Step-Through Debugging of GLSL Shaders

Hilgart, Mark
School of Computer Science, DePaul University, Chicago, USA

Abstract

Programmers writing in the OpenGL Shading Language (GLSL) today rely on printf-style debugging methods and IDEs that set up test environments. However, as shaders get more complex, traditional step-through debugging on shaders running in their intended application becomes more useful.

This paper describes a rewriting method that enables step-through debugging on GLSL shaders that are running in the user's program. Full-screen debugging is used first to identify interesting pixels, then individual pixels are stepped through to completely understand their behavior. This rewriting method can be made transparent to the user, and serves as an alternative to planned library support.

I. Introduction

SGI brought hardware 3D acceleration to the scientific and visualization communities in 1983. Their first machine was the Iris 1000 and sold for tens of thousands of dollars. To allow programmers to use the hardware, SGI provided a library called IrisGL. This library would allow the CPU to send scene data and textures to the 3D hardware as well as control how it rendered the scene. This is also how OpenGL works, which was based on IrisGL. SGI to this day still only produces non-programmable 3D hardware, meaning the way the scene is rendered with their hardware can only be managed in a limited number of ways. Most notably the lighting model is fixed, and all materials tend to look similar. This also means non-3D uses are extremely limited.

The year 1983 also saw the world's first shader language written by Ken Perlin[16]. He devised it in order to quickly come up with new ways of generating noise. His programs would run once for each pixel on screen and compute the final pixel's color. The input for each pixel was surface position, normal, and material ID. This is recognized today as a fragment shader. When he presented it at SIGGRAPH, it was quickly recognized that it could be used more generally for all types of 3D scenes.

Pixar was working on films that were fully computer-generated in the mid-1980's. Possibly taking

inspiration from Perlin's presentations, they published the Renderman Shading Language (RSL) in 1988. RSL was designed to be very extensible and proved to be capable of this, as it is still used today. Pixar's Photo-Realistic Renderman (PRMan) uses RSL as its shading language, and is still used today to produce feature films including Pixar's Cars (2006). The first successful full-length feature film, Toy Story (1995), was started in 1991 and also used RSL.

Until 1996, SGI's expensive 3D hardware and Pixar's offline renderer PRMan characterized the computer graphics industry. In 1996, 3dfx developed the Voodoo Graphics chip and partnered with Original Equipment Manufacturers (OEMs) like Diamond to bring affordable hardware 3D acceleration to consumers[15]. Diamond's first product to use a 3dfx chip was Monster 3D. While Voodoo was meant to be used with 3dfx's proprietary API, Glide, a partial implementation of OpenGL called the miniGL driver was later developed to allow glQuake to run. This port of the very popular PC game Quake significantly raised the quality of graphics available to consumers and drove many Voodoo-based 3D card sales.

