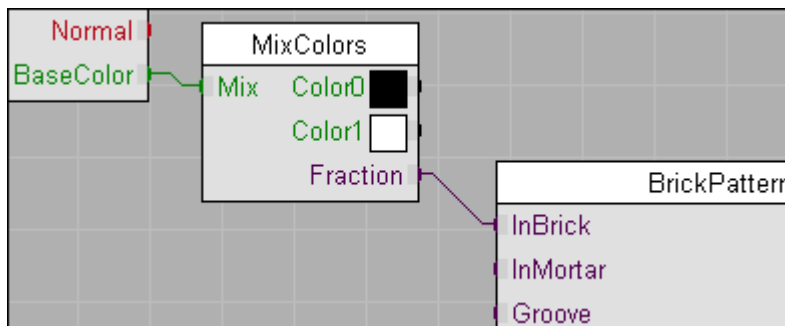
- Click the **Render** button to view the shader.

The **MixColors** block blends between two colors based on the <u>Fraction</u> input value. For the simple brick pattern, we will use the surface color attribute for the brick color and provide a shader parameter for the mortar color.

- In the *Globals* category select the **ColorIn** block.
- Place the **ColorIn** block to the right of the **MixColors** blocks.
- Connect the <u>ColorIn</u> output to the <u>Color1</u> input of the **MixColors** blocks.

- In the **Blocks** menu select **New Input** or click the New Input button in the toolbar.
- In the **Input Parameters** dialag click **Add** to add a new parameter.
- Set the parameter **Name** to `MortarColor`

- Set the **Type** to `color`.
- Set the default **Value** to `0.3 0.3 0.3`
- Click **OK** to close the **Add** dialog.
- Click **OK** to close the **Input Parameters** dialog.
- Move the new parameter block so it is above the **ColorIn** block.
- Connect the MortarColor output from the **Parameters** block to the Color0 input of the **MixColors** block.
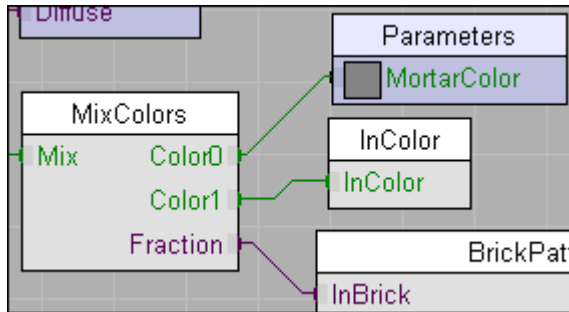
- Render a preview to see the new shader.

The bricks are colored blue (based on the color attribute), and the mortar is grey.

Varying Brick Color

We can make the pattern more interesting by varying the color of each brick.

- We will need some more space between the **BrickPattern** block and the **MixColors** block. Move the **BrickParms** and **BrickPatterns** blocks about 2 inches or 4 centimeters to the right.
- In the *Color* category select the **VaryItemColor** component.
- Place the **VaryItemColor** component between the **MixColors** block and the **BrickPattern** block.
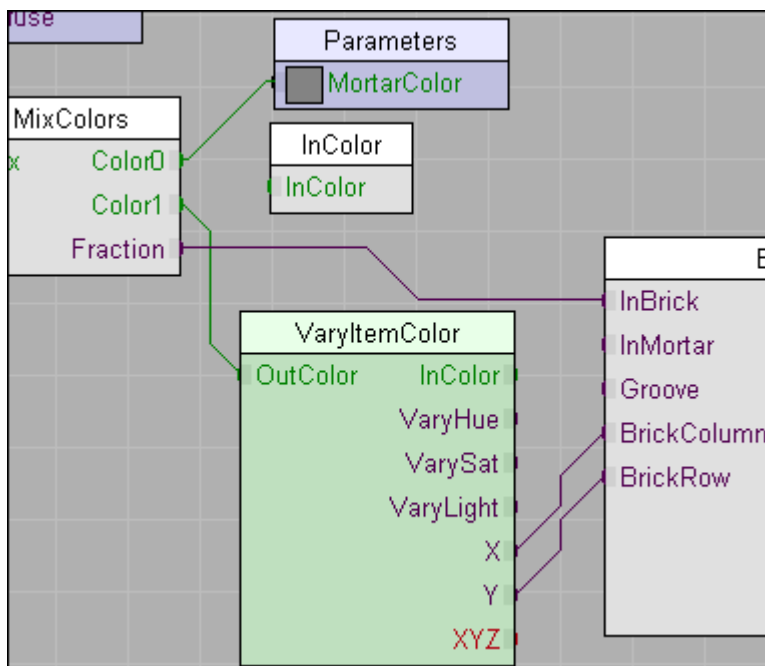
- Connect the <u>OutColor</u> parameter of the **VaryItemColor** block to the <u>Color1</u> input of the **MixColors** block.
- Connect the <u>BrickColumn</u> output of the **BrickPattern** block to the <u>X</u> input of the **VaryItemColor** block.
- Connect the <u>BrickRow</u> output of the **BrickPattern** block to the <u>Y</u> input of the **VaryItemColor** block.
- By default the **VaryItemColor** block takes its color input from the surface color attribute, so we don't need the **InColor** block anymore. Select it and delete it.

