OpenGL Shader Builder User Guide

Graphics & Imaging > OpenGL

2008-06-23

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Figures and Tables

Introduction

OpenGL Shader Builder is a tool for developing and debugging programs for the graphics processing unit (GPU). It can help you visualize and preview shader objects without the complexity of surrounding code. Using it, you can:

- Get immediate feedback as you enter and modify GPU programs
- Explore the effect of changing texture parameters
- Track down link and compile errors
- Observe the effect of making changes to symbol values

Developers who are writing GPU programs will want to read this document to find out how to use OpenGL Shader Builder. You can use the shader builder with programs written with OpenGL Shading Language or with older-style ARB vertex and fragment programs. OpenGL Shader Builder also supports geometry shaders, a recent addition to the OpenGL specification.

Organization of This Document

This document is organized into the following chapters:

- "Getting Started" (page 9) gives an overview of the user interface and the main features of OpenGL Shader Builder.
- "Building Shaders" (page 19) provides step-by-step instructions for the most common tasks you can accomplish.

See Also

You may want to consult these documents as you develop shaders for the GPU:

- OpenGL Shading Language (PDF) provides an overview of shaders and a complete reference to the language.
- OpenGL Shading Language (GLSl) Quick Reference Guide (PDF) is a two-page list of symbols that includes cross-references to the full specification.

The following OpenGL specifications define the extensions that support GPU programs:

- GL_EXT_geometry_shader4 is for generating primitives.
- GL_ARB_fragment_program is for processing fragments.

- GL_ARB_vertex_program is for processing vertices.

Getting Started

OpenGL Shader Builder is a development environment for writing, testing, and experimenting with OpenGL shaders. OpenGL Shader Builder not only speeds development for seasoned shader developers, but it can help those new to writing shaders to explore how shaders work. Using it, you can focus on the shader code; OpenGL Shader Builder takes care of the rest. You can use it to:

- Parse source code and check the syntax
- Compile and link source code files to create shader objects
- Change and animate the values of uniform variables
- Preview textures before applying them to an object
- Benchmark performance
- Enable and disable a shader so you can see its effect more clearly

After you install the developer tools, you can find OpenGL Shader Builder in this directory:

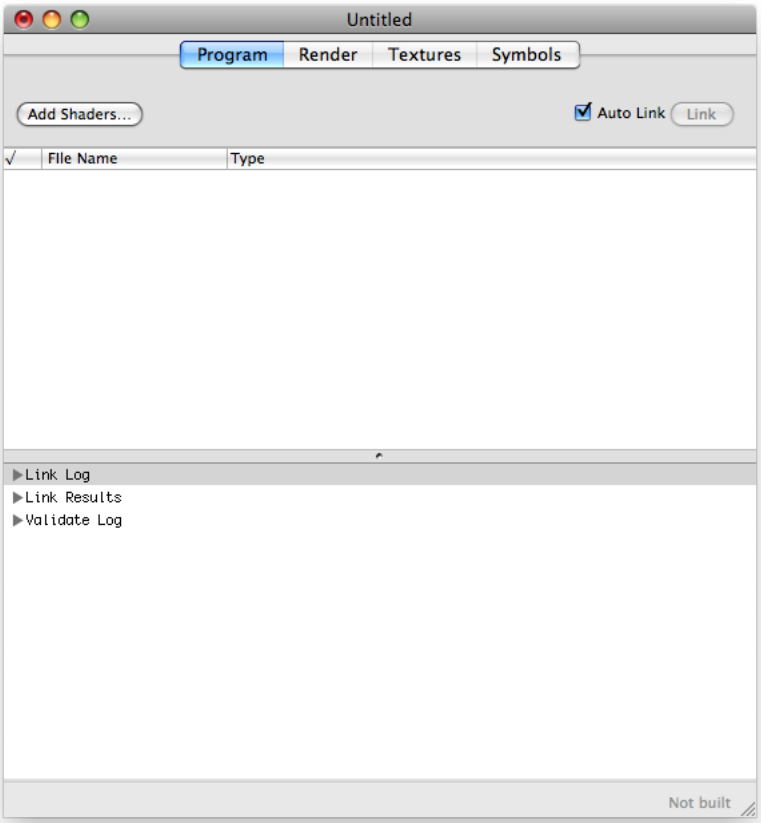
```
/Developer/Applications/Graphics Tools/
```

If you've used OpenGL Shader Builder before, you'll notice that the user interface for version 2.0 is a bit different from the previous version. Before you start using it, you'll want to get acquainted with the four views it provides—Program, Render, Textures, and Symbols.

Program View

When you launch OpenGL Shader Builder, it opens to the Program view shown in Figure 1-1 (page 10). You use this view to manage source code files and to check linking and validation of the code.

