

Chapter 12

OpenGL Tools

One of Apple's goals with Mac OS X was to make it the premier OpenGL platform for both users and developers. To make Mac OS X the best OpenGL platform for developers, Apple includes OpenGL developer tools as part of the Xcode Tools. This chapter explains how to use these tools.

OpenGL Profiler

OpenGL programs demand high performance to be usable, making these programs the most likely to require profiling. But if you profile an OpenGL program with Instruments or Shark, you'll notice a problem. The OpenGL function calls the program makes don't appear in the profile. Use OpenGL Profiler to profile your program's OpenGL function calls.

OpenGL Profiler works like a normal profiler, but it focuses exclusively on OpenGL function calls. For each OpenGL function call your program makes, OpenGL Profiler reports the number of times your program called the function and the amount of time spent in the function. In addition to profiling, OpenGL Profiler helps you debug your OpenGL programs. You can set breakpoints, run scripts of OpenGL commands, and view the contents of OpenGL's buffers.

Choosing a Program to Profile

When you launch OpenGL Profiler, the main window (see Figure 12.1) opens. Your first step is choosing a program to profile. How you choose a program to profile depends on whether or not the program you want to profile is currently running. If the program is running, select the Attach to application radio button. A list of all running applications appears in the window. Select the program you want to profile from the list.

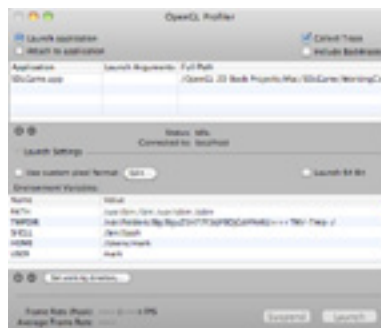


Figure 12.1

OpenGL Profiler launch window

Setting Environment Variables

To add an environment variable, click the + button below the environment variable list. Enter the variable's name in the Name column and the variable's value in the Value column.

Setting Breakpoints

To be able to perform certain tasks, such as viewing the contents of OpenGL buffers and resources, you have to set breakpoints in OpenGL Profiler. Breakpoints in OpenGL profiler work similarly to breakpoints in Xcode's debugger. They pause your program. The main difference is in OpenGL Profiler you set a breakpoint when your program calls an OpenGL function, not when it reaches a particular line of code.

Choose Views > Breakpoints to open the breakpoints window, which you can see in Figure 12.3. You can set a breakpoint at any OpenGL function call and at any Core OpenGL (CGL) function call. CGL is the way Mac OS X programs access OpenGL. If you use the Cocoa OpenGL classes, you're indirectly using CGL; the Cocoa classes sit on top of CGL.

On the left side of the window, you'll see an alphabetical list of function calls with three columns: Before, After, and Execute. All the functions initially have the Execute column selected, which means the functions execute normally. Disabling the Execute column for a function means OpenGL Profiler skips the function when it finds a call to that function in your program. Core OpenGL functions must execute so OpenGL Profiler prohibits you from disabling Execute for those functions.

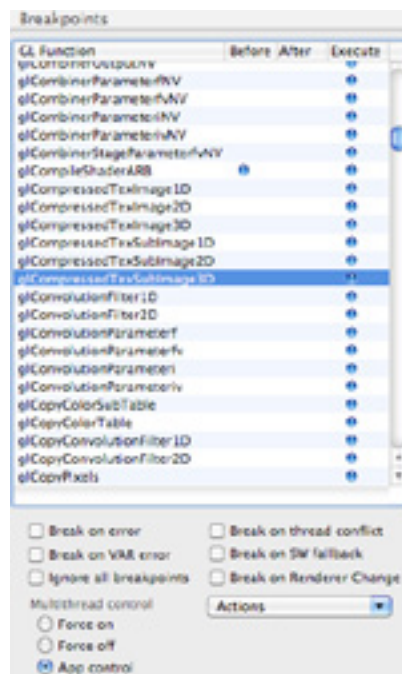


Figure 12.3

Breakpoints window

4 Chapter 12: OpenGL Tools

You can set a breakpoint before an OpenGL function call, after, or both. Setting a breakpoint before and after a function lets you see the function's effects on OpenGL's buffers. Click the Before or After column to set a breakpoint on a function. A blue dot signals that you set a breakpoint.