The **VaryItemColor** block varies its input color randomly based on the <u>X</u>, <u>Y</u>, and <u>XYZ</u> item indices. Vshade blocks that produce patterns of discrete elements like bricks, tiles, or cells provide output parameters identifying the particular element being shaded that can be used to vary shading values for each item. In this case we use the brick row and column to vary the brick color.

- Double click the <u>VaryLight</u> parameter of the **VaryItemColor** block and change its value to 0.1
- Render a preview.

Each brick now has a slightly different color. Try experimenting with the <u>VaryHue</u> and <u>VarySat</u> parameters to see their effect. You may wish to create shader parameters to allow the color variation to be set by a user.

5.6   Continuous Color Patterns: Simple Marble

This tutorial shows how to use continous function or pattern to make a colored pattern and how to tweak the blending between colors.

- Start a new shader and select the **`Surface_Plastic`** template.
- Save the shader as `SimpleMarble.vsl` in a convenient directory.
- In the *Color* category select the Mix component.
- Place the **Mix** component to the right of the **Plastic** block.
- Connect the <u>Mix</u> output to the <u>BaseColor</u> input of the **Plastic** block.
- In the *Patterns 3D* category select the **MarbleVeins** component.
- Position the **MarbleVeins** component to the right of the **MixColors** block.
- Connect the <u>InMarble</u> output from the **Marble** block to the <u>Fraction</u> input of the **MixColors** block.
- Select the **ShadingPosition** component in the *Transformations* category.
- Place the **ShadingPosition** component to the right of the **Marble** block.
- Connect the <u>ToP</u> output from the **TransformPoint** block to the <u>Position</u> input of the **Marble** block.
- Click the **Render** button to render a preview image.

We have a simple marble shader that mixes between black and white. Next we add user control over the colors. We'll use the surface color attribute for the base color and create a shader parameter for the vein color.

- In the *Globals* category select the **ColorIn** component.
- Place the **ColorIn** component above the **Marble** block.
- Connect the <u>ColorIn</u> output to the <u>Color1</u> input of the **MixColors** block.
- To create an input block for the remaining color parameter, select the **MixColors** block and choose **Block for Inputs** from the **Blocks** menu.
- Double-click the title bar of the new input block and edit the parameter to change it's name to `VeinColor`. Close the dialog.
- Render a new preview.

Users can now set the marble and vein colors. We can provide users more control over the transition between the two using the **TweakRange** component.

- To make room for the new component, move the **Marble**, **TransformPoint**, and **ShadingParms** blocks about 2 inches or 4 centimeters to the right.
- In the *Transitions* category select the **TweakRange** component.
- Place the **TweakRange** component between the **Marble** block and the **MixColors** block.
- Connect the Out parameter of the **TweakRange** component to the Fraction input of the **MixColors** block.
- Connect the InMarble output from the **Marble** block to the In input of the **TweakRange** block.
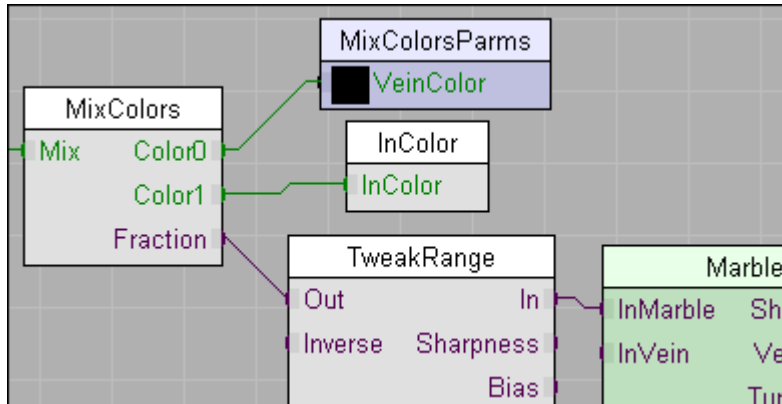
- In the **Run** menu select **Settings...** and change the preview scene to Square to better observe the effects of modifying the marble output.
- Render a preview image to see the shader without modifications.
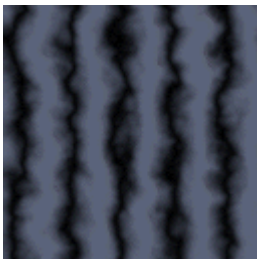
The **TweakRange** block has two basic parameters for modifying the input range - Sharpness and Bias. The Bias parameter can be used to shift the source signal towards the higher or lower end of the output range.

- Double-click the Bias parameter of the **TweakRange** block and change its value to 0.8
Preview the shader.
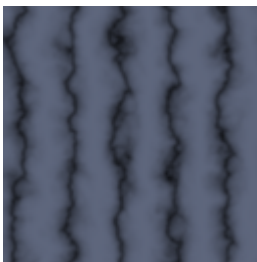
Increasing bias shifts the source signal towards the high end of the range, making the veins narrower.