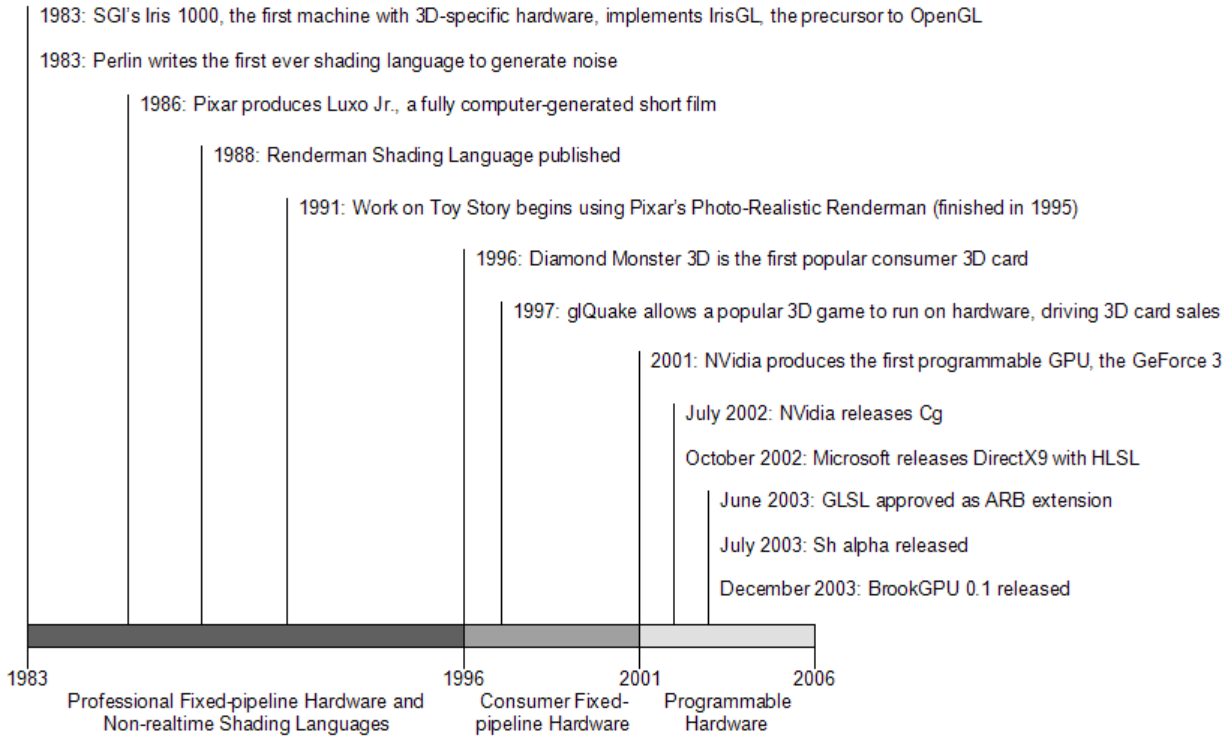

Figure 1. Timeline of shading language advances and related events.

Consumer hardware remained constrained to fixed-pipeline hardware until 2001. During the 1996-2001 period, this fixed hardware could not be used to accelerate scenes that used flexible shading languages like RSL. Instead, programmers writing for these fixed-function cards drew on experience gained writing programs to run on SGI's hardware. This meant that progress in flexible

shader languages continued only in non-realtime software implementations. Since the processing was still done on the CPU, there was no advantage to adapting non-3D applications to RSL.

NVidia released the first ever programmable GPU in 2001, the GeForce 3. Finally, languages like Perlin's could be implemented in hardware. The only language available at the GeForce 3's release to program it was assembly language. Nevertheless, it was immediately clear that the possibilities with this type of hardware were tremendous. While the card couldn't immediately handle the complexity of a movie like Toy Story, the limitations preventing the feat such as shader size and scene size would soon be lifted.

NVidia then developed "C for Graphics" (Cg), a C-like language for programmable GPUs such as the GeForce 3. This language allowed programmers to write at a more comfortable abstraction level. Later that year, Microsoft followed with HLSL which shipped with DirectX 9[14]. Finally, GLSL was approved as an OpenGL extension in 2003[11]. Cg, GLSL, and HLSL are all very similar languages, differing most notably in their interfaces to the host applications and their implementation models.

Cg and HLSL can be compiled either at runtime or compile time to assembly code, which is then passed to the driver to be compiled to GPU machine code. While Cg and HLSL compilers are readily available and their optimized assembly output can be examined, GLSL's compilers are hidden in the display driver and the assembly is not accessible. Even if it could be read, the manufacturers can change intermediate formats at any time, or even compile directly to machine code. As will be shown later, this significantly limits the available debugging methods.

Once shaders could be run in hardware, it was realized that the parallel architecture that was needed to run them efficiently could also be used to solve non-3D problems. In the same year GLSL was approved, two GPU programming language projects were started which aimed to make writing non-3D GPU applications easier: Sh[17] and BrookGPU[18]. Brook programs are compiled and run completely on the GPU, requiring the same host-GPU relationship that traditional 3D applications do. Brook is designed primarly for non-3D GPU applications. In contrast, Sh is designed for both 3D and non-3D applications, and is written as a C++ library. The library uses the capabilities provided by the 3D card to accelerate operations when possible. Both approaches abstract the GPU's capabilities further than Cg-type languages in order to speed development. They are still under active development and in 2006 are approaching non-beta releases.