In addition to setting breakpoints for individual functions, you can tell OpenGL Profiler to pause execution when your program commits an OpenGL error. Below the function list is a series of checkboxes to break on various types of OpenGL errors. You can break on the following errors as well as temporarily ignore breakpoints:

- Any OpenGL error.
- A VAR error, which occurs when there is a problem using the vertex array range extension.
- A thread conflict.
- SW fallback, which occurs when a graphics card does not support a particular feature and OpenGL has to fall back to its software implementation.
- Renderer change.

Multithread Control

If you're running Mac OS X 10.5 or later, the OpenGL framework can do its work in a separate thread, which can improve the performance of OpenGL applications on Macs with multiple CPU cores. The multithread control radio buttons let you control OpenGL's multithreading. You can turn multithreading on and off (turning it off makes debugging easier), or you can choose to control it from your application. To turn on multithreading from your application, call the function `CGLEnable()` and supply the constant `kCGLCEMPengine` as the second argument.

Breakpoint Actions

Below the checkboxes is the Actions pop-up menu. Use the Actions pop-up menu to set and remove breakpoints for all functions as well as attach scripts. Refer to the section "Scripts Window" later in this chapter for more information on attaching scripts.

Profiling Your Program

Before you start to profile, you must tell OpenGL Profiler what to record as your program runs. There are two checkboxes in the upper right corner of the main window. Selecting the Collect Trace checkbox tells OpenGL Profiler to record a list of all the OpenGL function calls your program makes as it runs.

Selecting the Include backtraces checkbox tells OpenGL Profiler to record the call stack. Recording the call stack lets you see what functions in your program made the OpenGL calls. Your application will run slower in OpenGL Profiler if you record the call stack.

Click the Launch button to launch your program. The program continues to run until it hits a breakpoint or you click the Suspend button. Click the Resume button to continue profiling. Clicking the Force Quit button quits the program you're profiling.

Viewing the Profiling Data

The main OpenGL Profiler window doesn't provide much profiling data about your program. It tells you the current and peak frame rates. To see more data about your program's performance, you must use OpenGL Profiler's auxiliary windows, which you can open from the Views menu.

Trace Window

The trace window shows every OpenGL function call your program made. The window lists the function calls in the order your program made them, one function call per line. By examining the trace you can find unnecessary function calls, improving your program's performance. Looking at the function trace also helps you debug your program. You can learn your program is passing bad data to OpenGL functions and discover missing functions, functions your program should be calling but isn't calling.

The trace window has controls to show more data in the trace window. Selecting the Line #s checkbox adds a line number to each listing. Selecting the Context checkbox adds the OpenGL context to each listing. Selecting the Timing checkbox adds the amount of time your program spent in the function. The trace window reports the time in microseconds. Selecting the Verbose checkbox adds the amount of time your program spent in each function along with two timestamps. The first timestamp is when your program entered the function, and the second timestamp is when your program exited the function. Selecting the RenderID checkbox adds the renderer's memory address to the trace window

Clicking the Call Stack button opens the call stack drawer. Clicking a function's link from the trace window displays the function's call stack in the call stack drawer. The Include Backtraces checkbox must be selected in the OpenGL Profiler window for function names to appear in the call stack drawer. If you don't include backtraces, the call stack drawer shows only the memory address of each function.

Statistics Window

The statistics window, shown in Figure 12.4, shows the profiling statistics of the OpenGL functions your program calls. It tells you the total amount of time, measured in microseconds, your program spent in OpenGL function calls along with the percentage of time your program spent in OpenGL. For each OpenGL function your program calls, the statistics window tells you the following information:

- The number of times your program called the function.
- The total time, measured in microseconds, your program spent in the function.
- The average time, measured in microseconds, your program spent in the function.
- The percentage of the OpenGL time your program spent in the function.
- The percentage of time your program spent in the function.

When OpenGL Profiler samples the OpenGL functions in your program, it breaks the samples into slices, 25 slices by default. Use OpenGL Profiler's preferences to change the number of slices. If you sample your program for a short period of time, there may be fewer than 25 slices. The statistics for each slice build on the previous slice. If your program calls a function 20 times during the first slice and 15 times during the second slice, when you look at slice 2, it reports 35 function calls. Move slice by slice to see how your program performs over time.

Buffers Window

The buffers window lets you see the contents of the alpha, back, stencil, and depth buffers. To be able to view a particular buffer, you must be using it in your program. If you don't use the stencil buffer, there's not much point in being able to look at it with OpenGL Profiler. You can view the buffers only when OpenGL Profiler stops at a breakpoint so make sure you set some breakpoints if you want to examine buffers.

