OpenGL Shading Language Course

Chapter 1 – Introduction to GLSL

By
Jacobo Rodriguez Villar
jacobo.Rodriguez@typhoonlabs.com

CHAPTER 1: INTRODUCTION

INDEX

An Introduction to Programmable Hardware

Brief history of the OpenGL Programmable Hardware Pipeline

2000
Card(s) on the market: GeForce 2, Rage 128, WildCat, and Oxygen GVX1

These cards did not use any programmability within their pipeline. There were no vertex and pixel shaders or even texture shaders. The only programmatically think was the register combiners. Multi-texturing and additive blending were used to create clever effects and unique materials.

2001
Card(s) on the market: GeForce 3, Radeon 8500

With GeForce 3, NVIDIA introduced programmability into the vertex processing pipeline, allowing developers to write simple 'fixed-length' vertex programs using pseudo-assembler style code. Pixel processing was also improved with the texture shader, allowing more control over textures. ATI added similar functionality including some VS and FS extensions (EXT_vertex_shader and ATI_fragment_shader) Developers could now interpolate, modulate, replace, and decal between texture units, as well as extrapolate or combine with constant colors. They could perform some other basic pixel operations.

2002
Card(s) on the market: GeForce 4

NVIDIA's GeForce 4 series had great improvements in both the vertex and the pixel stages. It was now possible to write longer vertex programs, allowing the creation of more complex vertex shaders.

2003
Card(s) on the market: GeForce FX, Radeon 9700, and WildCat VP

The GeForce FX and Radeon 9700 cards introduced 'real' pixel and vertex shaders, which could use variable lengths and conditionals. Higher-level languages were also introduced around the same time, replacing the asm-based predecessors. All stages within the pixel and vertex pipeline were now fully programmable (with a few limitations).

3Dlabs shipped their WildCat VP cards, which allowed for 'true' vertex and fragment (pixel) shaders with loops and branching, even in fragment shaders. These were the first cards to fully support the OpenGL Shading Language (GLSL).

Until now, all vertex and pixel programming was done using a basic asm-based language called 'ARB_fp' (for fragment programs) or 'ARB_vp' (for vertex programs). Programs written in this language were linear, without any form of flow control or data structure. There were no sub-routines and no standard library (containing common functions). It basically processed arithmetic operations and texture access, and nothing more.

With the creation of GLSL, graphics cards could take advantage of a high level language for shaders. With a good compiler, loops and branches could be simulated within hardware that natively didn't support them. Many functions were also introduced, creating a standard library, and subroutines were added; GLSL pushed the hardware to its limits.

2004
Card(s) on the market: WildCat Realizm, GeForce 6, and ATI x800 cards

These cards are the latest generation of programmable graphics hardware. They support a higher subset of GLSL, including direct texture access from vertex shaders, large program support, hardware-based noise generation, variable-length arrays, indirect indexing, texture dependent reading, sub-routines, and a standard library for the most common functions (like dot, cross, normalise, sin, cos, tan, log, sqrt, length, reflect, refract, dFdx, dFdy, etc.). They can also use a long list of built-in variables to access many OpenGL states (like gl_LightSource[n].position, gl_TexCoord[n], gl_ModelViewMatrix, gl_ProjectionInverseMatrix, etc.). Data structures are supported as well through C-like structs.

Fixed Function vs. Programmable Function

Before programmable function pipelines, developers had to use the fixed function pipeline, which offered no magical vertex or pixel shaders.

The fixed vertex stage consisted of clip-space vertex computations, per-vertex normal, and all other common of per-vertex operations such as color material, texture coordinate generation, normal transformation, and normalisation.

The fixed fragment stage handled tasks such as interpolate values (colors and texture coordinates), texture access, texture application (environment mapping and cube mapping), fog, and all other per-fragment computations.

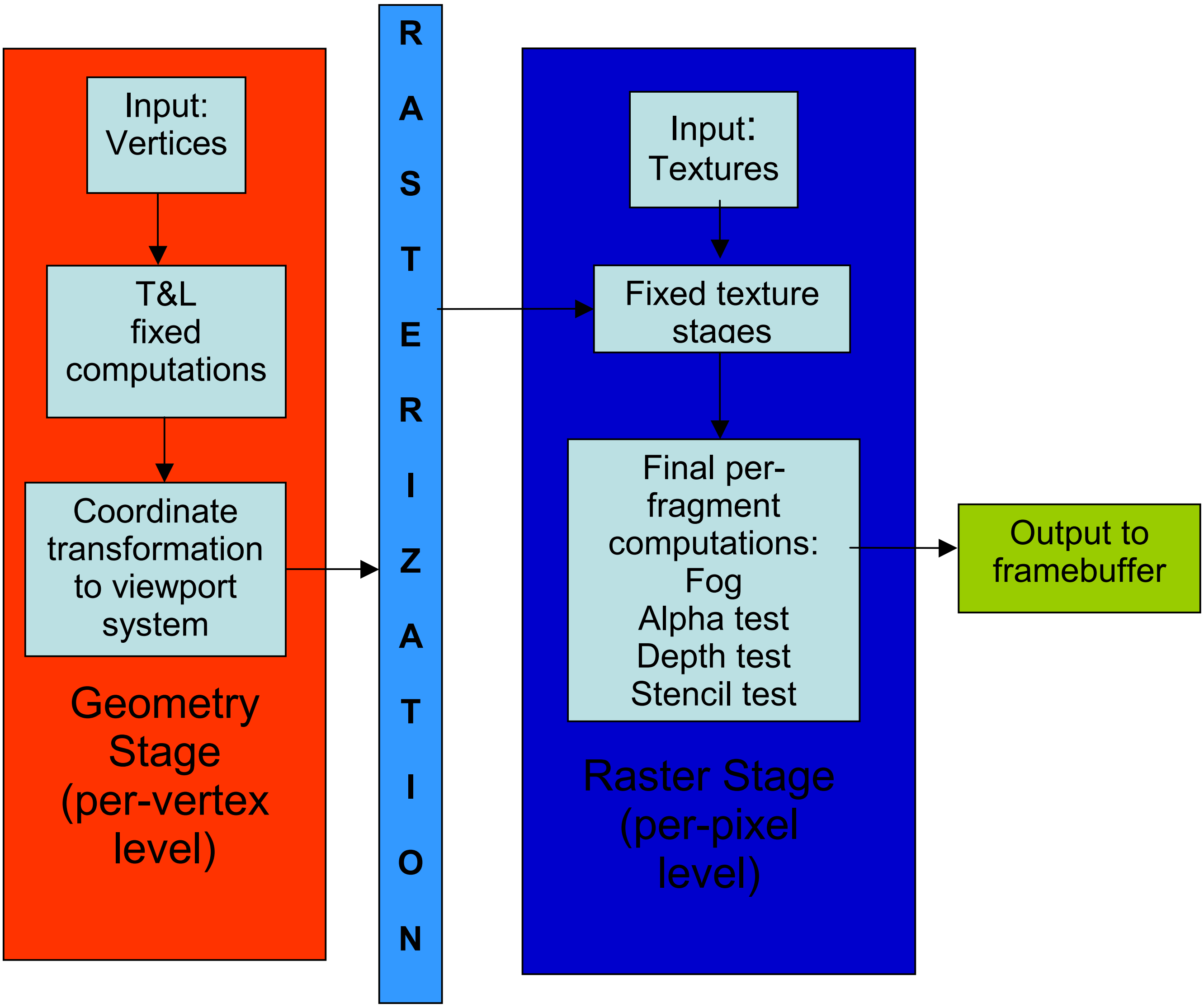
These fixed methods allowed the programmer to display many basic lighting models and effects, like light mapping, reflections, and shadows (always on a per-vertex basis) using multi-texturing and multiple passes. This was done by essentially multiplying the number of vertices sent to the graphic card (two passes = x2 vertices, four passes = x4 vertices, etc.), but it ended there.