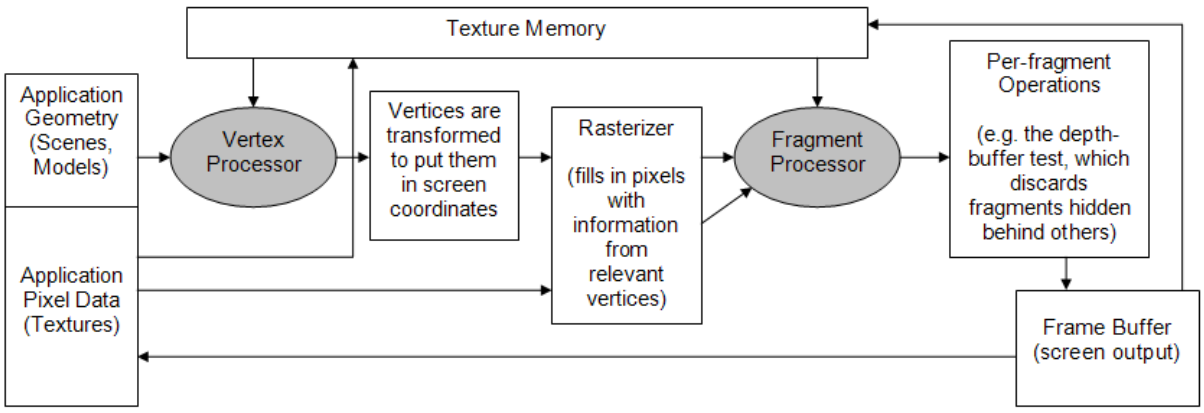

Figure 2. The OpenGL programmable pipeline.

Programmable GPUs are specialized in the way they run programs. Specifically, they are made up of two types of sub-processors which run the programs and have constraints on their inputs and outputs. This allows them to run much faster than CPUs for certain types of programs but slower for others.

Figure 2 is a diagram of the flow of data inside the GPU and between it and the CPU[11]. The host application runs on the CPU and is where all scene geometry is generated in the current version of OpenGL. The vertices of the geometry are fed to the vertex processor one by one, where the processor also has access to texture memory and non-vertex-specific variables from the host program. It then outputs a modified vertex, one by one.

The modified geometry is then transformed to screen coordinates and rasterized. This process fills in triangles, generating fragments from vertices. The vertices for each triangle are used to calculate each fragment's data. A fragment represents a square of the target space as a pixel does, but includes all other information necessary to generate the pixel in the frame buffer including depth, alpha, and texture coordinates. If a fragment is located directly over a vertex center, that fragment will get the values of that vertex. A fragment in the interior of a triangle will have its values interpolated by the values at the vertices.

The fragment processor is then run once for each input fragment, and it also has access to texture memory and non-fragment-specific host memory. The output of this processor is nearly finished pixels for the screen. They simply need to go through a chain of per-fragment operations before the final screen buffer pixel is output. One of these operations is the depth test, which discards fragments based on which is closest to the screen. The output can then be shown on screen or redirected to a texture, where it can be used as input to a later pass.

GLSL mirrors the GPU's architecture by defining two program types, vertex and fragment shaders, to correspond to the vertex and fragment processors on the GPU. This paper deals with fragment shader debugging. Vertex shader debugging may be possible using similar methods, but the visu-

alization and user interface will be different.

Many beginning shader writers are surprised at the lack of step-through debuggers for GLSL shaders. However after studying existing shaders, it becomes apparent that debuggers are not critical for the most common shaders, which often consist of a single function less than 50 lines long. Using printf-style debugging techniques along with IDEs is usually sufficient. The IDEs set up input textures, monitor texture output, and can control input variables and geometry.

Modern cards have lifted the 256 instruction limit to 65k. GPGPU techniques are pushing the limits of GLSL implementations with their growing complexity. Some examples are protein folding, graph flow simulations, and traditional image processing techniques. These new events are leading to much larger programs that make printf-style debugging more laborious and often insufficient. More powerful tools are needed.

II. Existing Debuggers

HLSL already has step-through debugging available via Microsoft's Visual Studio. It allows debugging of shaders in a UI very similar to traditional CPU debugging. The similarities include the ability to debug a running program instead of a separate test scene, a watch window showing variable state at the line being debugged, and line-by-line execution. This is possible by using a complete software implementation of Direct3D including the vertex and fragment pipelines. Microsoft has abandoned this debugger in Visual Studio 2005 and will be replacing it with PIX. This tool started in the X-Box SDK and is moving to the PC's DirectX SDK. It can capture DirectX call streams and some versions include a shader debugger similar to the one in Visual Studio 2003. It has an additional feature called pixel history debugging which allows the programmer to see what values each pixel receives including the final value.