GL Function	# of Calls	Total Time (usec)	Avg Time (usec)	% GL Time	% App Time
glGenTextures	4	2	0.64	0.00	0.00
glClearColor	4	7	1.78	0.01	0.00
glEnableVertexAttribArray	1,347	261	0.19	0.19	0.00
glVertexAttribPointer	1,347	263	0.19	0.26	0.00
glDrawElements	1	7	7.52	0.01	0.00
glDrawElementsInstanced	1,347	316	0.23	0.26	0.00
glDrawElementsInstanced	1,347	315	0.23	0.24	0.00
glDrawElementsInstanced	1,347	875	0.65	0.83	0.01
glClearColor	1	0	0.01	0.00	0.00
glTexCoord2f	8,964	248	0.03	0.34	0.01
glTexEnvf	2,141	440	0.20	0.33	0.01

Total elapsed GL function time: 11891.89 usec
 Estimated % time in GL: 1.55%
 Multithreaded: OFF
 Show slice: 1 of 25 Context ID: All Contexts

Figure 12.4

Statistics window

Resources Window

The resources window lets you examine your application's textures, programs, shaders, FBOs (Frame Buffer Objects), renderbuffers, VBOs (Vertex Buffer Objects), and VAOs (Vertex Array Objects) your application uses. You must set breakpoints if you want to examine your OpenGL program's resources. The resources are visible only when OpenGL Profiler stops at a breakpoint.

There are tabs at the top of the window to look at a particular resource category. When you choose a category, say textures, that category's instances fill the left side of the resources window. Select an instance to examine it.

For FBOs, VBOs, and VAOs, OpenGL Profiler shows a list of attached objects.

Textures and Renderbuffers

Textures and renderbuffers are both images so OpenGL Profiler displays similar information for each. Select a texture or renderbuffer from the list running along the left side of the resources window. In the center of the resources window is the image itself. Above the texture or renderbuffer is information about it like its size, target (1, 2, or 3-dimensional image), and format. Below the image are options for displaying the texture or renderbuffer in the resources window. For textures click the Show Texture Viewer Settings button to see the options. These options include the following:

- Zoom level, which lets you get a closer look at the image.
- Source blend mode.
- Destination blend mode.
- Background color.
- Background opacity.

The blend modes determine how the incoming fragment's color blends with the texture environment's color to create a final color for the pixel.

For textures you have the added option of specifying a mipmap level. Mipmaps are smaller versions of a texture used to provide level of detail. When the object moves farther away from the camera, OpenGL uses a smaller version of the texture. Cut the size of the texture in half until you reach a 1-by-1 pixel mipmap. If the base texture is 64-by-64 pixels, the mipmaps would be 32-by-32, 16-by-16, 8-by-8, 4-by-4, 2-by-2, and 1-by-1. With OpenGL Profiler you can view these smaller textures.

Programs and Shaders

Programs and shaders are programs that are meant to run on the graphics card instead of the CPU. The difference between programs and shaders is the language you use to write them. A program is lower-level, using a syntax that resembles assembly language. Shaders use GLSL, which has a syntax similar to C.

You can view the source and a log for each shader. You can modify the shader inside OpenGL Profiler and compile shaders as well.

Scripts Window

Use the scripts window, shown in Figure 12.5, to write scripts that execute when your program reaches a breakpoint. You can include any OpenGL function call in a script, but a script can contain only OpenGL function calls.

Click the + button to create a script. OpenGL Profiler adds it to the script list with the name Unnamed. Double-click the name to change the script's name. Use the text editor in the top half of the window to create the script.

There are two ways to run your script: manually and automatically. To run your script manually, your OpenGL application must be stopped at a breakpoint. Open the scripts window, select your script, and click the Execute button to run the script. If your script has any syntax errors, they appear in the log, which is below the text editor.

To run your script automatically when you reach a breakpoint,

1. Open the breakpoints window by choosing Views > Breakpoints.
2. Select a function from the list.
3. Choose Attach Script from the Actions pop-up menu.
4. A sheet opens with a list of scripts to attach. Choose a script from the list.
5. Decide when you want the script to run: before or after reaching the function.
6. Decide if you want your program to continue after executing the script.
7. Click the Attach button.