With the programmable function pipeline, these limits were removed. All fixed per-vertex and per-fragment computations could be replaced by custom computations, allowing developers to do vertex displacement mapping, morphing, particle systems, and such all within the vertex stage. Per-pixel lighting, toon shading, parallax mapping, bump mapping, custom texture filtering, color kernel applications, and the like could now be controlled at the pixel stage. Fixed functions were now replaced by custom developer programs.

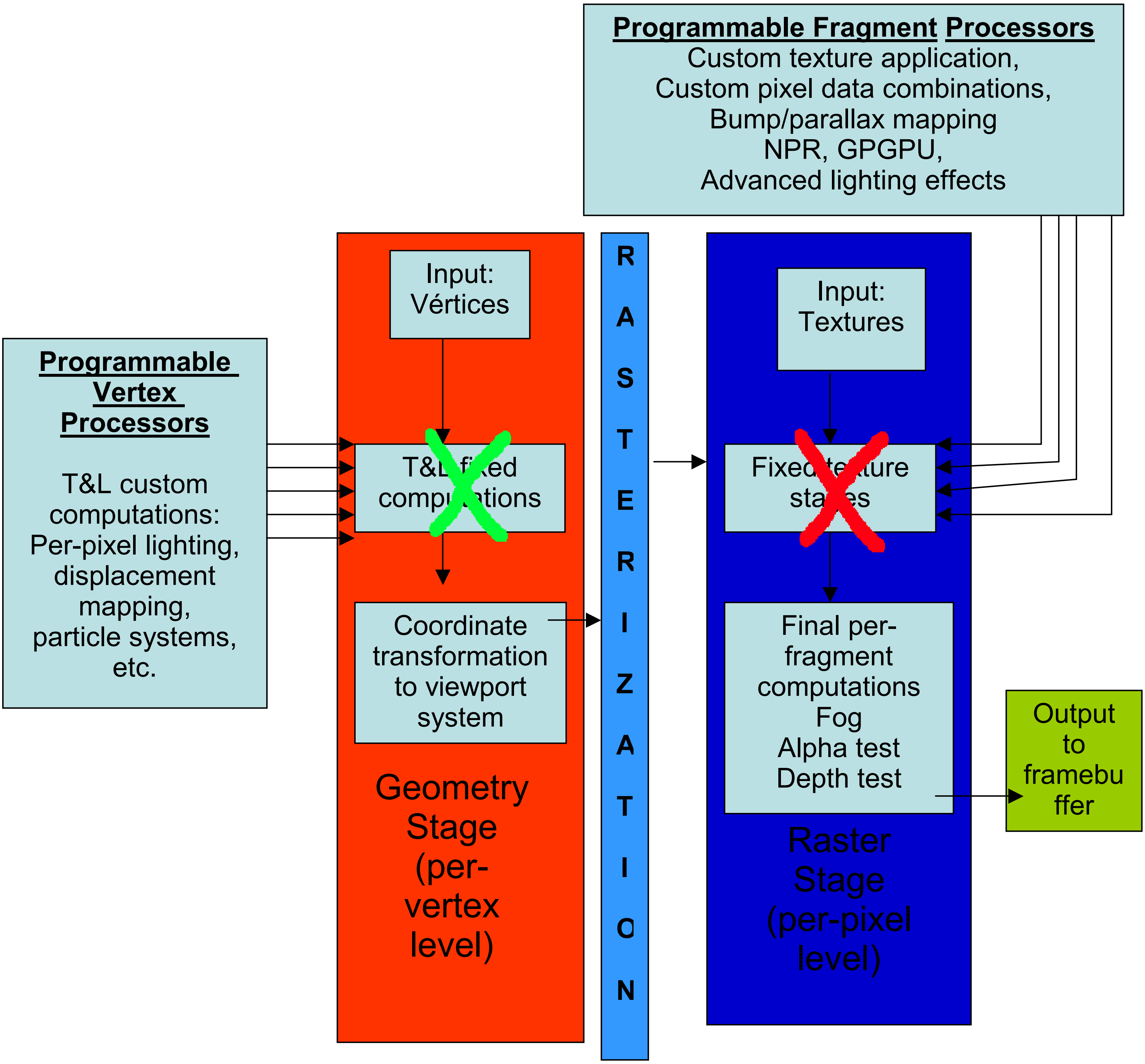
There are many advantages to a programmable pipeline. For example, some fixed functionalities could be disabled for simple shaders, producing a greater performance gain. Additionally, some CPU-based offline renders could now be calculated faster through the use of more complex shaders (imagine 3DSMax with hardware-based rendering, so scenes that usually take hours or even days to calculate are now displayed within a fraction of the time).

Another field where the programmable pipeline could be useful is as a co-processor to the CPU. Work on this area has already begun, and can be found at the General-Purpose Computation using Graphics Hardware (GPGPU) homepage (http://www.gpgpu.org). Many examples can be found here, including sound processors, fluid simulations, signal processing, computational geometry, imaging, scientific computing, and stream processing.

Fixed Function Scheme



Programmable Function Scheme



Shader 'Input' Data

Programmers can write self-contained standalone shaders, which don't require any extra data to run, in order to produce desired results

Shaders (both vertex and fragment) usually obtain some input values, such as textures, limit and timing values, colors, light positions, tangents, bi-normals, and pre-computed values, which are used to compute the final vertex position/fragment color for any given surface.

Uniform Variables
Uniform variables can use one of the GLSL-defined types. These read-only values (which should be treated as constants, as they cannot be changed) are then passed from the host OpenGL application to the shader.

GLSL data type	C data type	Description
bool	int	Conditional type, taking on values of **true** or **false**.
int	int	Signed integer.
float	float	Single floating-point scalar.
vec2	float [2]	Two component floating-point vector.
vect3	float [3]	Three component floating-point vector.
vec4	float [4]	Four component floating-point vector.
bvec2	int [2]	Two component Boolean vector.
bvec3	int [3]	Three component Boolean vector.
bvec4	int [4]	Four component Boolean vector.
ivec2	int [2]	Two component signed integer vector.
ivec3	int [3]	Three component signed integer vector.
ivec4	int [4]	Four component signed integer vector.
mat2	float [4]	2×2 floating-point matrix.
mat3	float [9]	3×3 floating-point matrix.
mat4	float [16]	4×4 floating-point matrix.
sampler1D	int	Handle for accessing a 1D texture.
sampler2D	int	Handle for accessing a 2D texture.
sampler3D	int	Handle for accessing a 3D texture.
samplerCube	int	Handle for accessing a cubemap texture.
sampler1DShadow	int	Handle for accessing a 1D depth texture with comparison.
sampler2DShadow	int	Handle for accessing a 2D depth texture with comparison.

From the host application, values could be passed to the shader as follows:

```
location = glGetUniformLocationARB(program,"light0Color");
float color[4] = {0.4f,0,1,1};
glUniform4fARB(location ,color );
```

The shader must first declare the variable before it can be used, which can be done as follows:

```
uniform vec4 light0Color;
```

If the variable `light0Color` is queried by the shader, it would return the value `{0.4, 0, 1, 1}`.