Figure 1-1 The Program view

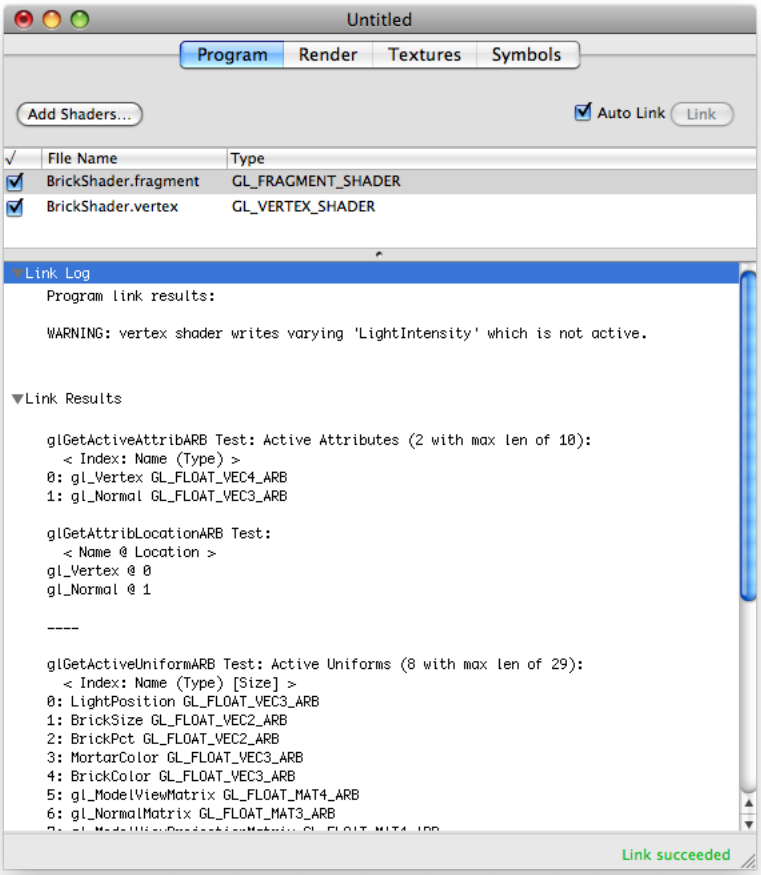


The top part of the view is used for listing source code filenames. You can add files in any of these ways:

- Drag previously created files to the window.
- Use the Add Shaders button to navigate and choose previously created files.
- Choose File > New > to create a new file for a GLSL program (vertex, fragment, geometry) or an ARB program (vertex, fragment).

Figure 1-2 (page 11) shows the Program view after you've adding fragment and vertex source code files. The link log, link results, and validation log appear below the file list. The link status, which in this case is "Link succeeded," appears in the lower-right corner of the window.

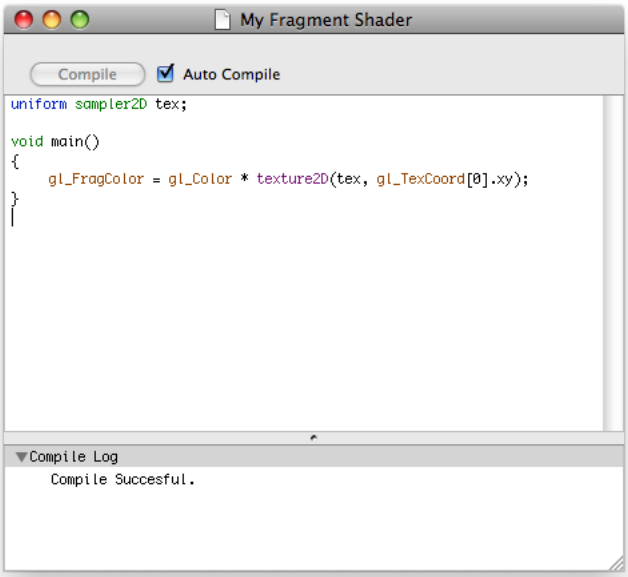
Figure 1-2 The Program view after adding two files



Source Code Files

You can view and modify the contents of each source code file by double-clicking its name in the file list. The file opens in a new window, as shown in Figure 1-3 (page 12). When you create a new source code file, it opens automatically in a new document window. In contrast, new source code files open in a new document window automatically. These new source code files contain template code that you can modify or replace to suit your needs.

Figure 1-3 A source code file opens in a separate document window

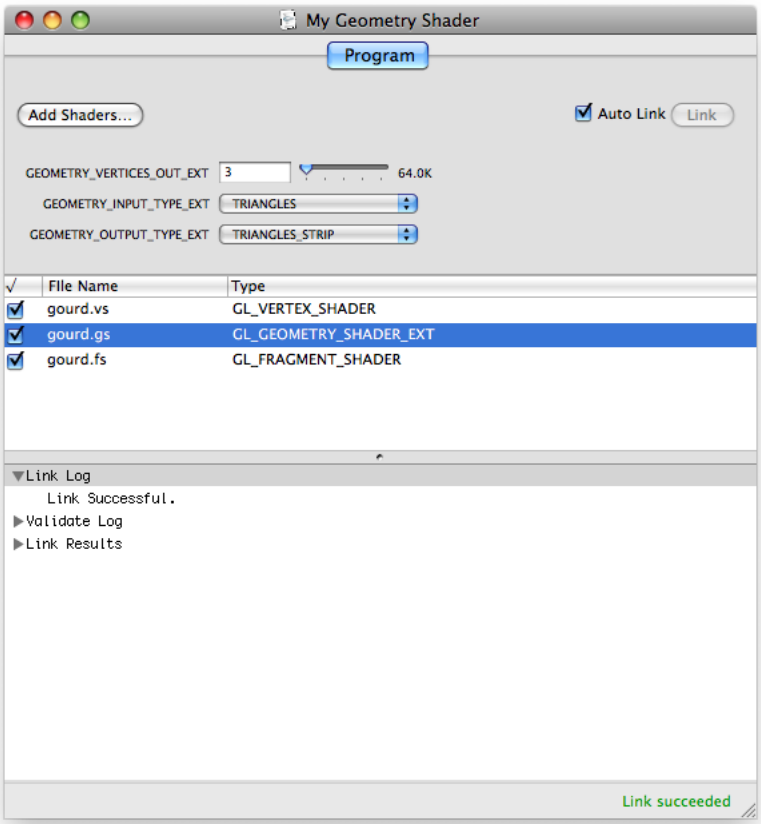


Controls for Geometry Shaders

A geometry shader object is made up of a geometry program and a vertex program. When you add the geometry source code file to the program list, the user interface changes (see Figure 1-4 (page 13)) to show controls for the following OpenGL parameters, which the GL_NV_geometry_shader4 extension defines:

- `GEOMETRY_VERTICES_OUT_EXT` is the maximum number of vertices produced by the geometry shader.
- `GEOMETRY_INPUT_TYPE_EXT` is the geometry that the shader takes as input: `POINTS`, `LINES`, `LINES_ADJACENCY_EXT`, `TRIANGLES`, or `TRIANGLES_ADJACENCY_EXT`.
- `GEOMETRY_OUTPUT_TYPE_EXT` is the geometry that the shader produces: `POINTS`, `LINE_STRIP`, or `TRIANGLE_STRIP`.

Figure 1-4 The Program view after adding a geometry shader

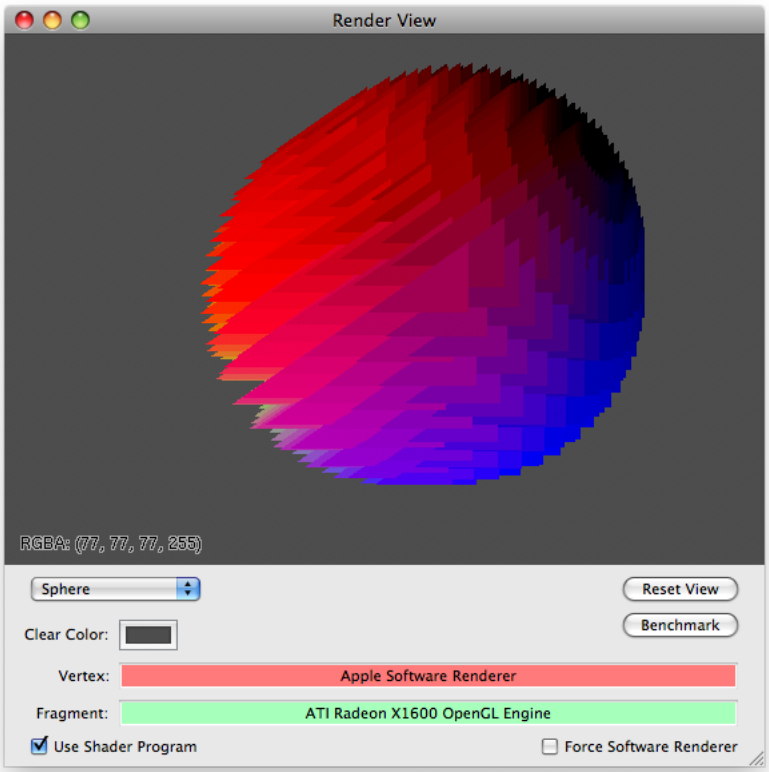


Render View

The Render view, shown in Figure 1-5 (page 14), visualizes what your code does. Although you can click the Render tab to switch between the Program and Render views, it's more efficient to double-click the Render tab to open the view in a separate Render window. That way, you can look at the rendering results side-by-side with the contents of the Program, Textures, and Symbols views.

For details on customizing the layout of windows, see "Creating and Saving a Layout" (page 19).

Figure 1-5 The Render view



The pop-up menu lets you choose from among several geometries—Teapot Wire, Teapot Point, Plane, Teapot, Squiggle, Sphere, or Torus—to apply your code to. You can interact with any of the 3D geometries by clicking and dragging the pointer.

Textures View

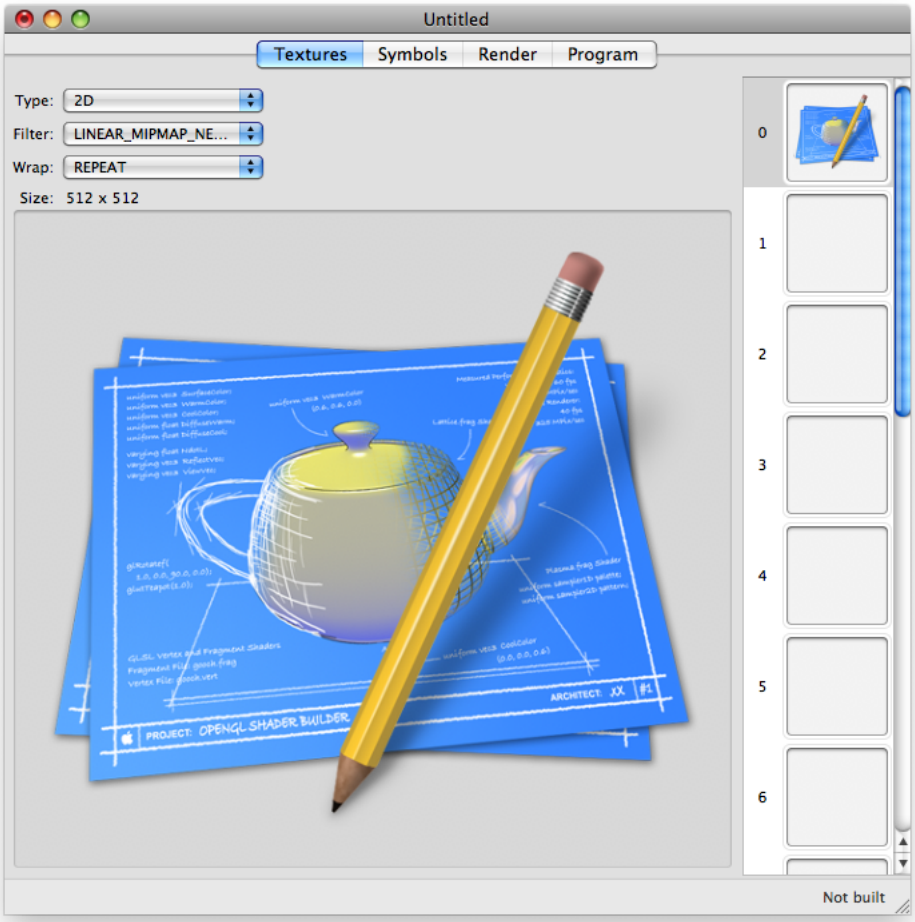
The Textures view, shown in Figure 1-6 (page 15), lets you add and set up textures to use as input to fragment programs. To add a texture, you simply drag it to one of the image wells on the right side of the view. When you select an image well, the texture appears on the left side of the view.