NVidia's FX Composer and ATI's RenderMonkey are two IDEs with similar features which run shaders on test scenes instead of in the user's application. They both allow modifying the input geometry, textures, and uniform values. Uniform variables are sent once from the CPU for each pass of the shader and can include the modelview and projection matrices or material properties, but are completely user definable[12]. Their output is simply the fragment shader's output. FX Composer and RenderMonkey are useful for testing small shaders on many different test scenes and quickly changing inputs and seeing the results. This is possible because the entire shader can be read and understood in one sitting, and errors can be found using printf-style techniques. However as shaders grow larger, this type of tool becomes inadequate.

There are many OpenGL debuggers which don't examine internal shader variable state, but can show other OpenGL state and calls which can be helpful in shader debugging. What sets these debuggers apart from the above IDEs is that they run on existing programs, allowing the tester to see accurately what the shaders will look like in their intended settings. This also saves time that would be spent setting up test environments. OpenGL state, which can be displayed, is similar to

uniform inputs to the programmable shaders, but are fixed by the OpenGL specification since they are used in the fixed pipeline. Examples include matrices, light position, ambient/diffuse/specular material properties, and texture stage combination operations. BuGLe[3], GLSurveyor[4], gDEBugger[5], and GLTrace[6] are some debuggers that show this type of information. GLSurveyor and GLTrace output text logs of calls and other GL state information. BuGLe and gDEBugger are graphical and provide tools like function call statistics, a texture browser, and a source code viewer in addition to the call logs and state information.

GLIntercept[8], based on GLTrace, also displays OpenGL state in running user programs like the above debuggers, but additionally it provides an editor to modify shaders as they're running on the card. The modified shader is recompiled and replaced at runtime. GLIntercept also adds several other debugging features to GLTrace including resource leak detection, performance logging, free-look camera, and OpenGL extension override/disable to test different code paths on the same test system. These combine to make GLIntercept a very promising platform on which to build.

Three projects were found which support debugging of fragment shaders. The first, Shadesmith, uses a rewriting method as this paper's project does, but rewrites assembly language instead of GLSL[2]. This is possible because the language being debugged is assembly, so it is always available to the debugger. Shadesmith does require modifying the host program, but the change is minimal. Details on the rewriting algorithm are scarce, being given only in the SIGGRAPH presentation slides[1].

The second fragment debugger is described in [7] but is not publicly available. Its main focus is a query language to provide context for debugging information along the pipeline from geometry and textures to fragment output. To facilitate this, variable values inside the shader can be selected during runtime. However, it is not described how the values are made available, so all that's known is simply that it is possible. Cg was chosen for the project, but the paper states it should be applicable to other high level languages including GLSL.

Finally, Apple's OpenGL Shader Builder allows step-through debugging but not in its native program[9]. It sets up test scenes similar to the IDEs by NVidia and ATI. Its step-through debugger operates on one pixel at a time, and this pixel along with any other inputs to the shaders can be specified before or during an execution. Shaders can also be edited and recompiled during debugging. The latest version of the Shader Builder supports GLSL as well as ARB assembly programs. Apple does not publish its debugging methods, so it may be done in either hardware or software. In addition to the debugger, a texture browser is available.

III. Shader Debugging Methods

The two minimal capabilities required by a step-through debugger are:

1) To show variable states at a given line to the programmer
2) To move the debug line to the next line to be executed

There is an OpenGL extension proposed that would provide these capabilities called MESA_program_debug[13]. It works by allowing the debugger to register a callback function which is called after each line is executed. In the callback, other functions would return variable state. However, the extension only provides an interface, not an implementation. It states that it is unknown whether a hardware implementation is possible, and if not, it would have to use a full software implementation as Visual Studio does for HLSL. Mesa is the closest to providing this since it is a full software implementation of OpenGL's fixed pipeline. However, a software shader implementation does not exist for Mesa at the time of this writing and has been put on hold.