- Change the Bias to 0.2 and render again.
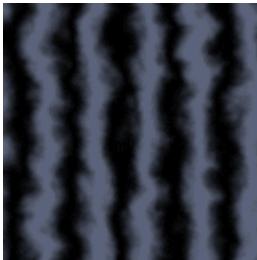
A bias less than 0.5 moves values towards the low end of the range, making the veins wider.

- Set the Bias back to the middle value 0.5 (no bias).
- Double-click the Sharpness parameter and set its **Value** to 0.7
- Preview the shader.
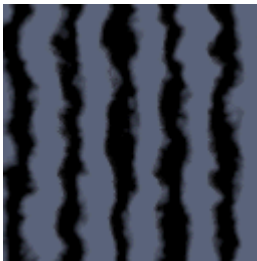
A higher Sharpness makes the transition between regions sharper.

- Change the Sharpness parameter to 0.3 and render the shader.
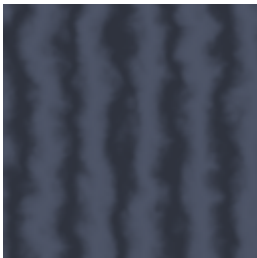
A Sharpness below 0.5 softens the transition between colors.

You can use the **TweakRange** block to modify any continuous function.

5.7   3D to 2D Projections

Every primitive has a default set of 2D texture coordinates that can be used to apply a 2D pattern to a surface. However, sometimes one wants to apply a 2D pattern that can't easily be fit onto the standard texture coordinates. A useful alternative is to generate 2D coordinates by converting or projecting a 3D point. Vshade provides an easy method of selecting various projections of a 3D point to use as 2D texture coordinates. This quick tutorial looks at the various projections and how to use them.

- Start a new shader and select the **Surface_Matte** template.
- Click the **Save** button and save the shader as ProjectionTest.vsl in a convenient directory.
- In the *Patterns 2D* category select the **TilePattern** component.
- Click in the shader construction area and place the **TilePattern** component just to the right of the

Matte block.

- Connect the <u>InTile</u> output from the **TilePattern** block to the <u>BaseColor</u> input of the **Matte** block.

The <u>X</u> and <u>Y</u> texture coordinate inputs to the **TilePattern** block default to the standard texture coordinates s and t.

- Render a shader preview to see the tile pattern with default texture coordinates.

- In the *Transformations* category select the **Project2D** component.
- Place the **Project2D** component just to the right and below the **TilePattern** block.
- Connect the <u>X</u> output of the **Project2D** block to the <u>X</u> input of the **TilePattern** block.
- Connect the <u>Y</u> output of the **Project2D** block to the <u>Y</u> input of the **TilePattern** block.
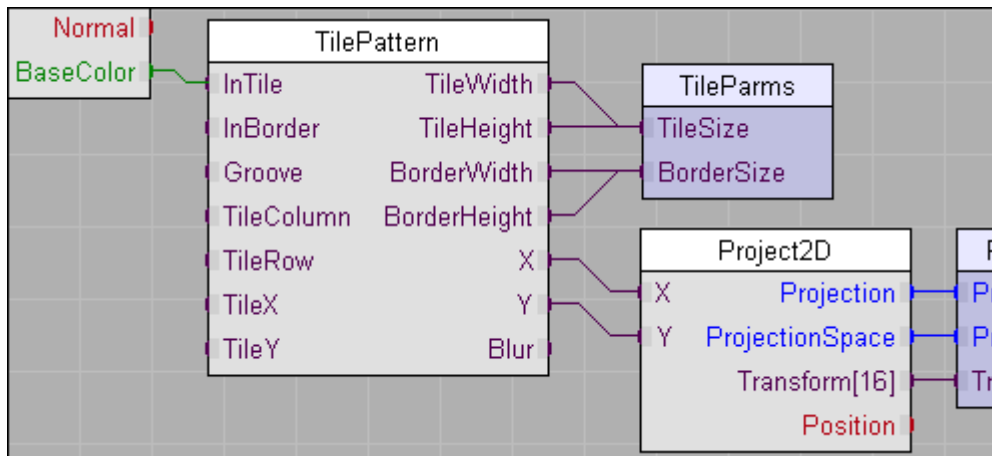
Now the **TilePattern** block will use the texture coordinates generated by the **Project2D** block. The **Project2D** block can perform five different 3D to 2D projections. We will look at them all.

"st" Projection

The default projection type is `"st"`, which just uses the standard 2D texture coordinates. If you render a preview of the updated shader, you will see the same image as the shader w/o the Project2D block. The `"st"` projection lets you easily offer users the choice of utilizing standard texture coordinates as well as any of the following 3D to 2D projection types.

"planar" Projection

- Double-click the <u>Projection</u> parameter of the **ProjectionParms** block and change its **Value** to `"planar"`.
- Preview the shader.
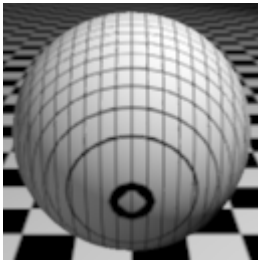
The planar projection uses the x and y components of the 3D point. Planar projections are useful for uniformly texturing fairly flat surfaces that may be constructed from multiple primitives.