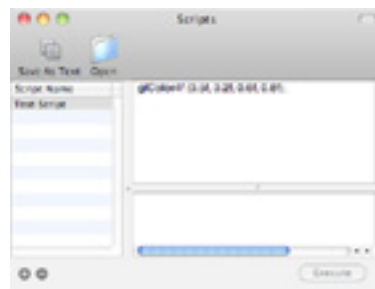


Figure 12.5

Scripts window

To detach a script from a function, select the function from the function list and choose Remove Script from the Actions pop-up menu.

Breakpoints Window

I covered the setting of breakpoints earlier, but there are two pieces of information that appear in the breakpoints window when your program reaches a breakpoint. The first piece of information is the call stack, the list of functions your program called leading up to the function where you set the breakpoint. The second piece is a snapshot of all the OpenGL state variables when you hit the breakpoint. The call stack and OpenGL state variables help tremendously during debugging. There are tabs to toggle between viewing the call stack and OpenGL state variables, even though they don't appear in the breakpoints window shown in Figure 12.3.

When viewing OpenGL state variables, there are checkboxes to show the changes that occurred since the last breakpoint and the changes from OpenGL's default state. The states that changed since the last breakpoint appear in magenta text, and the states that changed from the default state appear in blue text.

Pixel Format Window

The pixel format window displays the pixel format information for each OpenGL context your application uses. The information the pixel format window displays is the same information you can set when using custom pixel formats.

Messages Window

The messages window displays the log messages OpenGL Profiler generates as your program runs. These messages can help when debugging your application.

OpenGL Driver Monitor

OpenGL Driver Monitor collects statistics about your graphics card and its interaction with the CPU. It works at a lower level than OpenGL Profiler and works with the graphics card instead of a single application. Use OpenGL Driver Monitor to learn about things like how much video memory you're using, how much time the CPU spent waiting for the graphics card, the number of times the card swapped buffers, and the number of textures loaded on the card.

There are pop-up menus at the top of the window that let you set the minimum and maximum values the graph displays. The Linear/Log button controls the scale of the graph, the values running along the left side of the graph. The linear scale has even scales. The logarithmic scale uses uneven scales to fit as much data in the graph as possible. The data determines the scales in the logarithmic scale while the minimum and maximum values determine the scales in the linear scale. Look at Figure 12.7 to see the difference between the two scales.

Table View

If you don't want to fiddle with graph colors and just want to see the raw data, use the table view. Click the Table tab to switch from the graph to the table view. There's one column for each parameter in the watch list. After each sampling period, one second by default, OpenGL Driver Monitor adds a row to the table that contains the newly sampled data.

Renderer Info

Choosing Monitors > Renderer Info opens the renderer window, which displays information about the OpenGL renderers on your Mac. Every Mac has at least two renderers: the renderer for the graphics card and the software renderer, which appears as Generic in the window. OpenGL Driver Monitor reports lots of information for a renderer, including the following:

- The amount of video memory.
- The amount of texture memory.
- The OpenGL extensions it supports.
- The available sizes of the depth and stencil buffers.
- The available color depths for the color and accumulation buffers.



Figure 12.7

Linear (left) and logarithmic scales for the values 44, 76, 497216, 131, and 8671. See how the logarithmic scale shows the difference in the values 44, 76, and 131 while they appear identical in the linear scale.

OpenGL Shader Builder

OpenGL Shader Builder lets you create, test, and debug shaders. Shaders are programs that are meant to run on the graphics card instead of the CPU. The shader replaces a portion of the fixed-function OpenGL pipeline, giving you more control and flexibility in drawing a scene.

There are three types of shaders: vertex shaders, fragment shaders, and geometry shaders. A vertex shader gives you control over the transformation (converting a scene from world space to screen space) and lighting stage of the 3D graphics pipeline. Instead of feeding vertices to OpenGL and letting it take care of the transformation and lighting, each vertex goes through the vertex shader. Tasks you can perform in a vertex shader include:

- Vertex transformation, weighting, and blending
- Normal transformation, rescaling, and normalization
- Texture coordinate generation
- Texture coordinate transformation
- Lighting
- Color material application

A vertex shader doesn't have to do all the tasks in the list above, but if you don't perform one of the tasks in the vertex shader, OpenGL won't perform the task for you. If your vertex shader generates texture coordinates, OpenGL won't transform the texture coordinates for you. You'll have to transform them yourself.