Traditional debuggers work with the CPU, compiler, and operating system to follow the CPU's actual state as it executes the program and report it to the user[10]. To do the same for shaders would require access to the compiled shaders so the assembly could be modified. However, GLSL is passed in source form to the drivers at runtime where it is compiled. The implementation of the compiler may change at any time. For example, if a driver chooses to compile to a standard assembly language, that may aid in writing a debugger. But since the driver may remove that step and compile directly to machine language, any intermediate steps should not be used in designing a debugger. Furthermore, intermediate steps and the final machine language are not made available in a standard way, for example in a temporary file or position in memory, so retreiving them for rewriting is not reliable.

Since low-level access is not available, the only remaining method is to modify the GLSL. This creates a large amount of overhead both in shader size and running time, but it is guaranteed to work on any compliant GLSL platform. Due to the large increase in shader size, this method is only feasible on modern cards where shader size limits are substantially higher.

IV. Shader Rewriting

Printf-style debugging is a manual debugging method currently used by shader programmers. It consists of setting the shader's output color, gl_FragColor, to the variable being watched. Care is then taken that the output color is not overwritten before the shader finishes running. This debugging method can be implemented automatically, but the output may not be as intuitive as that of a traditional program running on a CPU.

One of the advantages of this technique when applied to shader programming is that output from all of the pixels can be viewed at once. However, it must be kept in mind that the execution paths of the program for each pixel is not necessarily the same, since the inputs from the vertex shaders may be different for each pixel. For example, if an input from the vertex shader is different for two pixels, and that input causes an assignment to be skipped for one pixel but not the other, then it is unknown to the user which assignment resulted in the output.

In fact, the amount of thought necessary to interpret the results of printf-style debugging, combined with the common need to make small adjustments to the code so the output is more easily

interpreted, means it is not very useful to make this process automatic. The debugging scenario would be a programmer clicking on different variables in their program, viewing the output, and spending most of their time thinking about how the value was obtained and probably then rewriting the code by hand.

The solution is to find a way to do step-through debugging for the whole screen at once. If all of the shaders can be stepped forward one line with a single click, and the debug lines of each pixel visualized and explored, this would help debug all of the shader's execution paths at once.

There is traditionally only one debug line indicator since only one execution path is examined at a time. To see the paths of thousands or millions of pixels at once, multiple indicators with pixel counters indicating the number of pixels at that line can be used. Then each indicator could be selected and the corresponding pixels at that line shown on screen to find execution patterns. The problem is that it can't be shown how the execution got to that point. Therefore this technique would be used to find interesting pixel groups, and then single-pixel debugging could be applied by selecting the pixel via mouse-over on the 2D output.

The conclusion is therefore that automatic printf-style debugging is probably not helpful, full-screen step-through debugging may be helpful in finding interesting pixel groups, and single-pixel step-through debugging is where the execution path and complete understanding of why a variable's state is what it is can be found. The following rewriting techniques will explore how to implement the latter two methods.

Both methods require setting the output color to the watch variable's value. If the watch variable is a vec4 (4-component float type), the assignment is a simple copy. If it is a vector of a smaller dimension or a scalar, it can be expanded to fill the vector. Other data types such as matrices and textures require non-trivial visualization methods. Some of these methods are described in section VI.

To illustrate the rewriting methods, a sample shader is introduced in its original form:

```
void main()
{
    float x = 0.1;
    for (int i=0;i<3;i++)
    {
        x *= 2.0;
    }
    gl_FragColor = vec4(x,0.0,0.5,1.0);
}
```

The first technique examined to return a variable's value was to set the watch variable to the output color and then use a return statement to skip the rest of the shader. This was meant to ensure the output color would not be overwritten by the remaining shader code. This method works if the debug line is in the main function, but in other functions it only returns to the main function. Here

is what it looked like:

```
void main()
{
    float x = 0.1;
    for (int i=0;i<3;i++)
    {
        x *= 2.0; gl_FragColor = vec4(x,x,x,1.0); return;
    }
    gl_FragColor = vec4(x,0.0,0.5,1.0);
}
```

This chooses the correct variable because GLSL handles scoping issues and always chooses the correct one. However it only works in the main function; in other functions it would only return to the calling function, and gl_FragColor would be overwritten.