"spherical" Projection

- Double-click the Projection parameter of the **ProjectionParms** block and change its **Value** to `"spherical"`.
- Preview the shader.

The spherical projection projects the position onto a unit sphere centered at the origin and takes the X and Y coordinates from the latitude and longitude of the sphere.

"cylindrical" Projection

- Change the Projection parameter of the **ProjectionParms** block to `"cylindrical"` and preview the shader.
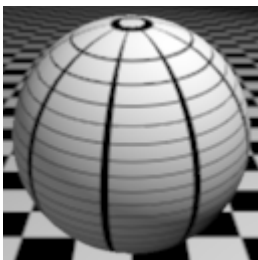
The cylindrical projection projects the point onto a cylinder around the Z-axis. A cylindrical projection is useful for texturing objects generated from a surface of revolution or similar to one.

"box" Projection

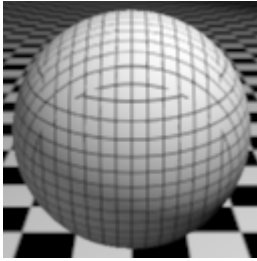- Change the Projection parameter of the **ProjectionParms** block to `"box"` and preview the shader.

The box projection projects the point onto the XY, YZ, or XZ plane depending on the dominant axis of the surface normal. A box projection is useful for texturing an object made of several orthogonal surfaces such as building.

- In the Run menu select **Settings** and change the preview file to `Box`.
- Re-render to see the box projection applied to a box.

5.8 Reflections with Angle Blend

This tutorial shows how to add reflections to the standard plastic shader using the **AngleBlend** component. This provides an alternative shiny plastic shader to the one included with Vshade. Plastics have the property that they are more reflective at glancing angle than when seen headon. The shiny plastic shader included with Vshade uses a fresnel function that is based on real-world physics to control the falloff of reflections. The **AngleBlend** block computes a function simply based on the angle between the surface normal and the incident viewing direction.

- Start a new shader by clicking the **New** button and selecting the **Surface_Plastic** template.
- Click the Save button and save the shader as `ShinyPlasticFalloff.vsl` in a convenient directory.
- We will need some space between the **Plastic** blocks and the output parameters. Select the **Plastic** and **PlasticParms** blocks by dragging a rectangle from right to left that crosses both. Then click in the title bar of one of the blocks and move both to the right about 2 inches.
- Open the *Reflections* category in the library and select the **Reflections** component. Click in the shader construction area and place a **Reflections** component below the **Plastic** block.
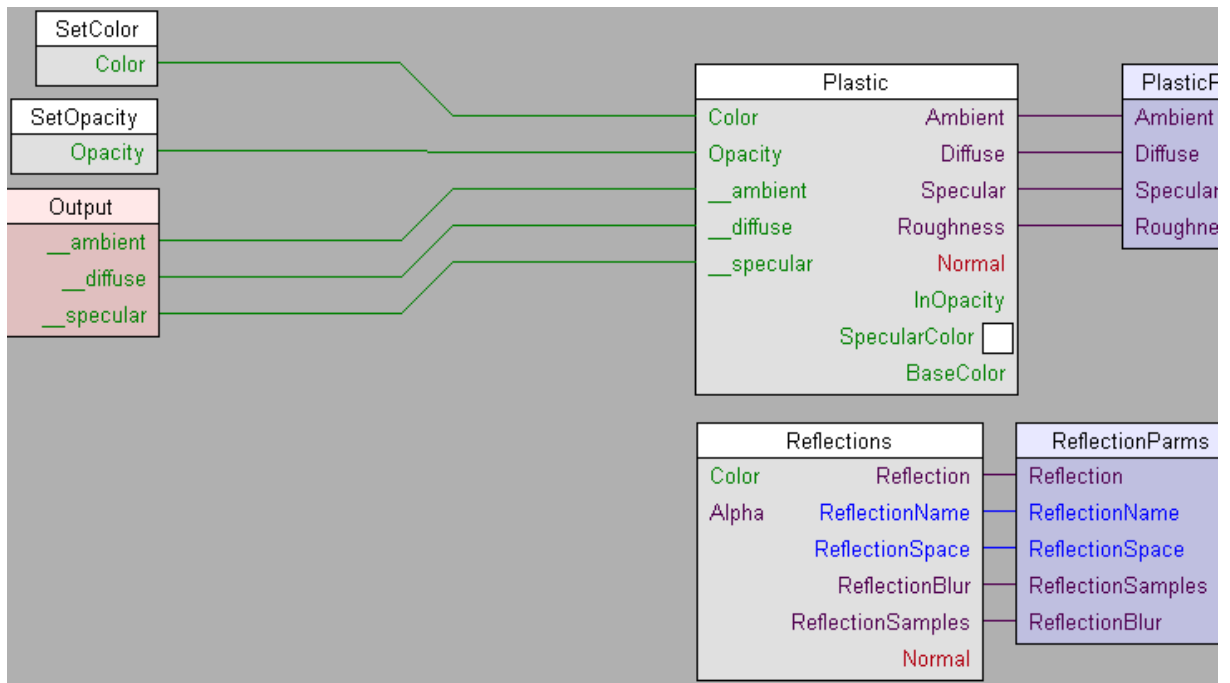
- Open the *PointFunctions* category in the component library and select the **AngleBlend** component. Place the component underneath the **Reflections** block in the shader construction area.