After selecting a texture, you can adjust any of the following by choosing the appropriate OpenGL constant from the provided pop-up menus:

- Texture types: `1D`, `2D`, `Rectangle`, `3D`, `Cube_MAP`, `SHADOW_1D`, `SHADOW_2D`, or `SHADOW_RECTANGLE`
- Methods of filtering: `NEAREST`, `LINEAR`, `NEAREST_MIPMAP_NEAREST`, `LINEAR_MIPMAP_NEAREST`, `NEAREST_MIPMAP_LINEAR`, or `LINEAR_MIPMAP_LINEAR`
- Wrapping modes: `REPEAT`, `CLAMP`, `CLAMP_TO_EDGE`, `CLAMP_TO_BORDER`, or `MIRRORED_REPEAT`

When you change the texture type, filter, or wrapping mode, you get immediate feedback on the effect. As a result, you'll be able to compare filtering methods and wrapping modes easily.

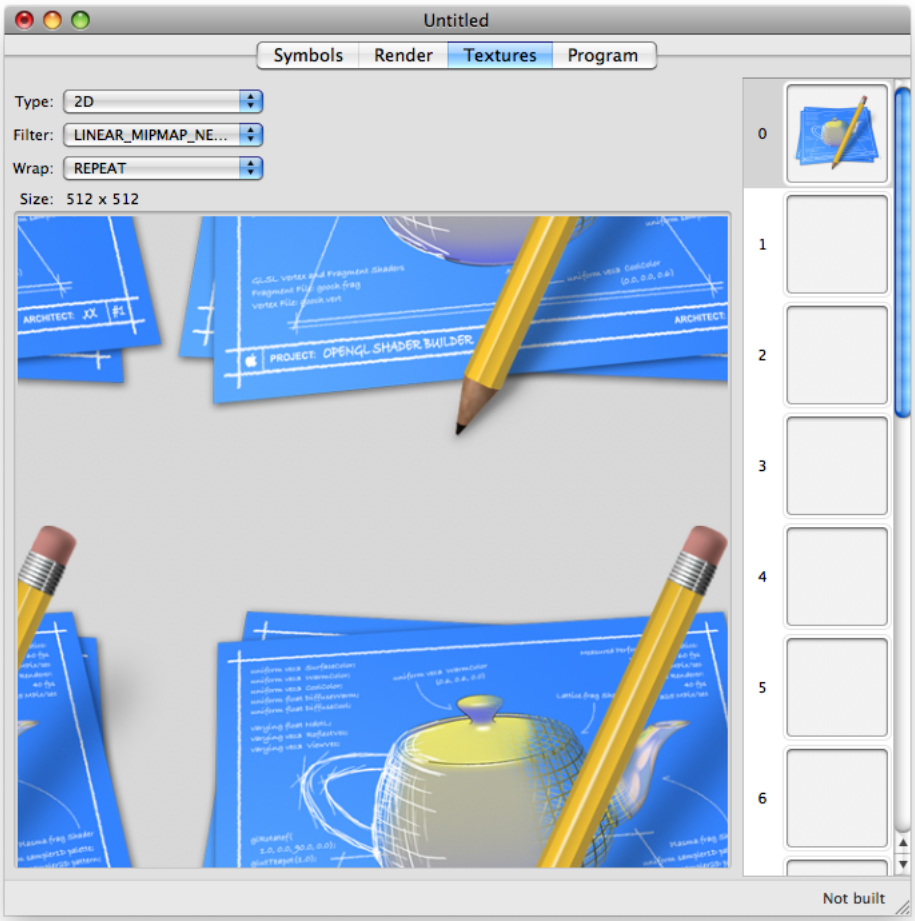
Figure 1-6 The Textures view



You can get an idea of how OpenGL maps a texture to an object by looking an an alternate view of the texture. To see an alternate view, double-click the texture that appears in the view on the left. The default view is a flat representation of the texture without the wrapping mode. The alternate view maps the texture in the dimension of its type (1D, 2D, 3D, Cube) and applies the filtering and wrapping modes.