Textures must also be passed via uniforms. When passing textures, the developer must send an integer, which represents the texture unit number. For example, passing `0` would tell the shader to use `GL_TEXTURE0`, and so on:

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, mytexturebaseID);
location = glGetUniformLocationARB(program, "baseTexture");
glUniform1iARB(location, 0); // Bind baseTexture to TU 0.

glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, mytexturebumpID);
location=glGetUniformLocationARB(program, "bumpTexture");
glUniform1iARB(location, 1); // Bind bumpTexture to TU 1.
```

The uniforms are declared as `sampler2D` within the shader (though the actual texture unit will be discussed at a later point):

```
uniform sampler2D baseTexture;
uniform sampler2D bumpTexture;
```

Vertex Attributes

These variables can only be used within vertex shaders to pass per-vertex values. There are two types of attributes: *defined* and *generic*.

Defined attributes are normals, texture coordinates, per-vertex color materials, etc. Even the vertex position is a vertex attribute.

Generic attributes are those which the developer defines for meshes, like tangents, bi-normals, particle properties, and skinning information (bones).

When the developer creates a mesh, they must specify the 'Vertex Format.' This format is a collection of vertex attributes which will be sent to the vertex shader (like position, color, normal, texture coordinate, and tangent). For defined attributes, we have standard OpenGL functions like `glVertex3f`, `glNormal3f`, `glColor`, and `glTexCoord2f`. For generic attributes, we have the `glVertexAttrib` call.

The method for passing generic attributes is a little different. The GL_ARB_vertex_program extension holds a number of slots where attributes can be placed. These slots are shared between the defined and generic attributes, meaning defined slots can be overwritten and their attribute lost. Defined attributes always use the same slot numbers, so you can choose which one to overwrite, or use a free slot (you can ask OpenGL for a free slot).

The following code could be used to pass a generic attribute to a shader through a given slot:

```
int slot = 9; //A random slot.
glBindAttribLocationARB(program, slot, "fooAttribute");
glBegin(GL_TRIANGLES);
      glVertexAttrib3fARB(slot,2,3,1);
      glVertex3f(0,1,0);
      glNormal3f(1,0,0);

      glVertexAttrib3fARB(slot,2,1,1);
      glVertex3f(0,0,1);
      glNormal3f(1,0,0);

      glVertexAttrib3fARB(slot,2,3,2);
      glVertex3f(1,0,0);
      glNormal3f(1,0,0);
glEnd();
```

To access the attribute from the vertex shader, the variable has to be declared as follows:

```
attribute vec3 fooAttribute;
```

Attributes only can be declared with **float, vec2, vec3, vec4, mat2, mat3**, and **mat4**. Attribute variables cannot be declared as arrays or structures.

Vertex arrays can also be used to pass attributes, with calls like `glVertexAttribPointerARB`, `glEnableVertexAttribArrayARB`, `glBindAttribLocationARB` and `glDisableVertexAttribArrayARB`. See the appendix for how to use these generic vertex attribute calls.

Varying Variables

It is possible for a vertex shader to pass data to a fragment shader by use of another type of variable. Varying variables will be written by the vertex shader and read into the fragment shader (though the actual variable within the vertex shader will not be passed). The fragment shader will then receive the perspective-corrected and interpolated (across the primitive's surface) value of the variable written by the vertex shader. The best example of varying variables (sometimes called interpolators) is texture coordinates. Texture coordinates are established by the vertex shader, loaded as vertex attributes, and then written into varying variables in order to pass an interpolated value in a perspective-correct fashion into the fragment shader.

For example:

[Vertex shader]

```
varying vec2 myTexCood;
void main()
{
  // We compute the vertex position as the fixed function does.
  gl_Position = ftransform();
  // We fill our varying variable with the texture
  //coordinate related to the texture unit 0 (gl_MultiTexCoord0 refers to the TU0
  //interpolator).

  myTexCoord = vec2(gl_MultiTexCoord0);
}
```

[Fragment shader]

```
varying vec2 myTexCoord;
uniform sampler2D myTexture;
void main()
{
   //Use myTexCoord by any way, for example, to access a texture.
   gl_FragColor = texture2D(myTexture, myTexCoord);
}
```

Shader 'Output' Data

Vertex Shader

The main objective of the vertex shader is to compute the vertex position within the clip-space coordinates. To do this, GLSL's built-in **`gl_Position`** variable can be used (which has a `vec4` type) in one of two ways:

a) `gl_Position = ftransform();`
This is usually the best way, as ftransform() keeps the invariance within a built-in fixed function.

b) `gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;`
This will compute the correct vertex position, as it multiplies the vertex position (`gl_Vertex`) by the model-view which is once again multiplied by the projection matrix (`gl_ModelViewProjectionMatrix` is a built-in uniform `mat4` variable, which holds the result of `gl_ModelViewMatrix * gl_ProjectionMatrix`). However, this will not keep the invariance within a fixed function, and could prove problematic in multi-pass algorithms.

Fragment Shader

The main objective of a fragment shader is to compute the final color (and optionally, depth) of the fragment being computed. To do this, GLSL's built-in `gl_FragColor` variable can be used (which also has a `vec4` type):

```
gl_FragColor = vec4(1, 0, 0, 0);
```

The above example will write a pure red color with an alpha value of `0` to the framebuffer.

There are more values that can be written within the vertex and fragment shaders, like information relating to clipping plane(s), point parameters. and fragdepth, but all of these are optional.

If a vertex shader doesn't contain a **`gl_Position`** variable, it won't compile, and instead will simply generate compilation errors instead. The same is true of fragment shaders if **`gl_FragColor`** is not used. These two variables are absolutely mandatory for a successful shader.

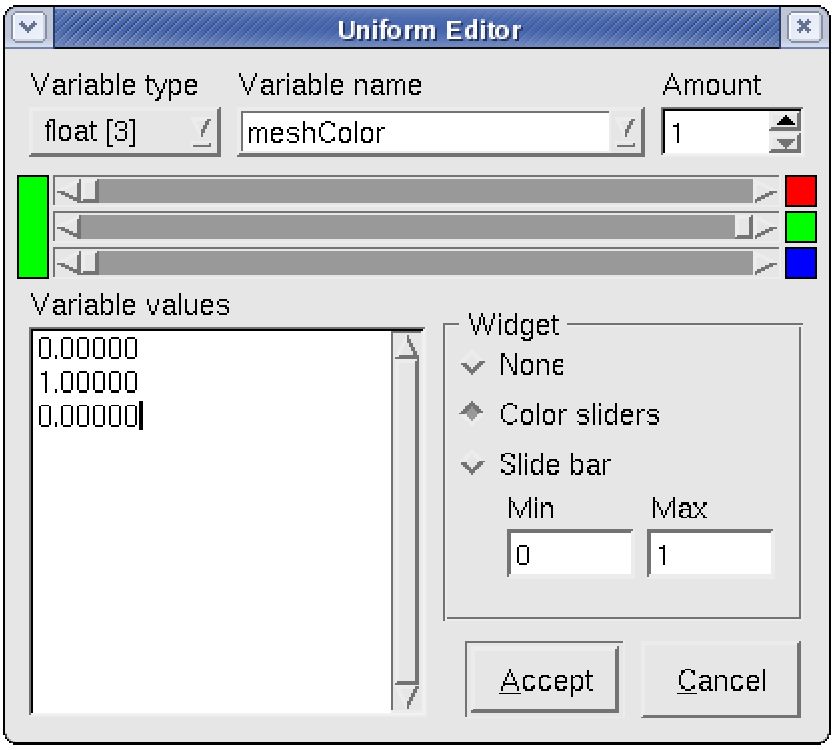
Simple GLSL Shader Example

We will now create a simple shader using TyphoonLabs' **OpenGL Shader Designer** (SD) IDE. This example will consist of a uniform variable with one color, and apply that color to a pre-defined mesh.

Open SD and select **File > New Shader Project** from the main menu. This will create a new workspace, adding both an empty vertex and fragment shader to the project while resetting all fields back to their defaults.

Right-click within the 'Uniform Variables' window (bottom-left area of the user interface) and select **New Uniform** from the context menu. Once the 'Uniform Editor' dialog appears, enter the following values:

Variable name: *meshColor*
Variable type: *float[3]*
Widget: *Color sliders*
Variable values: *(see screenshot below)*



Now press **Accept**, which will close the current dialog and apply your changes.

Select the 'New.Vert' tab within SD's main user interface and enter the following code:

```
void main()
{
    gl_Position = ftransform();
}
```

We don’t need anything else inside the vertex shader besides the correct position of the vertex within clip coordinates (handled by the built-in `ftransform()` function).

Select the 'New.Vert' tab within SD's main user interface and enter the following code:

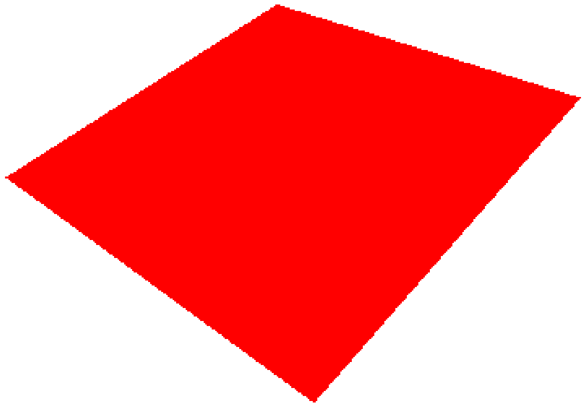
```
uniform vec3 meshColor;
void main()
{
  gl_FragColor = vec4(meshColor,1.0);
}
```

The **line** **`uniform vec3 meshColor;`** allows us to access the values held within our uniform variable, which we then use in the line **`gl_FragColor = vec4(meshColor,1.0);`**. We must use the **`vec4`** constructor, as **`gl_FragColor`** is a **`vec4`** type variable, meaning this constructor will construct a **`vec4`** variable for the first three components equal to the **`meshColor`**, with 1.0 as an alpha value.

Our shader example is now finished. Select **Build > Compile Project** from the main menu to view the results. If no errors were generated, a green-colored mesh should appear within the 'Preview' window (top left-hand corner of the user interface). If that is not the case, check the uniform variable and compiler output to see where the problem lies.

You can easily change the color of the shader result by right-clicking the **`meshColor`** variable within the 'Uniform Variables' window, then selecting **Floating Editor** from the context menu. A slider-bar widget will now appear, allowing you to dynamically control the overall color of the mesh. Other types of widgets can also be created, like color pickers and sliding-bars with up to four components.

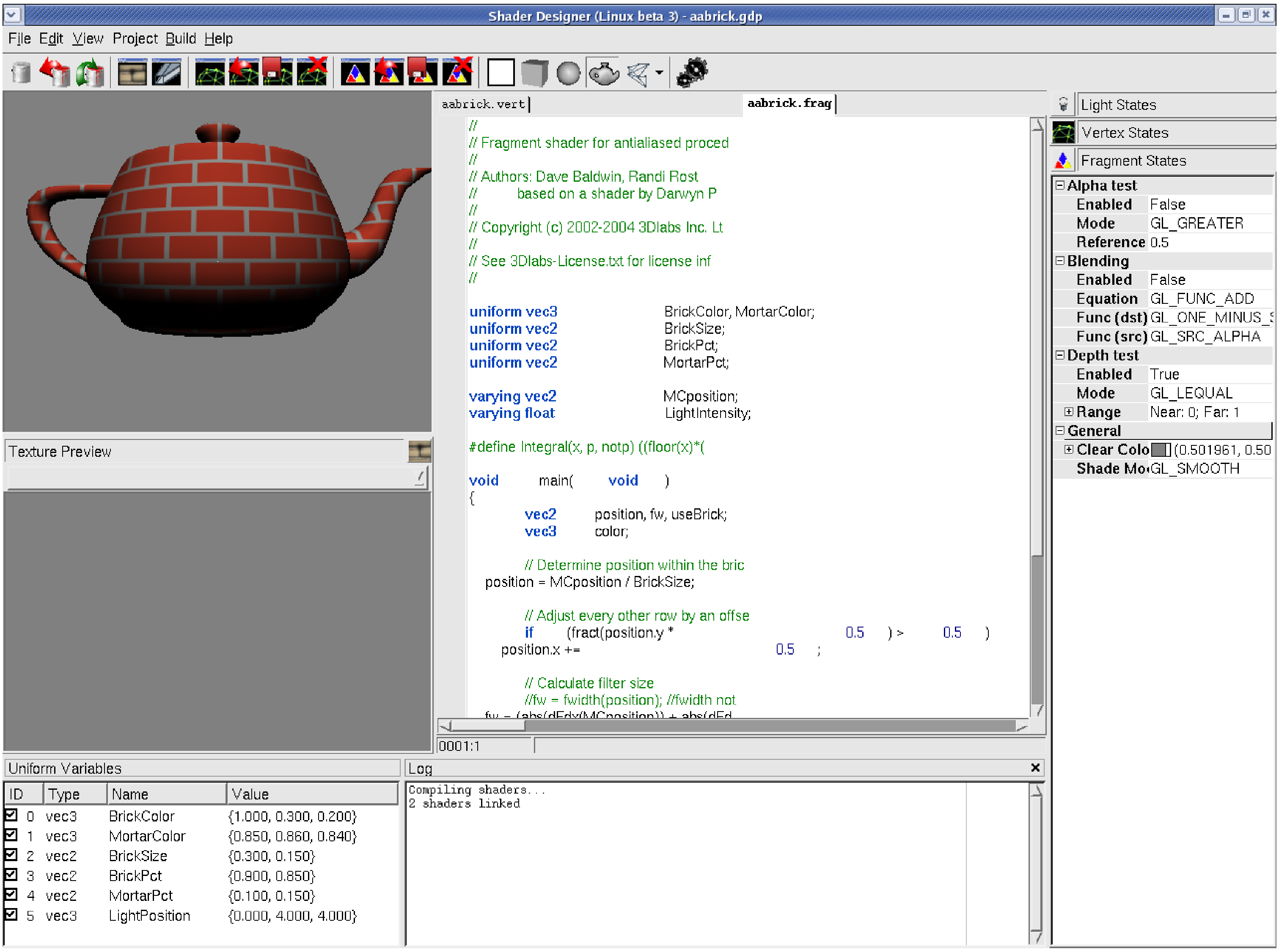
A **meshColor** value of `1,0,0` should look something like this (a rotated plane):



Shader Designer IDE

We'll now take a more detailed look at Shader Designer's user interface, covering basic operations like simple source file management, uniforms, and textures.

User Interface



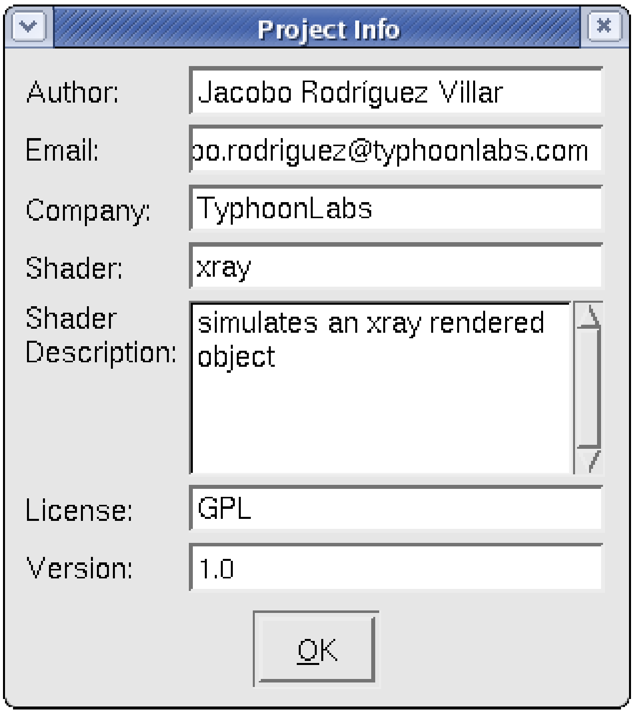
This is the main application window, which is divided into the following sections:

Toolbar
This allows quick access to commonly used operations. Reading from left to right, you can select New project, Open project, Save project,etc.

Menu
This allows you to access the complete feature-set of Shader Designer, including the toolbar entries. Some of the new options are:

Validate will compile shaders using the 3DLabs' generic GLSL compiler. This allows developers to write shaders that are compatible with the GLSL specification, which is very useful when trying to create portable shaders.

Project info will display the current project's information (metadata) through fields like author, company, license, copyright, comments, and version number.



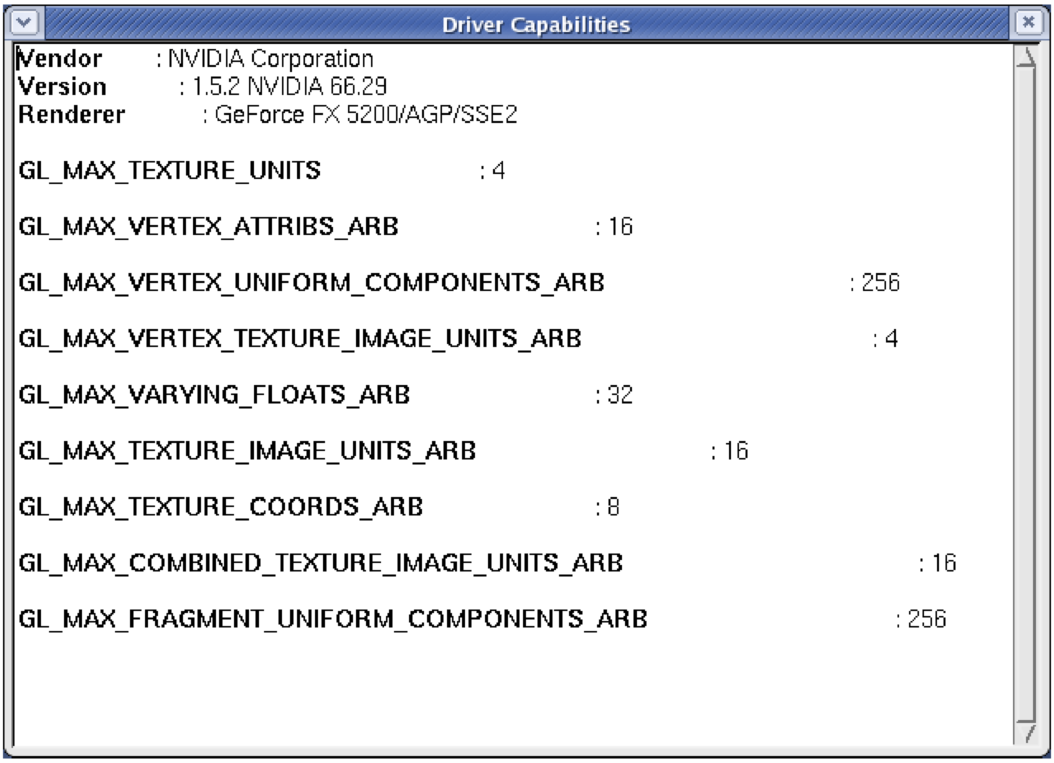
Font will allow you to change the default font used within the code window.

Take snapshot will take a screenshot of the current project.