We want to multiply the reflected color by the Edge output from the **AngleBlend** block and add the result to the output color for the shader.

- Open the *Color* category and select the **Multiply** component. Place a **Multiply** block next to the **Reflections** block. Connect the Color output from the **Reflections** block to the top input to the Multiply block. Connect the Edge output from the **AngleBlend** block to the second input to the **Multiply** block.
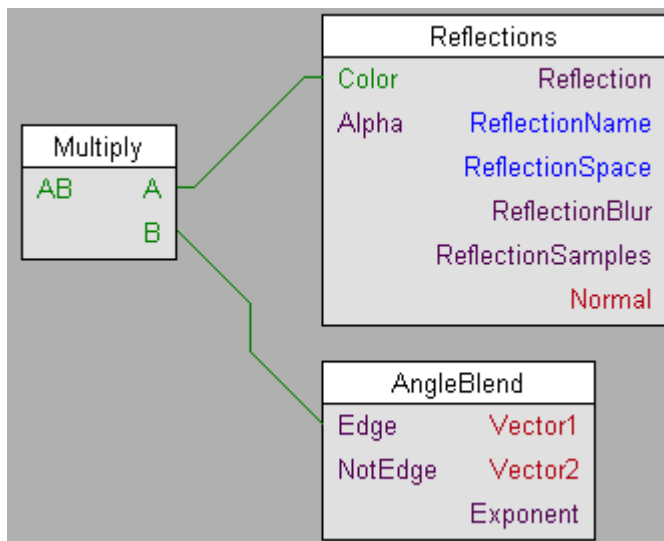
- In the *Color* category of the library select the **Add** component. Place the **Add** component to the left and above the **Multiply** component. Connect the output from the **Multiply** component to one input

of the **Add** component, and connect the <u>Color</u> output from the **Plastic** block to the other **Add** input. Finally connect the **Add** output to the **SetColor** input.
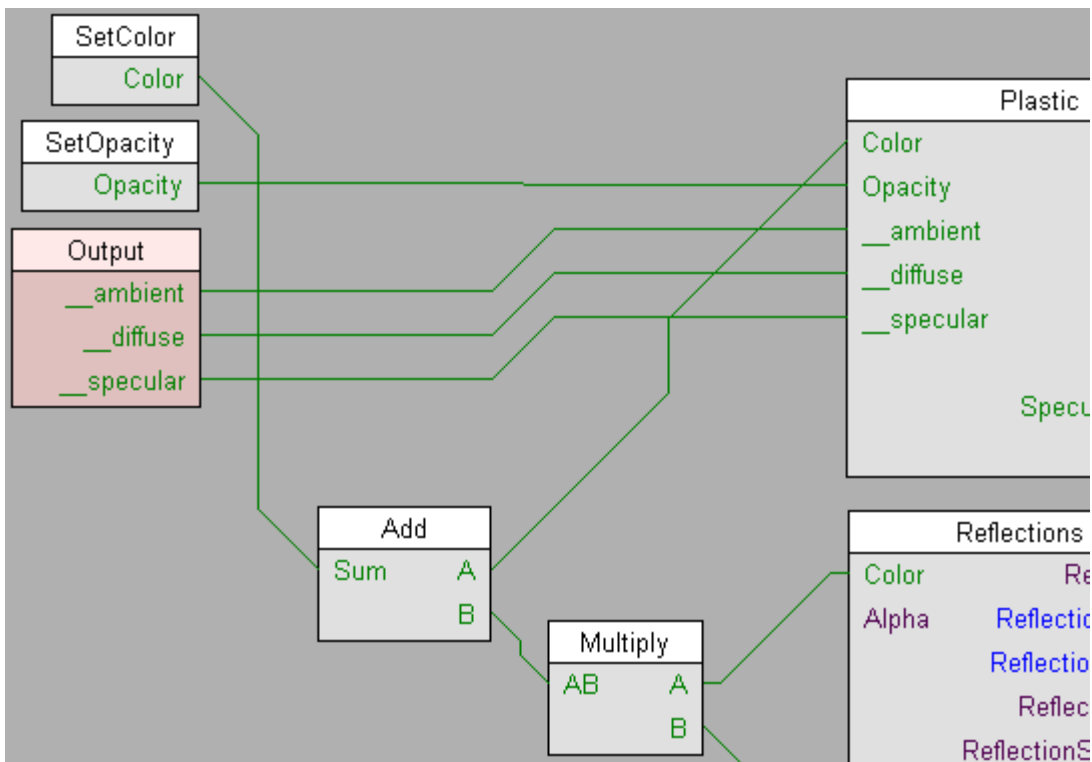
- Click the Render button to see the shader with reflections.

Adding a shader parameter that controls the exponent to the **AngleBlend** function will allow a user to control the extent to which reflections are affected by the viewing angle.