Fragment shaders, sometimes referred to as pixel shaders, operate on fragments. You can think of a fragment as a pixel wannabe. The fragment contains data for a pixel, such as its primary color and position. OpenGL performs a series of tests on each fragment to determine whether or not the fragment becomes a pixel. Fragment shaders generate a final color and a depth value for the fragment, from which OpenGL performs its tests. Fragment shaders can perform the following tasks:

- Texture access.
- Texture application.
- Color sum, which creates a final color from the fragment's primary and secondary colors.
- Fog, blending a fog color with the fragment's color.
- Operations on interpolated values in the texture, color sum, and fog stages.

A fragment shader doesn't have to do all the tasks in the list above, but OpenGL won't perform the task for you if the fragment shader doesn't do it. If your fragment shader performs fog operations, OpenGL won't perform color summing for you. You'll have to do any color summing yourself.

A geometry shader generates geometry. It takes graphics primitives, like points, lines, and triangles as input and creates new primitives as output. Geometry shaders can be used to create procedural geometry, add detail to meshes, and break down complex polygons into groups of simpler polygons.

Creating a Project

To use OpenGL Shader Builder, you must create a project. A project contains shader source files and textures. A project can contain as many shaders as you want, but you might find testing individual shaders difficult if you have dozens of shaders in a project.

When you launch OpenGL Shader Builder, it creates an empty project for you. To create additional projects, choose File > New > Project. A window will open asking you if you want to create shader templates that will be added to the project. Select the checkboxes of the templates you want to create and click the Create button. If you want to create an empty project, click the No Templates button.

The project window initially shows the program view, which you can see in Figure 12.8. The program view has three sections. The top section has a button to add existing shaders to your project and a checkbox to automatically link your project. If your project has a geometry shader, there are controls that let you specify the types of input and output to the shader along with the number of vertices for the shader's output.

The center section contains a list of the shaders in your project. For each shader, there are three columns of information.

- A checkbox indicating whether or not the shader is active.
- The shader's name.
- The type of shader.

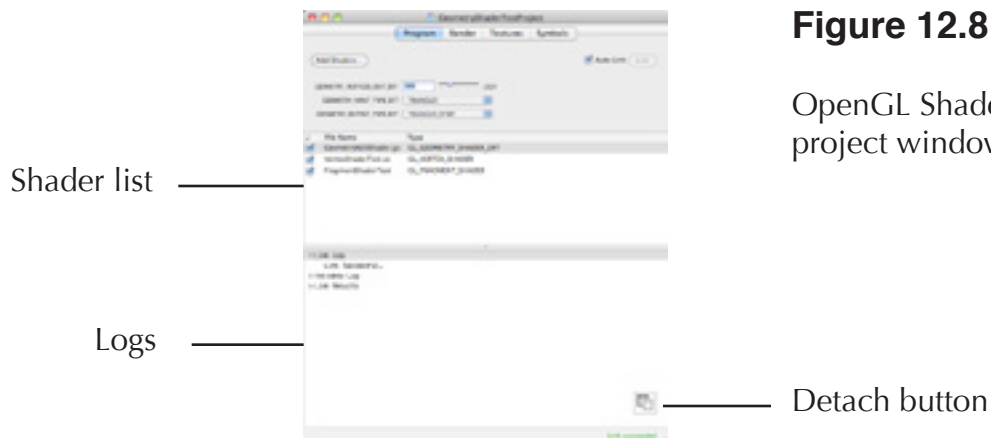


Figure 12.8

OpenGL Shader Builder
project window

The bottom section reports any linking and validation errors in your project. The link log shows any linking errors. The validation log shows any validation errors in your project's shaders. The link results section shows what the linker generated.

Adding Shaders

To add a blank shader to your project, choose File > New. There are five shaders you can create.

- GLSL Vertex Shader
- GLSL Fragment Shader
- GLSL Geometry Shader
- ARB Vertex Program
- ARB Fragment Program

When you add a new shader, it initially has the name Untitled. Save the shader to name it.

If you have shaders you've written previously that you want to use in the project, click the Add Shaders button to add them.

There are two languages you can use to write shaders: GLSL and ARB. GLSL has a C-like syntax, and ARB has an assembly language syntax. GLSL is better for writing large shaders. ARB's advantage is it's older, which means more Macs support it. Apple added hardware GLSL support in Mac OS X 10.4.3. If you're supporting earlier versions of Mac OS X, you should use ARB. Apple's GLSL support isn't as big a problem as it seems. Every Intel Mac supports GLSL, and most Power PC Macs capable of running Mac OS X 10.5 support GLSL.