Figure 1-7 (page 16) is the alternate view for the texture shown in Figure 1-6 (page 15). This view shows the repeating pattern caused by choosing the `REPEAT` wrapping mode.

Figure 1-7 The Textures alternate view

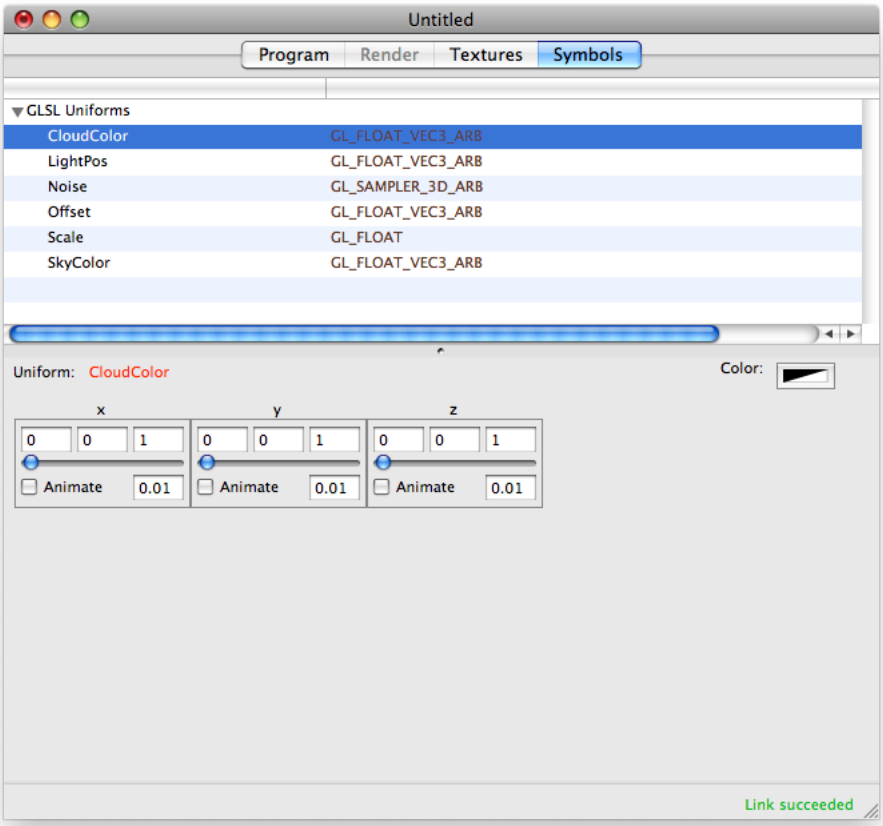


For more details on working with textures, see "Adding Textures" (page 20) and "Using Alternate Texture Views" (page 21).

Symbols View

After you add shaders to the Program view, you can view its uniform variables in the Symbols view, as shown in Figure 1-8 (page 17). You can modify and animate GLSL uniform variables and ARB environment and local parameters.

Figure 1-8 The Symbols view



For more information, see "Modifying Uniform Variables" (page 23).

Building Shaders

This chapter explains how to use OpenGL Shader Builder to set up projects, add shader code, add resources, and modify variables. Before reading this chapter, you should already be familiar with OpenGL and know how to write at least one of the following:

- ARB fragment and vertex programs. See the OpenGL extensions GL_ARB_fragment_program and GL_ARB_vertex_program.
- GLSL fragment and vertex shaders. See OpenGL Shading Language (PDF).

OpenGL Shader Builder also supports geometry shaders, a recent addition to the OpenGL specification (see the OpenGL Extension GL_NV_geometry_shader4). Geometry shaders are not supported on all graphics cards. But because the Apple software renderer steps in as a fallback when necessary, you can use OpenGL Shader Builder to develop them.

Creating and Saving Projects

A **project** is the set of resources that make up one program—vertex, fragment, and geometry source files, and textures. As with any development environment, you can name and save projects. You can also have more than one project open at a time.

When you launch OpenGL Shader Builder, it opens to an empty, untitled project. To save the project, choose File > Save Project and enter a project name. A project can contain as many source files and textures as you'd like. Using the checkboxes in the file list, you can select which source files to make active.

Creating and Saving a Layout

A **layout** specifies the location and number of windows that you want OpenGL Shader Builder to provide when you open new and existing projects.

To create and save a layout:

1. Launch OpenGL Shader Builder.
2. Double-click each tab whose view you want to open in a separate window.
3. Arrange the windows to suit your preference.
4. Choose Window > Save Layout.

Whenever you launch OpenGL Shader Builder, it automatically sets up the environment for you using your preferred layout.