The next method that was considered was to use a temporary, global (per-fragment) value. This temporary would save the value and then be used to set the output color on the last line of main:

```
vec4 tempVariable;

void main()
{
    float x = 0.1;
    for (int i=0;i<3;i++)
    {
        x *= 2.0; tempVariable = vec4(x,x,x,1.0);
    }
    gl_FragColor = tempVariable;
}
```

The above technique assumes the variable being saved is a vec4. If it is a vec3, vec2, or a scalar, values can be repeated to make it fit in to a vec4. Variable type can be determined by looking up the variable in the compiler's symbol table. This may require re-running the parser to determine which variable's type to use if the same variable name is used multiple times in the program. At the debug line, the parser would be stopped and the symbol table should contain exactly one instance of that variable.

This second method fixes the return problem and will work for any debug line in the program. The problem now is that if the debug line is executed multiple times, the temporary value will be overwritten each time and only the value from the loop's final iteration will be saved to the output color.

To display a value at a specified execution count requires either step-through debugging or a breakpoint which can stop execution and then restart it until it is hit again. To implement breakpoints, a counter would keep track of how many times the debug line is executed and then an if statement would set the temporary output variable only if the counter didn't reach its limit. This way it gets

around needing to know what the next line is, which as will be described requires many changes to the shader code.

One way to find the next line to be executed is to dynamically interpret the code to find structures that can alter the flow of execution, such as loops, method calls and if statements. This requires writing an interpreter which could read GLSL at runtime and provide just enough functionality to know the execution order. This is a promising direction for future work.

Another way is to instrument each line of the source code and let the program output the line that was executed next. This adds a great deal of overhead but is very easy to implement. Depending on whether full-screen or single-pixel debugging is being used, the output will either be the next line for each pixel in the entire screen or only the next line for the pixel being debugged. The only difference is how the next-line information is stored, which is determined by inputs to the fragment shader in the case of full-screen debugging. This can be calculated once at the beginning of either debugging method and then the rest of the rewriting is identical, since both methods output to a texture to be read back to the CPU. For single-pixel debugging, the texture to be read back has dimensions 1x1.

In the following example, next-line instrumentation is added to the code which was rewritten using the temporary variable method. Since both the next line and fragment color need to be read back, Multiple Render Targets (MRTs) are used. Note that the second target must be unclamped to store line numbers higher than one. Limits due to inaccurate integer storage in a floating point texture should be considered, but should not be a problem since GPUs work with 32-bit floats.

```
int nextLine=-1;
bool save=false;
vec4 tempVariable;

void main()
{
  if (save) { nextLine=3; save=false; }   float x = 0.1;
  if (save) { nextLine=4; save=false; }   for (int i=0;i<3;i++)
  if (save) { nextLine=5; save=false; }   {
  if (save) { nextLine=6; save=false; }     x *= 2.0;
                                            tempVariable = vec4(x,x,x,1.0);
                                            save=true;
  if (save) { nextLine=7; save=false; }   }
  if (save) { nextLine=8; save=false; }   gl_FragData[0] = tempVariable;
                                          gl_FragData[1].x = float(nextLine);
}
```

Three instructions per line is the minimum for this type of instrumentation. The ”if” statement is necessary because each line must check a condition so it doesn't overwrite the saved information on later lines. An assignment is needed to set a flag once the next line is reached so, again, the variable isn't overwritten in later lines. And the third instruction, the assignment, is necessary on

each line because it isn't known at compile time which line will be executed next, so it must be encoded on each line.

This method assumes no expressions span lines, or the instrumentation will be added in the middle of the line and cause errors. Multi-line expressions can either be packed in to one line, or they can be detected and the added code can be omitted for the additional lines. Omitting added lines preserves the original code as closely as possible, and it is not significantly harder to implement, so it is the recommended method. Detecting multi-line expressions can be done by adding a field to the parser to indicate the beginning and ending line of each expression.

In addition to not adding code to expressions spanning lines, it must also not be added to lines outside of functions. These lines can also be found with the help of the parser.