A geometry shader requires a vertex shader to accompany it. If you add a geometry shader to a project, make sure the project also has a vertex shader. Geometry shaders let you specify the input type, the output type, and the number of vertices the shader outputs.

Writing a Shader

Double-click a shader in the Program view to open it in an editor. Use the editor to write the shader. At the bottom of the editor is a compile log that reports any compiler errors in your shader.

OpenGL Shader Builder provides constant feedback, letting you know when you make a syntax error. The constant feedback can be annoying because OpenGL Shader Builder reports errors before you can finish a line of code. Finish typing your shader's code before looking for syntax errors.

Deselecting the Auto Compile checkbox in the editor turns off the constant feedback and activates the Compile button. Click the Compile button to manually compile the shader.

Adding Textures

Each project comes with a texture of that can be applied to models for testing purposes. That texture's image is OpenGL Shader Builder's application icon. If you want to use your own images as textures in OpenGL Shader Builder, click the Textures tab in the project window. A list of texture units runs along the left side of the window with an image well for each unit. Drag the image to the image well of the texture unit you want to use. If your shader uses texture unit 2, drag the image to texture unit 2.

For your images to appear on the models in OpenGL Shader Builder, your shader must use the texture unit that you dragged the image to. If you drag an image to texture unit 3, but your shader doesn't do anything with texture unit 3, your image is not going to appear on any of the OpenGL Shader Builder models.

Looking at Variables

Click the Symbols tab to view the variables in your project. You can examine and modify uniform variables in GLSL. Uniform variables are variables an OpenGL program sends to a shader, and they are read-only in a GLSL shader. OpenGL Shader Builder lets you experiment with the values of uniform variables.

What you can set for a GLSL uniform variable depends on the variable's type. A simple variable like `sampler2D` has one component: `x`. A 4-by-4 matrix has 16 components: 4 rows multiplied by 4 columns.

For ARB vertex and fragment programs you can examine and modify environment and local parameters. Each parameter represents a color and has four components.

- `x` is the red component.
- `y` is the green component.
- `z` is the blue component.
- `w` is the alpha component.

Each component in a GLSL or ARB shader has three text fields, which you can see in Figure 12.9. Use the left text field to set the minimum value, which is initially 0. Use the right text field to set the maximum value, which is initially 1. Acceptable minimum and maximum values depend on the component. Use the center text field to specify the current value of the component. Dragging the slider also changes the current value.

If you select the Animate checkbox, the current value of the component constantly changes. Use the text field next to the checkbox to specify how much the value should change. The initial change value is .01. If you keep the initial values, the current value will change from .01, .02, .03, and so on all the way to 1, then go down to .99, .98, .97, all the way back to 0. After selecting the Animate checkbox you may want to click the Render tab to see the effects of changing the component's value.

Compiling a Project

OpenGL Shader Builder is initially set up to automatically compile your shaders. At the bottom of each shader window is the compile log. It will report any errors in your shader. The error reporting is constant, which means errors will appear in the compile log as you're typing your code. The errors will go away when you finish the line of code, assuming there are no errors in the line you finished.

If you don't like the automatic compilation, deselect the Auto Compile checkbox in the shader window toolbar. Click the Compile button to compile the shader.

OpenGL is also initially set to automatically link your project. If you don't want to automatically link your project, deselect the Auto Link checkbox in the program view. Click the Link button to link the project.

Testing a Shader

After writing a shader, you should test it to make sure it works. OpenGL Shader Builder simplifies shader testing. Click the Render tab in the project window to see the shader applied to a model. There is a pop-up menu beneath the view to choose a model.

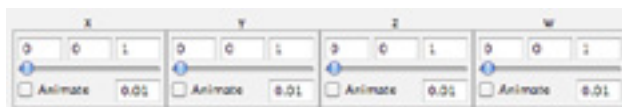


Figure 12.9

Variable component text fields

Benchmarking

Benchmarking measures how fast the project runs. To benchmark your project, click the Render tab in the project window. Click the Benchmark button to open the Benchmark window. Enter the amount of time you want to benchmark in the Run Time text field. Click the Run button. After the amount of time you entered passes, the Benchmark window tells you the frames per second.