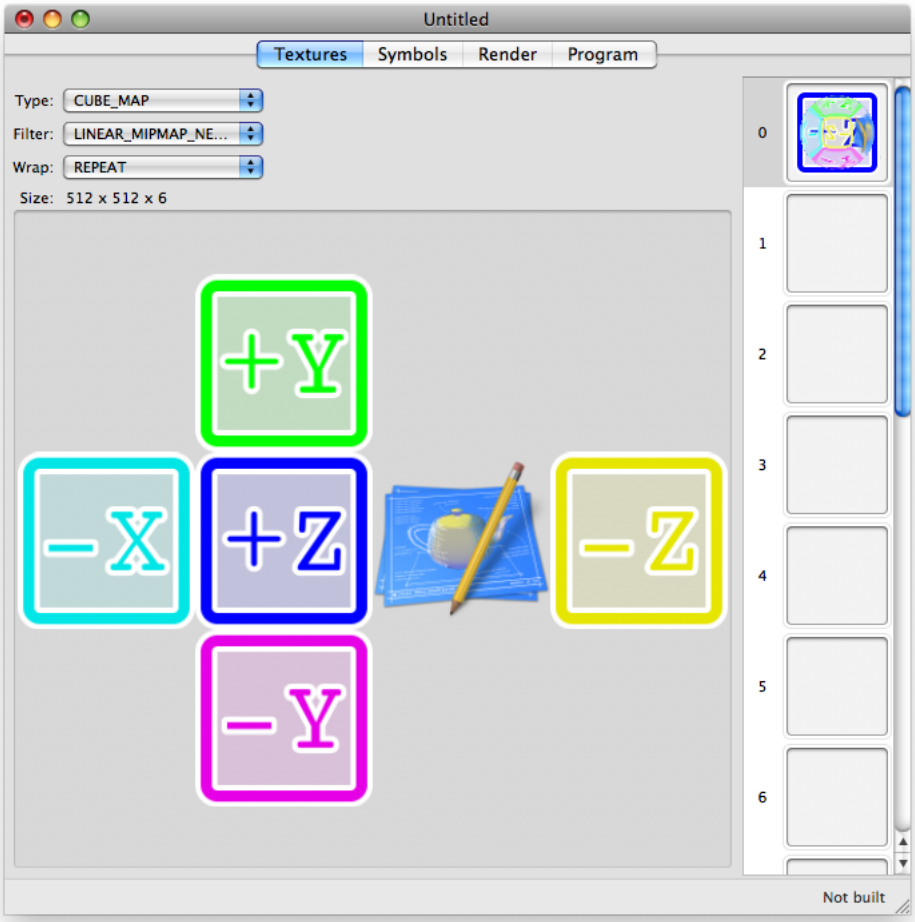
Adding Textures

To add a texture to the Textures view, drag the texture file to an image well. You can add any of these texture targets: `1D`, `2D`, `RECTANGLE`, `SHADOW_1D`, `SHADOW_2D`, and `SHADOW_RECTANGLE`.

To add a `3D` texture, you drag all the necessary files to an image well. The number of images in this texture target must be a power of 2 (2, 4, 8, 16, 32, and so on). Otherwise, OpenGL Shader Builder inserts a default image in the *z* direction.

You can also drag `CUBE_MAP` textures to an image well, but because this texture targets require more than one file, you'll first need to name the files so that OpenGL Shader Builder can place them properly. Figure 2-1 (page 20) shows the layout that OpenGL Shader Builder uses for cube maps.

Figure 2-1 Cube face layout for a cube map



Tip: When adding cube maps or textures, after you add one image file to the image well, you can then drag other images to the individual faces that OpenGL Shader Builder automatically generates.

To add a cube map :

1. Name each texture file using a convention that specifies the location within the cube map or 3D texture.

   For cube maps, you can use any of the conventions listed in Table 2-1 (page 21). For example, if the base filename for a cube map is `mycube`, you could name the texture files: `mycube_back`, `mycube_down`, `mycube_forward`, `mycube_left`, `mycube_right`, and `mycube_up`.

2. In the Finder, select the texture files and drag them to an image well.

Tip: If you name the cube map files correctly and place them in the same directory, you need to drag only one file in that directory to the view on the left side. OpenGL Shader Builder then reads all files in entire directory and places all the images for you.

Table 2-1 Naming conventions that map texture files to cube faces in a cube map

X+ face	X– face	Y+ face	Y– face	Z+ face	Z– face
posx	negx	posy	negy	posz	negz
xpos	xneg	ypos	yneg	zpos	zneg
right	left	top	down	front	back
rt	lf	up	dn	ft	bk
+x	-x	+y	-y	+z	-z