- Click the New Input block button in the toolbar to create a new block for input parameters to the shader.
- In the Input Parameters dialog click the **Add** button to add a new parameter.
- Name the parameter `ReflectionFalloff`
- In the Help field type `falloff of reflection with angle`
- Leave the type as float. Change the default value to 2.
- Click **OK** to close the Add Parameter dialog, then click **OK** again to close the Input Parameters dialog.
- Position the new **Parameters** block to the right of the **AngleBlend** block.
- Connect the <u>ReflectionFalloff</u> parameter to the <u>Exponent</u> input of the **AngleBlend** block.
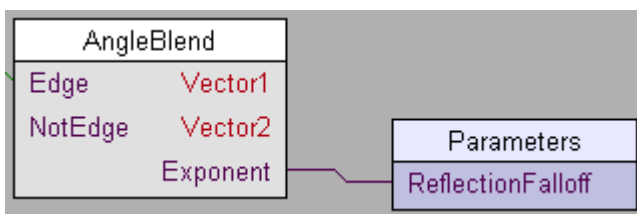
- Try rendering the shader with different values of <u>ReflectionFalloff</u> to see its effect. You can display a dialog for changing the <u>ReflectionFalloff</u> value by double-clicking the parameter name near the block edge.

5.9   A Rippled Displacement Shader

This tutorial shows how to build a displacement shader based on a 3D pattern. In this case we will use a noise pattern called Brownian that simulates the effects of pseudo-random motion. Here's what the Brownian pattern looks like:
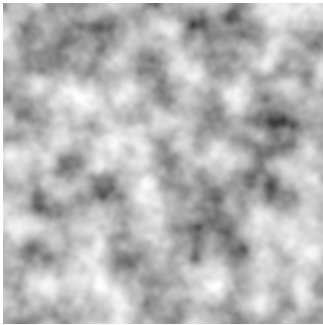
We'll use this to produce a ripple-like appearance.

- Click the New button and select the **Displacement** template.
- Click the Save button and save the new shader as MyRipples.vsl in a convenient directory.

Brownian computes a pattern based on a 3D point. Now, we can't just use the input position for that point, because that point is in "current" space which will change, for example, if the camera position changes. Instead, we want to convert the surface location to a stable coordinate space such as the "shader" space in which the displacement shader is declared. For stable 3D patterns, remember to always use the **ShadingPosition** component in the *Transformations* category as the starting point for a 3D pattern.

- Expand the *Noise* category and select the **Brownian** component.
- Place the **Brownian** component in the shader below the **DisplaceParms** block.
- Connect the <u>fbm</u> output from the **Brownian** block to the <u>Bump</u> input of the **DisplaceN** block.
- Expand the *Transformations* category and select the **ShadingPosition** block.
- Place a **ShadingPosition** component to the right of the **Brownian** block.
- Connect the <u>ToP</u> output from the **TransformPoint** block to the <u>Position</u> input to the **Brownian** block.

The shader should look approximately like this:
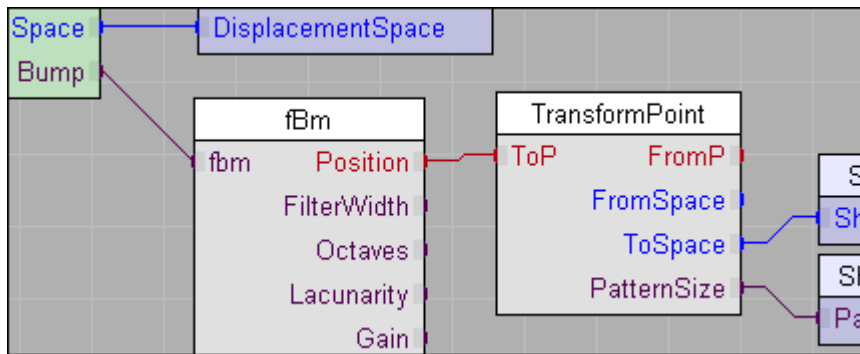
- Click the **Render** button to see the shader.

Notice the sharp contrast in the bump features. We would like a subtler effect.

- Double-click the Bump parameter in the **DisplaceParms** block and change its Value to 0.1
- Render again to see the more subtle effect.

We might like to give users more control over the shader by allowing them to set the input parameters to the Brownian function. We can easily do this as follows:

- Select the **Brownian** block.
- In the **Blocks** menu select **Block for Inputs**.

Vshade automatically creates a parameter block for all the unconnected inputs to the **Brownian** block.

5.10   A Wood Shader with Decals

This tutorial shows how to create a simple wood shader with 2 optional texture decals.

- Start Vshade.
- Click the **New** button to begin a new shader and choose the `Surface_Plastic` template.
- Click the **Save** button and save the shader as `Decal2DWood.vsl` in a convenient directory.
- In the Library window expand the *Materials* category and select the **Wood** component.
- Place the **Wood** component to the right and slightly below the **Plastic** block.
- Connect the Color output of the **Wood** block to the BaseColor input of the **Plastic** block.
- Render a preview to see the base wood shader.

Adding the First Decal

Now we'll add a couple decals to the base shader. The *Layers* category contains two decal components - **Decal2D** and **Decal3D**. **Decal2D** positions the decal using standard texture coordinates. **Decal3D** positions a decal using a projection. For this tutorial we will use the **Decal2D** component. This same procedure can be used with the **Decal3D** component to add projected decals to a shader.