Window Layouts

Initially OpenGL Shader has one window for the project with four tabs: Program, Render, Textures, and Symbols. You might want separate windows open, such as having the Render window open so you can see the changes you make.

In the lower right corner of the project window is the Detach button. Clicking the Detach button creates a separate window of the current tab. Closing the separate window reattaches it to the main project window. The Render, Textures, and Symbols views have a Reattach button in the lower right corner. Clicking the button reattaches the window to the main project window.

Using Your Shaders in an OpenGL Program

You've written your shaders, tested them in OpenGL Shader Builder, and they ran perfectly. How do you get the shaders into your OpenGL program?

Assuming you're using Xcode, add the shaders to your project. You don't have to add the OpenGL Shader Builder project file. Just add the shaders. Xcode can't compile shaders. You want Xcode to copy the shaders to the application bundle when it builds your project, just like graphics and audio files. The easiest way to ensure this is to add the shaders to the Resources folder in the Groups and Files list. Select the Resources folder and right-click. Choose Add > Existing Files.

After adding the shaders to your project, you have two coding tasks in your OpenGL program. The first task is to create the shader. The second task is to create a program object and attach the shader to it. I'm going to use the OpenGL 2.0 syntax. If you're using ARB extensions to support older graphics cards, the syntax may differ slightly.

Creating a Shader

Creating a shader requires the following coding tasks:

1. Create a shader object by calling `glCreateShader()`.
2. Load the shader from disk.
3. Read the shader source code by calling `glShaderSource()`.
4. Compile the shader by calling `glCompileShader()`.

Creating a Shader Object

Use the function `glCreateShader()` to create a shader object. The function `glCreateShader()` returns an integer value. Supply one of the following values:

- `GL_VERTEX_SHADER` for a vertex shader.
- `GL_FRAGMENT_SHADER` for a fragment shader.
- `GL_GEOMETRY_SHADER` for a geometry shader.

The following code creates a vertex shader:

```
GLuint shader;  
shader = glCreateShader(GL_VERTEX_SHADER);
```

Loading a Shader

How you load the shader from disk depends on what framework you're using. If you're using Cocoa, you would use Cocoa's `NSBundle` class to load the file from the application bundle. The following code demonstrates how to load a shader file from the application bundle:

```
NSString* filename;  
NSString* extension;  
  
NSBundle* appBundle = [NSBundle mainBundle];  
NSString* shaderFile = [appBundle pathForResource:filename  
ofType:extension];  
  
// Convert the NSString to a C String for OpenGL to use  
const GLchar* shaderFileGL;  
shaderFileGL = (const GLchar*)[shaderFile UTF8String];
```

Reading Shader Source

Call the function `glShaderSource()` to read the shader source. `glShaderSource()` takes four arguments:

- The shader index you created by calling `glCreateShader()`.
- The number of strings, which would be 1 to load a single shader.
- The string, which is the shader file you loaded from disk.
- The string length, which you can set to `NULL` if you don't care about the length.

The following code loads a single shader that was loaded from disk:

```
GLuint shader;
const GLchar* shaderFileGL;

glShaderSource(shader, 1, &shaderFileGL, NULL);
```

Compiling the Shader

To compile the shader, call the function `glCompileShader()`. Calling `glCompileShader()` is relatively easy. Supply the shader index you created by calling `glCreateShader()`.

```
glCompileShader(shader);
```

Creating a Program Object

Creating a program object takes four steps.

1. Create the program object by calling `glCreateProgram()`.
2. Attach the shader to the program object by calling `glAttachShader()`.
3. Link the program by calling `glLinkProgram()`.
4. Use the program object in your application by calling `glUseProgram()`.

The following code demonstrates the creation of a program object:

```
GLuint program;
GLuint shader;

program = glCreateProgram();
glAttachShader(program, shader);
glLinkProgram(program);
glUseProgram(program);
```

Cleanup

When you're done with a shader, there are three tasks to perform. The first task is to detach the shader from the program object by calling `glDetachShader()`. This function takes two arguments: the program object and the shader.

```
glDetachShader(program, shader);
```

The second task is to delete the shader by calling `glDeleteShader()`. Supply the shader to delete.

```
glDeleteShader(shader);
```

The final task is to delete the program object by calling `glDeleteProgram()`. Supply the program object to delete.

```
glDeleteProgram(program);
```