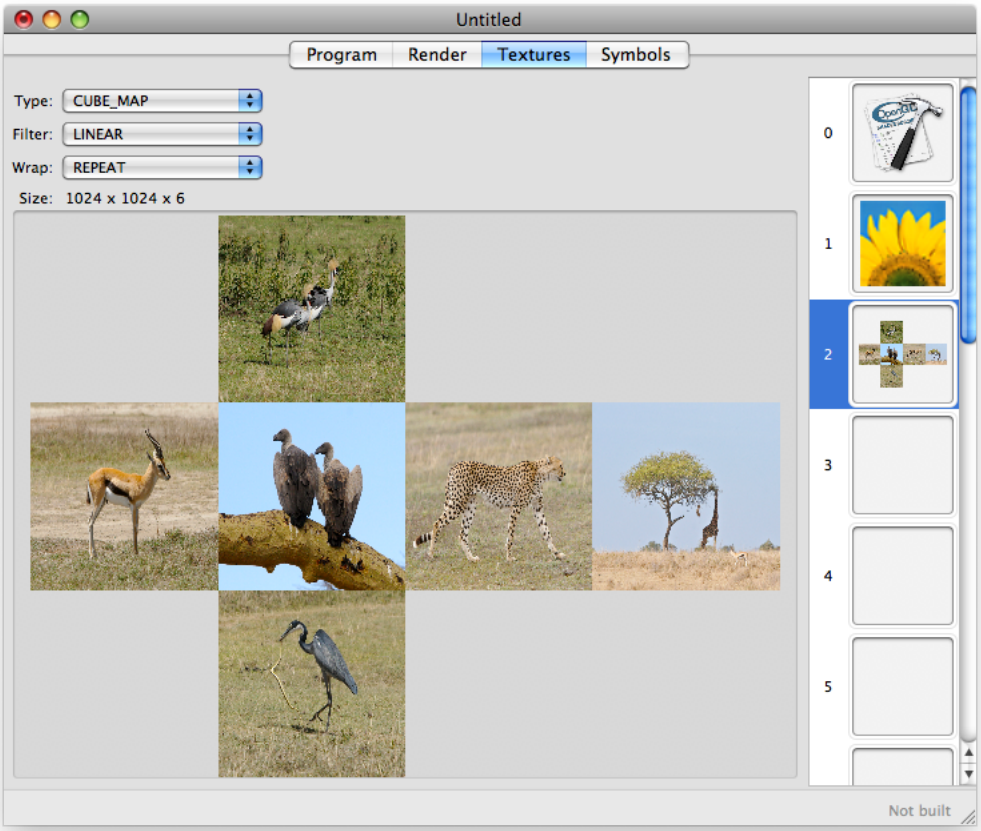
Using Alternate Texture Views

OpenGL Shader Builder provides two ways for you to view textures. The default view shows the texture data simply as a flat representation. The alternate view shows how the texture appears when applied to a target, using the filter and wrap modes that you choose. The alternate view is especially useful if you are unsure of how a particular filter or wrap mode will affect the outcome. You might also find the alternate view helpful to visualize 3D and cube maps.

The alternate view is particularly useful for cube maps. The default view in Figure 2-2 (page 22) shows the "unfolded" cube, while the alternate view in Figure 2-3 (page 23) projects the cube faces in three dimensions.

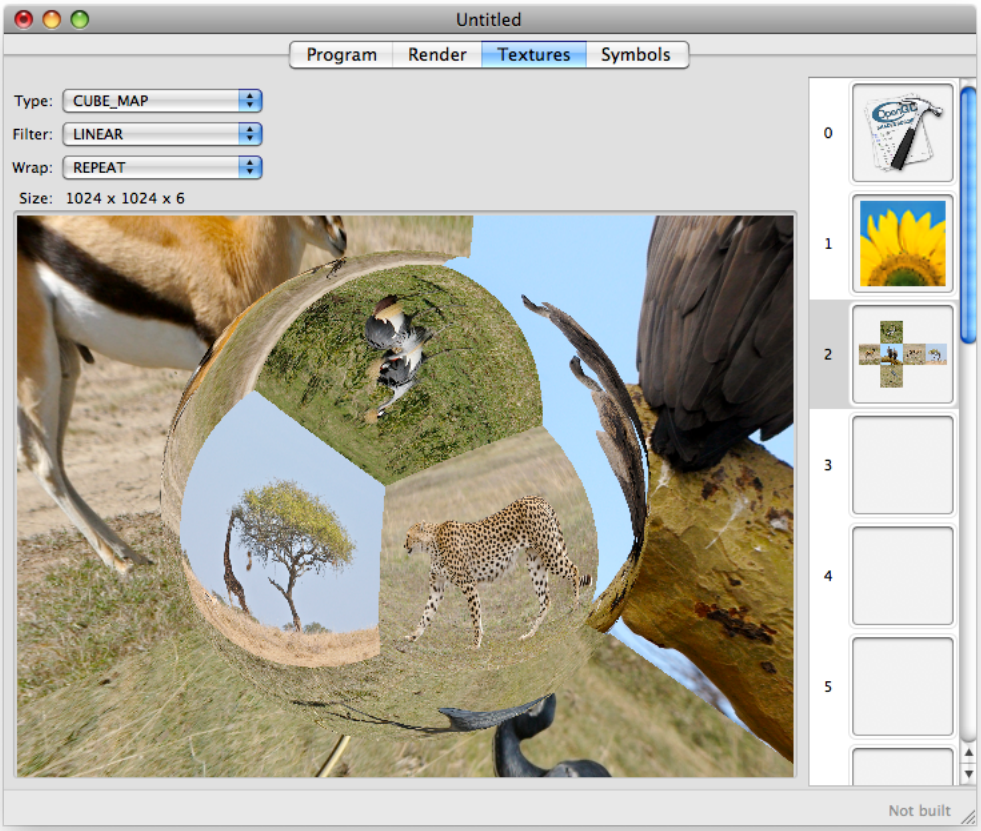
To see the alternate view, double-click the texture.

Figure 2-2 The default view for a cube map



If you have not supplied multiple files for a cube map or 3D texture or if you've not used a location-based naming convention (see "Adding Textures" (page 20)), you'll notice rectangles with a number or letters in them when you switch to alternate view.