The **Decal2D** components works by applying a decal on top of a base color. In our case, the base color is the output color of the **Wood** block. We'll need to place the **Decal2D** block between the **Wood** block and the **Plastic** block.

- Select the **Wood** and **WoodParms** blocks and move them to the right about 4 squares and down about 3 squares.
- Select the **Decal2D** block in the *Layers* category.

- Place the **Decal2D** block between the **Plastic** block and the **Wood** block.
- Connect the Color output of the **Wood** block to the BaseColor input of the **Decal2D** block.
- Connect the Color output of the **Decal2D** block to the BaseColor input of the **Plastic** block.
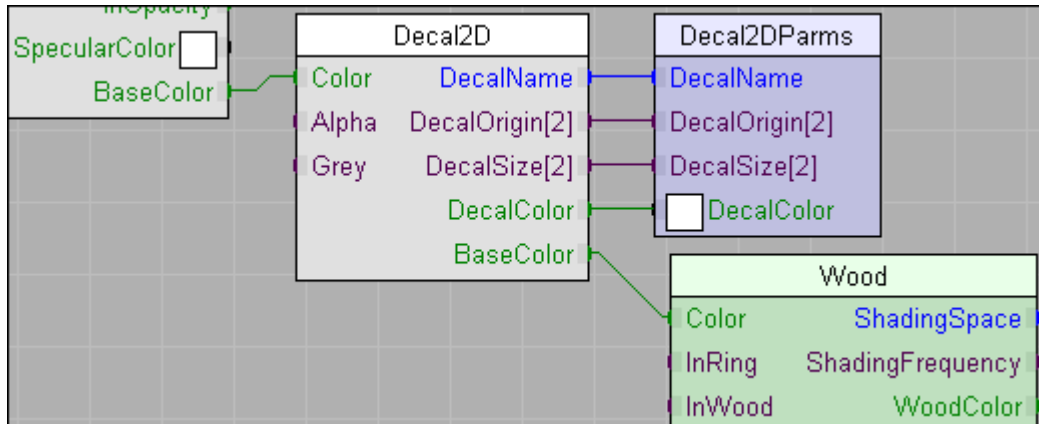
To see the result:

- Select the View tab, and change the preview Scene to Square in the list of files.
- Double-click the DecalName parameter, and set the Value to `sitex.tx`, a simple decal included with AIR.
- Render a preview, which should look something like:

with the decal applied over the wood.

Adding a Second Decal

Adding another decal is pretty much like adding the first: we'll insert another **Decal2D** block between the existing **Decal2D** block and the **Plastic** block. The only catch is that we'll need to rename the decal parameters to avoid conflicting parameter names.

- Select the **Decal2D**, **Decal2DParms**, **Wood**, and **WoodParms** blocks. Move them all 5 squares to the right and 3 squares down.
- Add a **Decals2D** block from the *Materials* category, placing it to the right of the Plastic block in the space you just created.
- Connect the Color output of the new **Decal2D** block to the BaseColor input of the **Plastic** block.
- Connect the Color output from the rightmost **Decal2D** block to the BaseColor of the new **Decal2D** block.
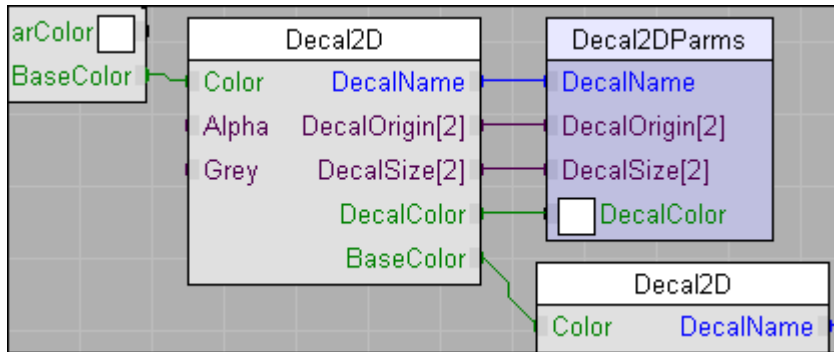
- Select the new **Decal2DParms** block. From the **Edit** menu choose **Prefix**. Change the prefix for all the parameters in that block from `Decal` to `SecondDecal`.

To see the second decal:

- Change the <u>SecondDecalName</u> to sitex.tx
- Change the <u>SecondDecalOrigin</u> to 0.3 0.3
- Set the <u>SecondDecalSize</u> to 0.5 0.5
- Change the <u>SecondDecalColor</u> to 1 0 0
- Render a preview to see a second, smaller decal on top of the first:

6   Reference

[3D to 2D Projections](#)

[Coordinate Spaces](#)

[Blend Modes](#)

6.1   3D to 2D Projections

Components included with Vshade that support 3D to 2D projections support the following projection types:

`st`	Use the default s,t texture coordinates
`planar`	Use the x and y components of the point
`box`	Project the point onto a plane orthogonal to the x,y, or z axes based on the largest component of the surface normal. This projection is useful for walls, boxes, and other shapes with flat sides that are perpendicular to a coordinate axis.
`spherical`	Project the point onto a sphere centered at the origin with the poles on the z-axis. The "latitude" and "longitude" of the sphere are used as the texture coordinates.
`cylindrical`	Project the point onto a cylinder wrapped around the z axis. The first texture coordinate is the angle around the axis of the point (measured counter-clockwise from the x-z plane) scaled to the range 0-1. The second texture coordinate is the z component.

Projection Space

For projection types other than st, the coordinate system for the point used for projection may also often be chosen (just as the shading space for a solid texture is) using a ProjectionSpace or equivalent parameter.

Texture Coordinate Transformation

Shaders may also accept a transformation matrix to be applied to the point prior to projection. The transformation matrix can be used to scale, translate, rotate, or shear the texture pattern.

6.2   Coordinate Spaces

AIR supports hierarchical modeling, in which objects are positioned by applying a series of translation, rotation, scaling, and arbitrary 3D transformation operations. Each transformation defines a new coordinate system. AIR provides names for those systems or spaces that are commonly used for shading calculations:

`object`   Coordinate system in which a geometric primitive is defined.

`shader`   Coordinate system in which a shader is declared.

`world`   Coordinate system at WorldBegin, after the camera has been positioned and before any transformations for objects have been declared. (The base coordinate system for a scene.)

`camera`   Coordinate system of the camera. The camera is assumed to be at the origin, with the y axis pointing up, the z-axis pointing in, and the x-axis increasing to the right.

`current`   The space in which calculations are performed within shaders. For *AIR* current space is camera space, but current space may be different for other renderers.

`screen`   Coordinate space after perspective projection (with z coordinates scaled and offset to 0 at the near clipping plane and 1 at the far clipping plane.)

`NDC`   Normalized device coordinates. A 2D device-independent coordinate system defined on the screen, with x running from 0 to 1 left-to-right, and y increasing from 0 to 1 top-to-bottom.

`raster`   A 2D coordinate system where x and y are the pixel coordinates of a point after projection to the screen.

6.3   Blend Modes

Vshade's *Layers* and *Bump Layers* components recognize the following blend modes:

`mix`   blend between layers based on the top layer strength

`add`   add layers together

`subtract`   subtract layer values

`multiply`   multiply layer values

`min`   use the minimum of the layer values

`max`   use the maximum of the layer values

7   History

Vshade 13.0 (September 2013)

- Prefix and suffix commands now work with output blocks
- Rendering to Air Show now uses float precision without gamma correction so the pixel viewer can be used to see raw color values
- Rendering a link to Air Show no longer wraps output values to keep them in the range 0 to 1
- New item in the **Run** menu allows the preview image to include any output variables when rendering to Air Show
- Replaced the old Gamma menu item in the **Run** menu with a new option to enable sRGB conversion of the preview image
- New **Run** menu item to indicate textures are in sRGB color space (enabled by default)
- New preview options to add a sky environment and enable indirect illumination
- The standard templates have been updated to use the latest version of their respective shading models
- New surface shader templates for physical plastic and physical metal
- Updated the TextureMap and Texture3D blocks under Materials
- The Processors menu item in the **Run** menu has been removed. Preview rendering now uses all detected processors.
- The deprecated fast shading item in the **Run** menu has been removed.
- New HexPattern component under Patterns 2D
- Window width and height may now be specified using two environment variables, VSHADE_WIDTH and VSHADE_HEIGHT
- Under Linux Vshade handles a failure to load fonts more gracefully
- Changed version number to match current Air release

Vshade 5.0 (August 2012)

- New iterator block for simplified looping
- The parameter increment and decrement buttons now accept modifiers: holding down the shift key while clicking will increment or decrement by 10x the normal amount. Holding down the control key while clicking will increment or decrement by 0.1 times the normal amount.
- Vshade issues an error message if the shader could not be compiled because write permission was denied to the output directory
- (Windows) The default output directory for compiled shaders is now $HOMEPATH/SiTex/shaders. (The old default was $AIRHOME/usershaders, but that directory is not writable if AIR is installed under Program Files in Windows 7).
- (Windows) Drop a .rib file in the Vshade window to assign it as the preview scene
- (Windows) New check to prevent multiple parameter edit dialogs

Vshade 4.0 (July 2011)

- Support for new environment and generic shader types in Air 11
- The status bar now displays the last undo-able action
- New Query library category with components for querying attributes, options, and dictionary entries
- New ColorTransform component in the Color category
- File dialog support for PTEX files
- Many small bug fixes for the Linux binary

Vshade 3.0 (August 2009)

- Support for TweakAIR
- New, improved parameter dialog

- Color dialog on Linux
- Drag-and-drop support for .vsl files under Windows
- New toolbar option to display the shader graph flowing left-to-right instead of right-to-left

Vshade 2.0 (December 2008)

- Support for new instancer and procedure shader types in AIR 8
- New RIB blocks for use in instancer or procedure shaders
- Loops
- New Star and Polygon components in Shapes 2D
- New File menu options to export and import shader parameter values
- On Linux Vshade no longer depends on libglade, allowing it to run on more modern Linux versions
- Many small UI fixes and enhancements