Finally, to allow control over which iteration of the loop to get the value from, a counter is added. The debugger maintains an internal hit count for each line based on the return values from the next-line feedback. This example corresponds to the second iteration of the loop with the debug line still 6:

```
int counter=0;

// this is a sampler2D for full-screen step-through debugging
uniform int counterTarget;

int nextLine=-1;
bool save=false;
vec4 tempVariable;

void main()
{
  if (save) { nextLine=3; save=false; }   float x = 0.1;
  if (save) { nextLine=4; save=false; }   for (int i=0;i<3;i++)
  if (save) { nextLine=5; save=false; }   {
  if (save) { nextLine=6; save=false; }     x *= 2.0;
                                            counter++;
                                            if (counter==counterTarget) {
                                              tempVariable = vec4(x,x,x,1.0);
                                            }
                                            save=true;
  if (save) { nextLine=7; save=false; }   }
  if (save) { nextLine=8; save=false; }   gl_FragData[0] = tempVariable;
                                          gl_FragData[1].x = float(nextLine);
}
```

V. Implementation

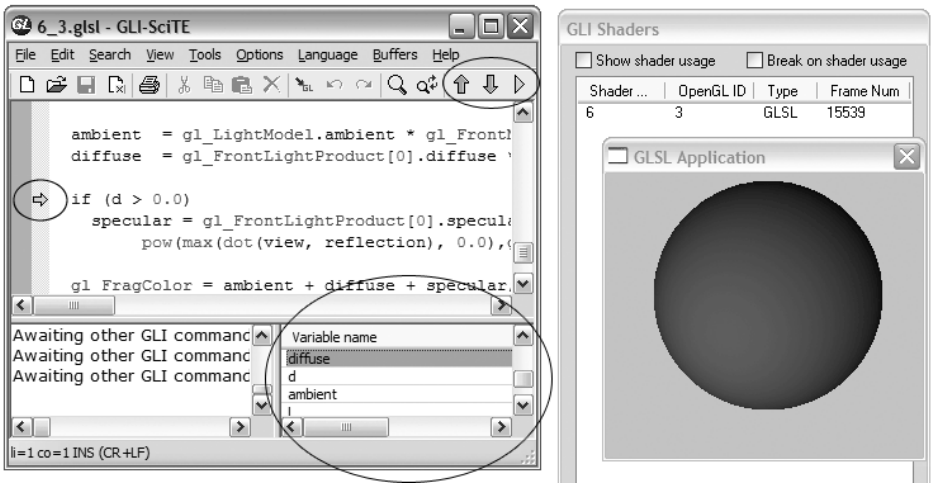

Figure 3. Screenshot of debugging modifications to GLIntercept.

GLIntercept was used as a base because it is open source and very modular, so adding the GUI elements and the debugging mode was relatively easy. It was the only open source project that had the ability to recompile shaders on a running program. Also, its additional debugging facilities mentioned in Section II complement shader debugging well.

GLIntercept uses SciTE, another open source project, for the shader editor. The SciTE module was modified to add a watch window and a step-through indicator and controls. The watch window is used to select a variable to monitor, and the only current controls are step-down and step-up. Step-into is the method described in Section IV, and step-over and step-out require more information from the program and therefore additional rewriting techniques. Figure 3 shows the step-down and step-up controls circled along with a button to start step-through debugging. Also circled are the watch window and debug line indicator.

The watch window uses a modified GLSL parser to retrieve a full list of variables. The original parser maintains a mutable symbol table which adds and removes variables as they pass in and out of scope. This was originally thought unusable since at the end of the program, all non-global variables are out of scope. The original plan called for a single list of all variables available in the program. To generate this, a second symbol table was added to the parser which was sent all of the add commands that the original table received, but the delete commands were not sent to this second table. This resulted in a symbol table containing all the variables in the program. This method resulted in multiple variables with the same name, so other ways of distinguishing between variables were explored.

To only list variables in scope, a separate parser pass is made each time the debug line changes. This makes the second table described above unnecessary. In each pass, the parser is stopped once

the debug line is reached, and the symbol table at that point is returned. It then contains only the symbols in scope at that time. The data types in the table can be used to write the conversion from vec3, vec2, float, and int variables to vec4.

The line indicator shows where the debug line is. The first version of the line indicator is simply a green marker that can be moved up and down by icons in the icon bar. It does not follow the execution of the shader and in fact can be set outside of any function. It can be used to select a line, see and select a variable in scope, and view the output. It can not be used to choose a particular execution inside a loop or recursive function.