Figure 2-3 The alternate view for a cube map

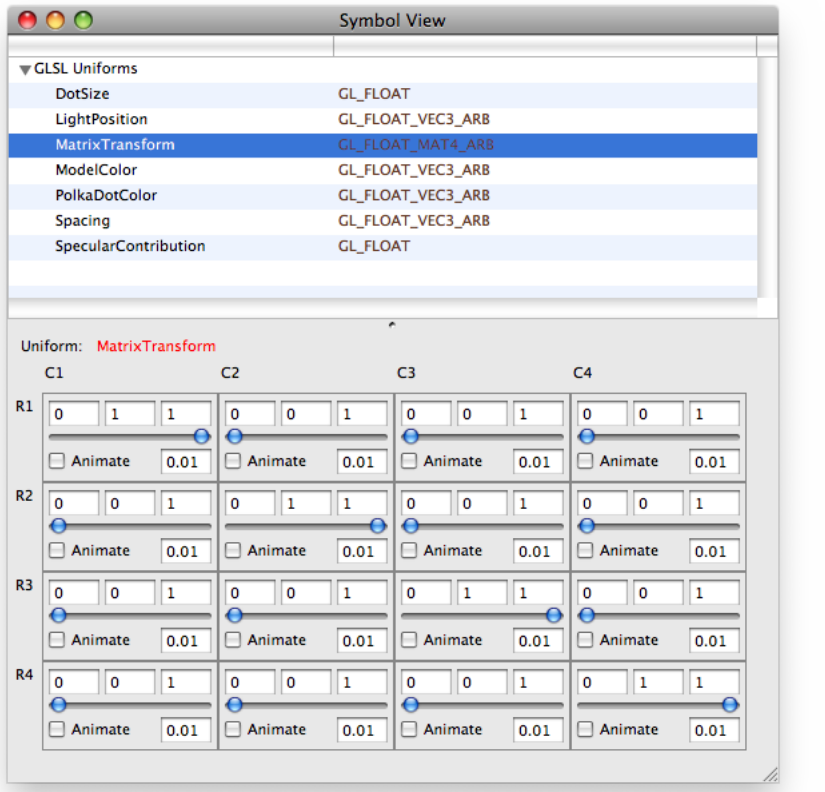


Modifying Uniform Variables

OpenGL Shader Builder automatically lists uniform variables from your source code in the Symbols view. How these uniform variable are displayed depend on the type of program. GLSL shaders use a shared symbol table for a single program object, so you'll see the uniform variables appear in a single list . In contrast, ARB fragment and vertex programs have their own symbol table per pipeline state which are separated by local and environment variables per stage. Therefore ARB local and environment variables appear in separate lists.

The controls for a variable reflect that variable's data type, no matter how complex the type (see Figure 2-4 (page 24)). You can use the controls to manually change a value, or you can select Animate to automatically vary a uniform from one value to another. No matter which you choose, you will get immediate feedback by looking at the Render view as long as the auto compile and auto link options are enabled.

Figure 2-4 Controls for a matrix structure



Building Shaders

To build a shader and make sure it runs correctly, follow these steps:

1. Open OpenGL Shader Builder.
2. Add shader source code.

   If you've already written the shader source files, click Add Shaders. Then, navigate to the files you want to add to the file list and choose them.

   If you want to enter the source code, choose File > New and then choose the type of shader you want to write. A source code file opens in its own window. You can modify the default code provided in the template.

   Tip: You can drag existing source files to the program list.

3. Check the Link Log to make sure the programs linked successfully. You might also want to check the link results.

By default, OpenGL Shader Builder has automatic linking enabled. If you disabled this feature, you'll need to enable it or click the Link button.

If the link fails, check to make sure that you added all necessary source files. For example, if you add a fragment or geometry program without adding the associated vertex program, linking fails.

4. In the Textures view, add any textures that are appropriate for your shader.

   You'll most likely want to replace the default texture.

5. In the Symbol view, animate one or more of the uniform variables.

6. In the Render view, choose a geometry from the pop-up menu.

   You can drag the pointer to move the rendered image.

After your shader is running, you may want to benchmark its performance.

Checking Shader Performance

It's a good idea to check the frame rate of your shader before and after you make adjustments to the code. You can measure the frame rate by following these steps:

1. In the Render view, click Benchmark.

   A benchmark window opens.

2. Enter the number of seconds for the benchmark test. Then, click Run.

   Note the elapsed time and the frames per second.

If you find the frame rate is much lower than you'd like, check to see if you are performing:

- Tasks inside a loop, such as setting state, that should really be performed outside the loop
- Complex calculations, such as arcsin. You can improve performance by pre-calculating results and storing them in a texture. Then, when you need a result, perform a texture look-up operation instead of the complex calculation.

After you are certain that your shader performs well in isolation, you should add it to your OpenGL application. Then, use OpenGL Profiler to make sure that your shader and the surrounding OpenGL application run as optimally as possible.

For information identifying and solving performance issues with OpenGL applications, see *OpenGL Profiler User Guide*.

Troubleshooting Errors

Shaders can have validation errors for a number of reasons. You'll want to be familiar with the OpenGL extensions that apply to the type of GPU program you are writing, because each extension outlines the conditions that can cause such errors. This section provides guidelines for a few of the common errors.

- Make sure that your code sets uniform variables for samplers at validation time.
- If your code depends on support for certain features, make sure you add a directive to require the appropriate extension, otherwise your code won't parse.
- If a geometry shader fails, check to see whether any of the following apply:
  - The geometry program has no associated vertex program for supplying varying variables.
  - You used the wrong input or output types. Geometry shaders use fixed input and output primitive types.
  - The value of `GEOMETRY_VERTICES_OUT_EXT` is `0`.

You can sometimes troubleshoot errors by examining well-written code and comparing it to our own. You may want to look at the following:

- The shaders in `/Developer/Examples/OpenGL`, which are available after installing the Developer Tools that come with Mac OS X.
- The *GLSLShowpiece* and other sample code that's available through the ADC Reference Library.
- Sample code that's available on http://www.opengl.org/.

Document Revision History

This table describes the changes to *OpenGL Shader Builder User Guide*.

Date	Notes
2008-06-23	New document that explains how to use OpenGL Shader Builder to develop and test GPU programs.

Document Revision History