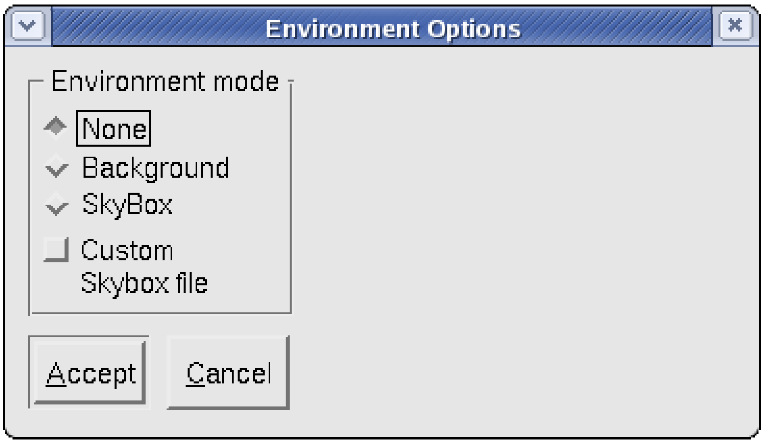
Continuous render toggles the GLSL mode, allowing shaders to be disabled. This is useful for cards which run shaders on the CPU (though CPU performance is a lot slower than GPU).

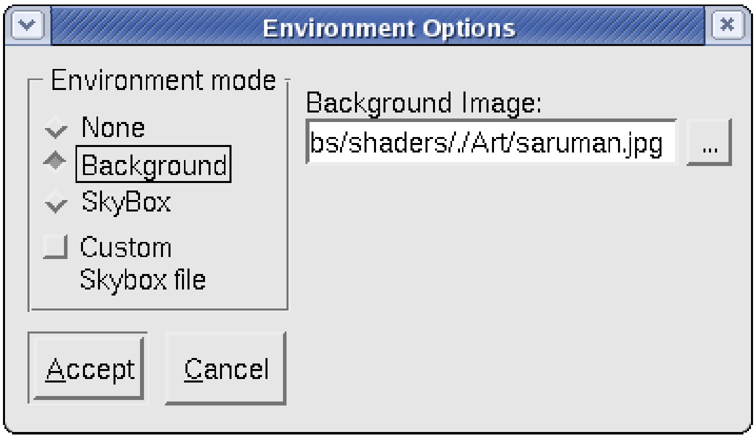
Cut, Copy and Paste is a standard feature, used within the code window.

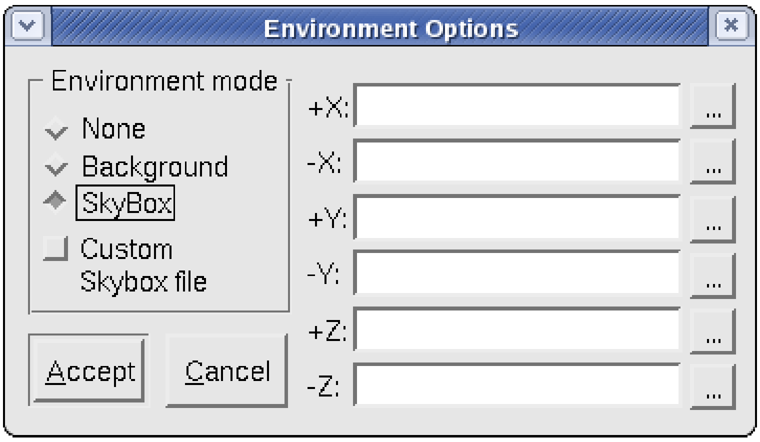
Driver capabilities will display a window containing the OpenGL information of your current graphics card. Here you can look for possible limitations, as well as the maximum number of textures and uniforms that your card can support.



Environment will configure the background setup of the 'Preview' window. You can choose whether to use a plain background (no image), import an image, or use a Skybox (a collection of six images forming a cube, which the mesh is placed within). With the exception of **none**, all options require you to import one or more images.

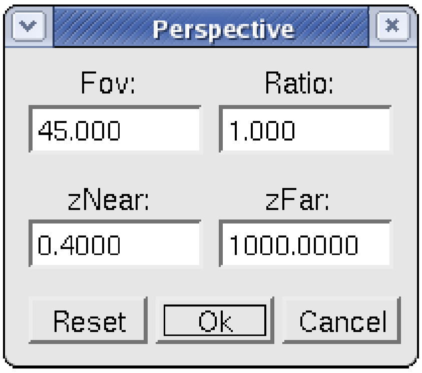




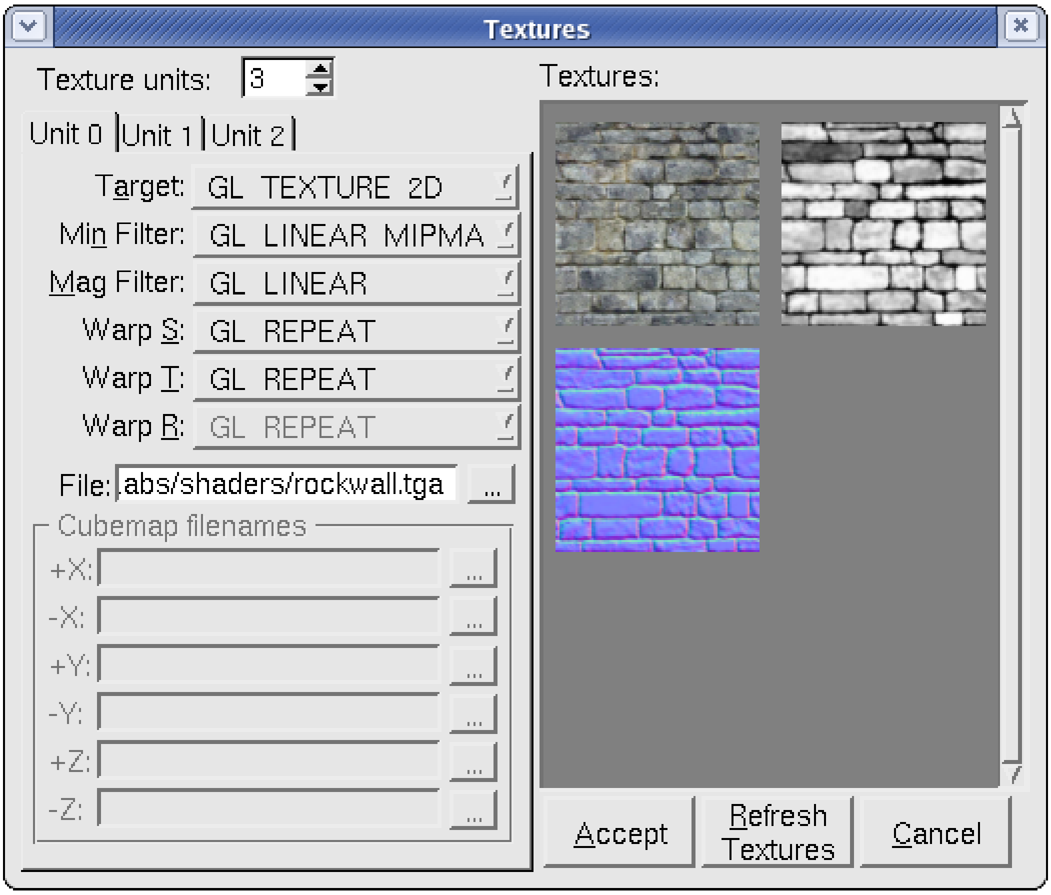


Ortho toggles the 'Preview' window between perspective mode and orthogonal, the latter of which is useful for imaging shaders or screen-space shaders (like Julia and Mandelbrot).

Perspective allows you to configure the settings used for the 'Preview' window's perspective mode.



Textures allows you to select the various textures that will be used in your shader, as well as configure their parameters. A preview thumbnail is provided for all of the images.



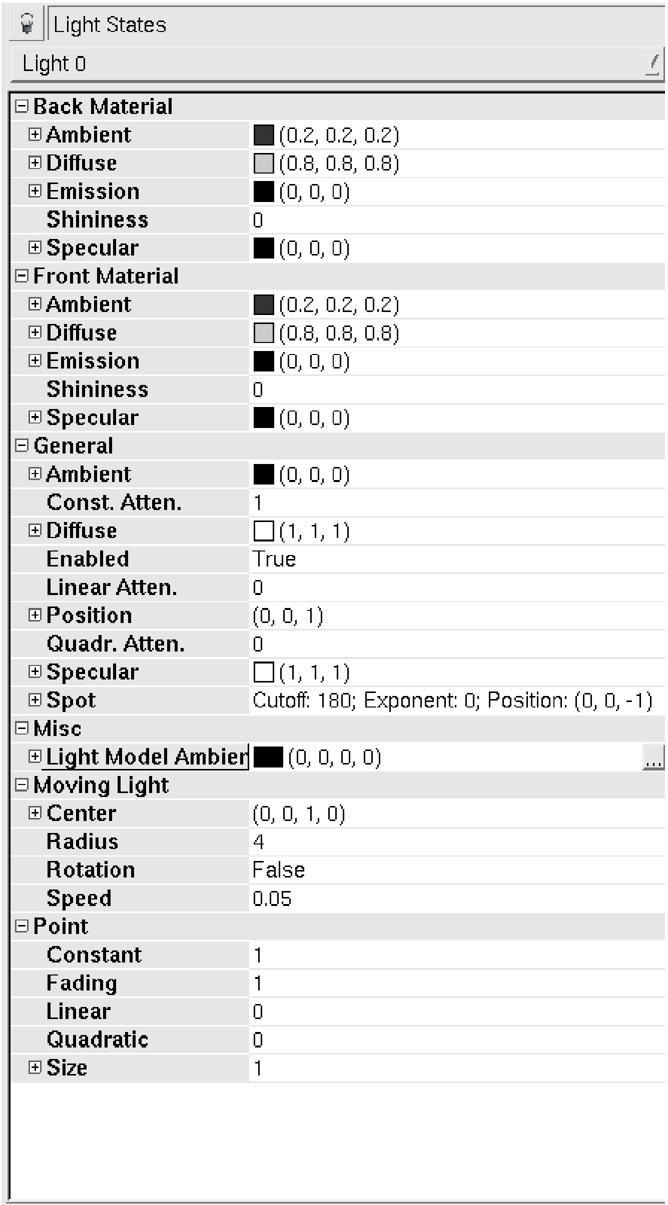
As this is one of the most complex dialogs, we'll take a closer look at the options:

First, you must select the number of textures you wish to use (this number is only limited by your graphic card's capabilities). Then, using each texture's tab, import the image(s) using the respective field(s). Next, choose the texture type (1D, 2D, etc.) and its filtering/clamping configuration. Use the **Refresh Textures** button to make sure your texture(s) still load, and if all is well, select **Accept** (which will apply your changes and close the dialog).

State Properties

This area of Shader Designer allows you to control the various OpenGL lighting, vertex, and fragment states of your shader, which will be stored/saved with the project (GDP) file.

Light States



Back/Front Material allows you to control the material used on the front and back faces of the mesh through Ambient, Diffuse, Specular, Emission, and Shininess options.

The values can be changed by clicking within the text field and manually editing the values, or by clicking the text field and selecting the '...' icon on its right-hand side. Although the alpha component is not usually visualized, it can be entered to the left of the other values (for example, 1,1,1,1), i.e. the ARGB pixel format.

These fields are also accessible from GLSL using built-in uniform variables:

```
struct gl_MaterialParameters
{
  vec4 emission; // Ecm
  vec4 ambient; // Acm
  vec4 diffuse; // Dcm
  vec4 specular; // Scm
  float shininess; // Srm
};
uniform gl_MaterialParameters gl_FrontMaterial;
uniform gl_MaterialParameters gl_BackMaterial;
```

General allows you to control the common OpenGL light settings, like Ambient, Diffuse, Specular, Position (using world coordinates), Attenuations (constant, linear, and quadratic), and Spot (cut-off angle, exponent, and direction). If the **Enabled** property is set to **False**, these parameters will be ignored.

These fields are also accessible from GLSL using built-in uniform variables:

```
struct gl_LightSourceParameters
{
  vec4 ambient; // Acli
  vec4 diffuse; // Dcli
  vec4 specular; // Scli
  vec4 position; // Ppli
  vec4 halfVector; // Derived: Hi
  vec3 spotDirection; // Sdli
  float spotExponent; // Srli
  float spotCutoff; // Crli  // (range: [0.0,90.0], 180.0)
  float spotCosCutoff; // Derived: cos(Crli)  // (range: [1.0,0.0],-1.0)
  float constantAttenuation; // K0
  float linearAttenuation; // K1
  float quadraticAttenuation;// K2
};
uniform gl_LightSourceParameters gl_LightSource[gl_MaxLights];
```

Misc allows you to control the ambient lighting model.

These fields are also accessible from GLSL using built-in uniform variables:

```
struct gl_LightModelParameters
{
  vec4 ambient; // Acs
};
uniform gl_LightModelParameters gl_LightModel;
```

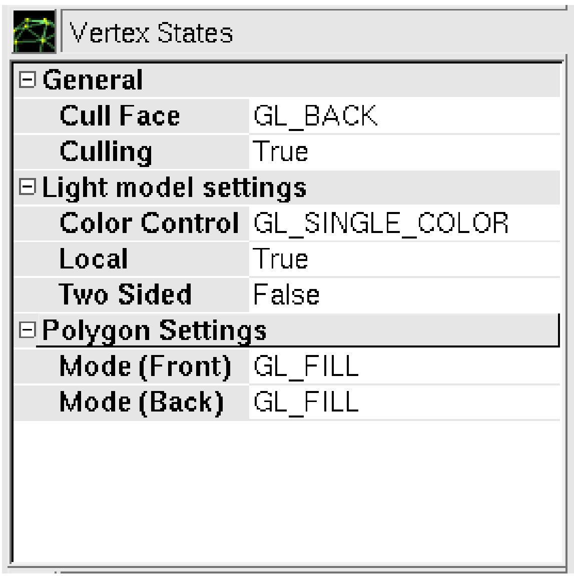
Moving light allows you to control Shader Designer's dynamic moving light effect, which is sometimes useful for testing bump map-style shaders (or other lighting algorithms). The light will rotate around a center at a given speed and distance.

Point allows you to control the point parameters (the POINT_PARAMETERS extension is needed for this feature).

These fields are also accessible from GLSL using built-in uniform variables:

```
struct gl_PointParameters
{
  float size;
  float sizeMin;
  float sizeMax;
  float fadeThresholdSize;
  float distanceConstantAttenuation;
  float distanceLinearAttenuation;
  float distanceQuadraticAttenuation;
};
uniform gl_PointParameters gl_Point;
```

Vertex States

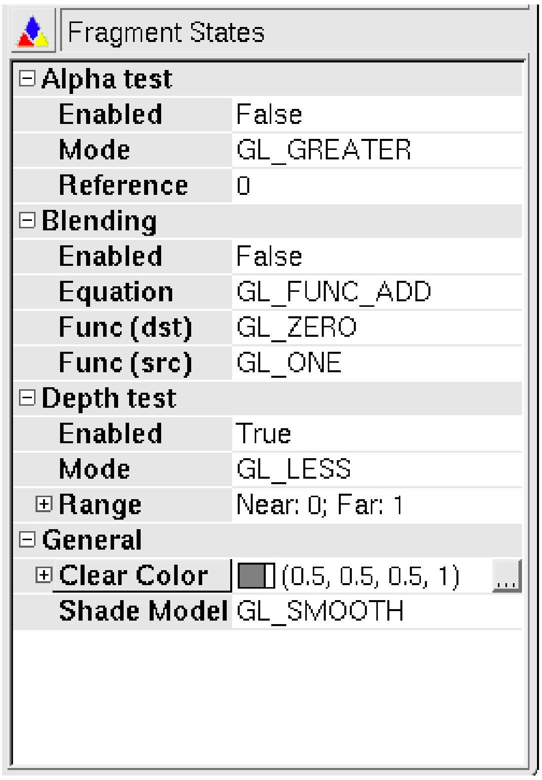


General allows you to control culling type used, if enabled.

Light Model Settings allows you to control three of four possible `glLightModelfv` parameters, `GL_LIGHT_MODEL_COLOR_CONTROL` (single color or separate specular color component), `GL_LIGHT_MODEL_LOCAL_VIEWER`, and `GL_LIGHT_MODEL_TWO_SIDE`.

Polygon Settings allows you to control the drawing mode for both faces (front and back) of polygons, using `GL_FILL`, `GL_LINE` or `GL_POINT`.

Fragment States



Alpha Test allows you to control the OpenGL alpha test stage, with **Mode** representing `alphafunc` and **Reference** representing the clamping reference.

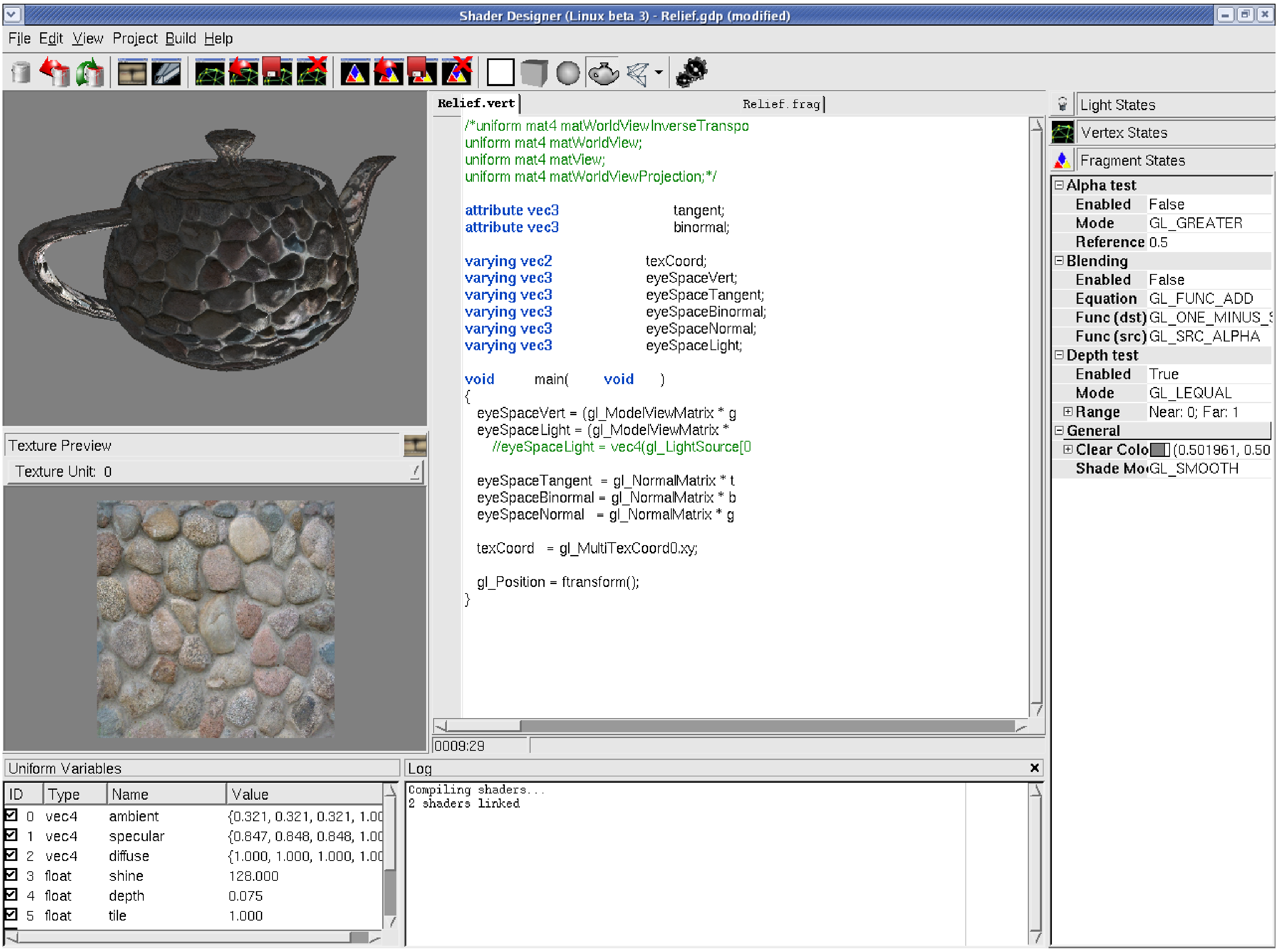
Blending allows you to control the OpenGL blending stage via a blend equation (the `GL_ARB_imaging` extension is needed for this feature) and a `blendfunc`, using **dst** (for destination) and **src** (for source) factors.

Depth test allows you to control the depth test mode and range.

General allows you to control the framebuffer's default color (RGBA mode) and flat or gouraud shading models.