With the debug line chosen and a selected variable verified to be in scope, the shader can be rewritten to output the selected variable to gl_FragColor using the first version of the temporary variable technique described in Section IV. Since the debug line at this point can not distinguish between multiple executions of the same line, which requires stepping through, no counter is used and the last value saved to the variable is what is seen in the output.

Once variable state could be viewed with the line selector which didn't obey program execution flow, more instrumentation was planned to be added to return the next executed line. All rewriting to this point was done with simple string replacement functions. The next step in the project would return the next lines in the execution flow in a texture.

VI. Future Work

Variable types other than scalars and vectors have non-trivial visualization methods that were not explored in this project. Matrices could have a color assigned to them to replace the fragment shader output, and their full value could be shown on mouse-over. Textures can be shown in a texture browser similar to what is available in Rendermonkey and FX Composer. More visualization options may be added, such as more useful color mappings to highlight zero vs. almost zero values.

Step-into debugging was the only type described and implemented. Step-over debugging requires moving to the next line if the line contains a function call, but otherwise the flow is the same. If the line is the last in a function, then the next line in execution will not be the next in source code, so function calls must be identified. This can be done with the GLSL parser. Implementing step-out debugging also requires additional information, notably where function bodies are located. The next line can then be set to the end of the current function where the step-into method can be used.

Determining the next line to be executed requires the most overhead in the rewriting method described. Implementing an interpreter which can compute the execution path without using the GPU would be a step towards a full software implementation, and significantly relieve the shader size constraint.

VII. Conclusion

The direct GLSL rewriting method described can be made relatively robust and should remain transparent to the user except for the performance and code size penalties. The overhead is great, but this solution can work immediately and provide a temporary solution until the MESA proposal is implemented. It also has the advantage of running on hardware and possibly providing a more accurate test platform in some cases.

References

[1] Purcell, Tim. “Fragment Program Debugging Tools. (SIGGRAPH slides on Shadesmith)” <www.gpgpu.org/s2004/slides/purcell.Debugging.ppt>

[2] Purcell, Tim. Shadesmith Fragment Program Debugger. <http://graphics.stanford.edu/projects/shadesmith/>

[3] Merry, Bruce. buGLe OpenGL Debugger. <http://bugle.sourceforge.net/>

[4] Gould, Dave. GLSurveyor. <http://www.glsurveyor.com/>

[5] Graphic Remedy. gDEBugger. <http://www.gremedy.com/>

[6] Hawk Software. GLTrace. <http://www.hawksoft.com/gltrace/>

[7] Duca et al. “A Relational Debugging Engine for the Graphics Pipeline.” *ACM Transactions on Graphics 24, 3* (Aug. 2005).

[8] Trebilco, Damian. GLIntercept. <http://glintercept.nutty.org/>

[9] Apple Computer, Inc. OpenGL Shader Builder. <http://developer.apple.com/graphicsimaging/opengl/shader_image.html>

[10] Rosenberg, Jonathan B. “How Debuggers Work.” Wiley Computer Publishing, 1996.

[11] Rost, Randi J. “OpenGL Shading Language.” Addison Wesley, 2004.

[12] Kessenich et al. “GLSL specification.” <http://oss.sgi.com/projects/ogl-sample/registry/ARB/GLSLangSpec.Full.1.10.59.pdf>

[13] Paul, Brian. “Mesa_Program_Debug specification”. <http://www.mesa3d.org/MESA_program_debug.spec>

[14] Microsoft. “Microsoft DirectX 9 and HLSL Press Release.” 22 Jan, 2003. <http://www.microsoft.com/presspass/press/2003/Jan03/01-22DirectXHLSLPR.mspx>

[15] Diamond Multimedia. “Diamond Monster 3D Press Release.” 31 Oct, 1996. <http://www.diamondmm.com/whats-new/press-releases/files/monsterishere.html>

[16] Perlin, Ken. “Making Noise. (GDC Presentation)” 9 Dec, 1999. <http://www.noisemachine.com/talk1/7.html>

[17] RapidMind Inc. Sh. <http://www.libsh.org/>

[18] Buck et al. BrookGPU. <http://graphics.stanford.edu/projects/brookgpu/>