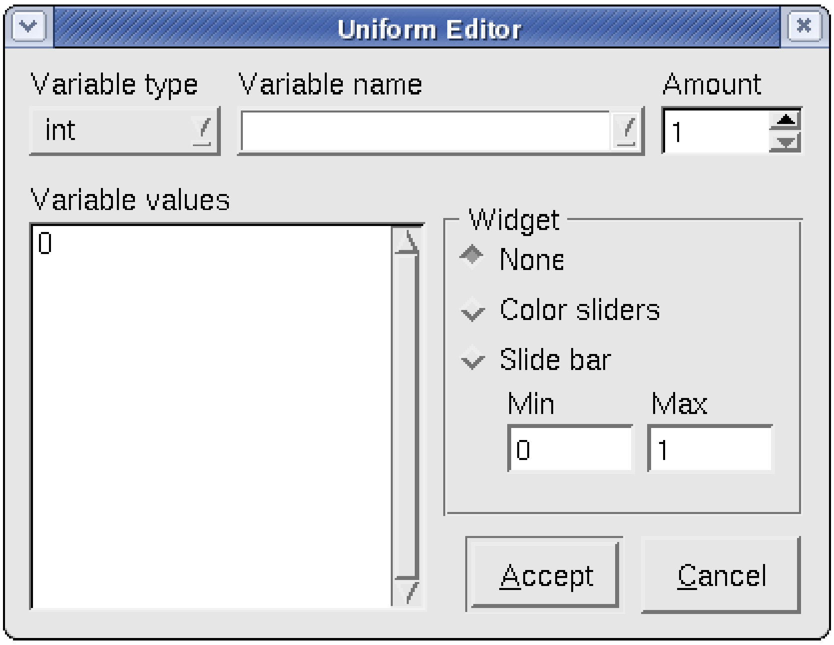
Code Window

The code window has some useful features, like intellisense, syntax highlight, and tooltips.



Uniform Variables Manager

In order to access this dialog, you will need to right-click within the uniform list and choose either **New Uniform** or **Edit**:

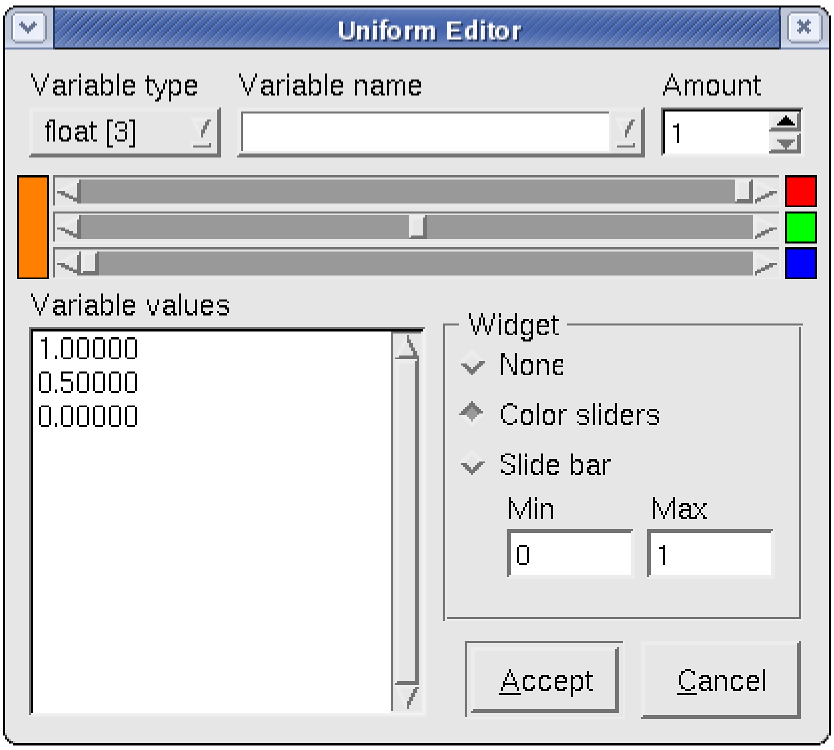


Once open, you can select the uniform type, name, amount (array size), variable values (each value must be separated by a new line), and the widget to be used if changing the value dynamically.

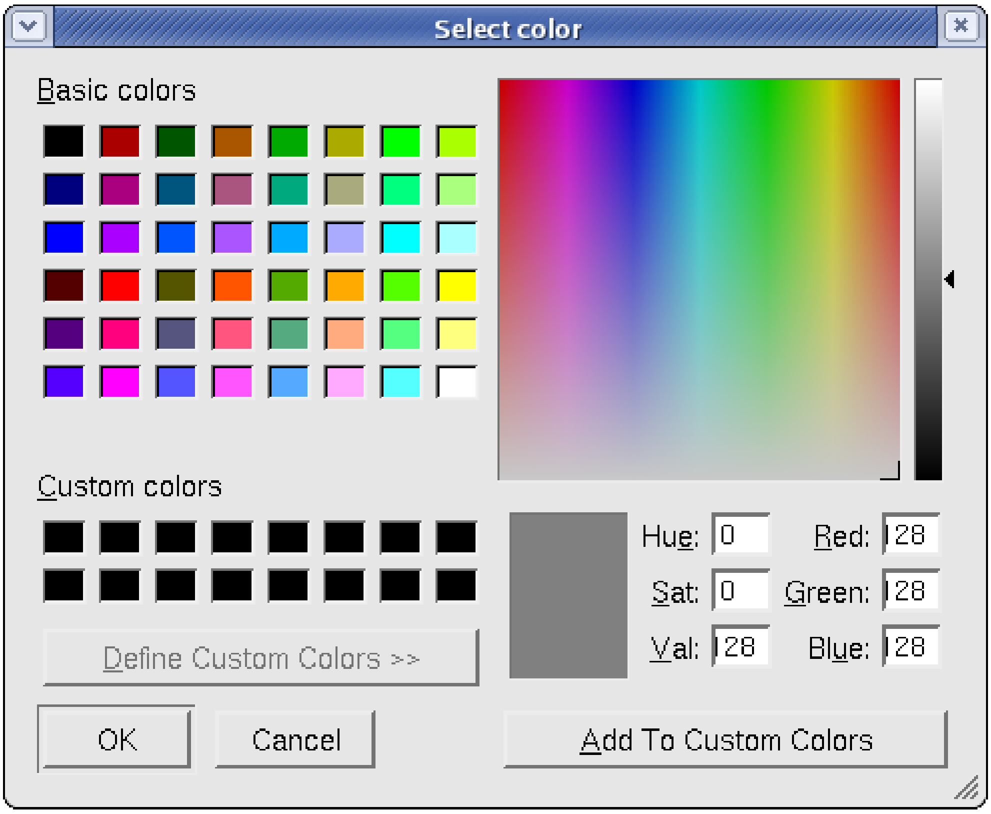
The widget options are as follows:

None will not use a widget for the uniform.

Color sliders can be used for `vec3` and `vec4` type variables, allowing the following control:

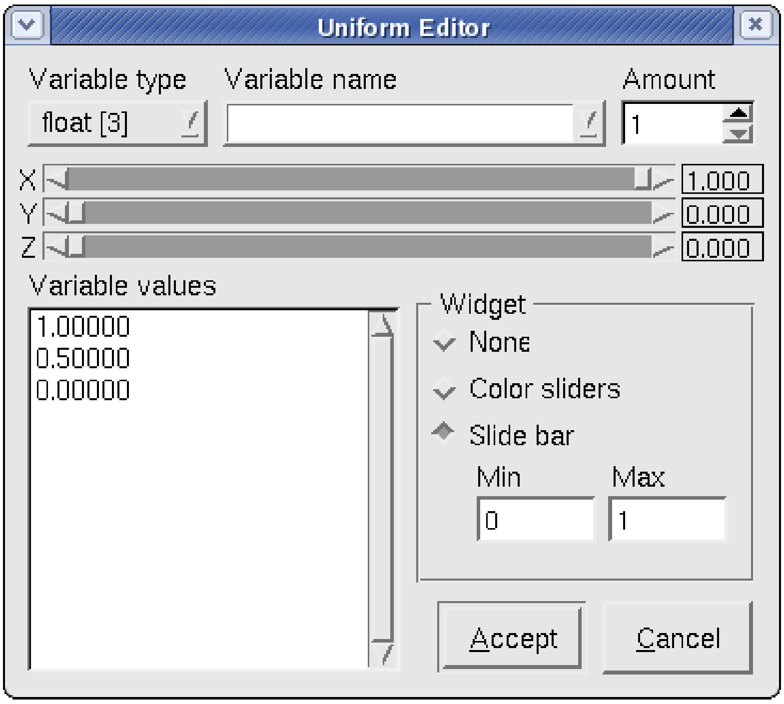


Selecting the colored quads will open a standard color-picker dialog:



Right-clicking the uniform variable from within the list and selecting **Floating editor** will allow you to change colors quickly:

Slider Bar works in a similar way to the color slider widget, but there are no colored boxes, and the user can control the maximum and minimum limits of the slider:



Right-clicking the uniform variable from within the list and selecting **Floating editor** will allow you to change colors quickly: