Graphics Hardware (2007)
Timo Aila and Mark Segal (Editors)

A Hardware-Aware Debugger for the OpenGL Shading Language

Magnus Strengert, Thomas Klein, and Thomas Ertl

Institute for Visualization and Interactive Systems
Universität Stuttgart

Abstract

The enormous flexibility of the modern GPU rendering pipeline as well as the availability of high-level shader languages have led to an increased demand for sophisticated programming tools. As the application domain for GPU-based algorithms extends beyond traditional computer graphics, shader programs become more and more complex. The turn-around time for debugging, profiling, and optimizing GPU-based algorithms is now a critical factor in application development which is not addressed adequately by the tools available. In this paper we present a generic, minimal intrusive, and application-transparent solution for debugging OpenGL Shading Language programs, which for the first time fully supports GLSL 1.2 vertex and fragment shaders plus the recent geometry shader extension. By transparently instrumenting the shader program we retrieve information directly from the hardware pipeline and provide data for visual debugging and program analysis.

Categories and Subject Descriptors (according to ACM CCS): I.3.4 [Computer Graphics]: Graphics Utilities – Software Support I.3.8 [Computer Graphics]: Methodology and Techniques – Languages D.2.5 [Software]: Software Engineering – Testing and Debugging

1. Introduction

Graphics processing units have evolved into highly flexible, massively parallel computing architectures and have reached a level of programmability roughly resembling their CPU counterparts. Recent advances include full integer arithmetics, unlimited program instruction count, and genuine dynamic flow control. Simultaneously, this progress in hardware technology was followed by the introduction of suitable high level shading languages, i.e. HLSL, Cg, and GLSL, nearly as powerful as traditional CPU programming languages. In addition, they offer also advanced features, like vector and matrix data types, reflecting the underlying SIMD architecture of the GPU. The high computational power of modern graphics processors combined with the ease of these high level programming languages allow for more and more complex algorithms to be implemented on the GPU as well as enabling a widening range of applications to benefit from the use of graphics hardware as a numerical co-processing unit. Accordingly, shaders tend to become more and more complex.

However, the development environment, in particular debugging and code analysis tools, did not keep up with the rapid advances of the hardware and software interface. As a result, developers typically spend quite a large amount of time locating and debugging programming errors. Thus, the availability of effective debugging means is essential for fast and efficient shader code development.

In comparison to a traditional debugger for sequential CPU programs, the special characteristics of graphics hardware pose additional challenges for a shader debugger. Most important, there is no direct access to the hardware, i.e. there is no specific low-level debugging interface. Second, it has to deal with the intrinsic parallelism of the graphics hardware that requires to deal with thousands or even millions of threads running in parallel.

In general, current solutions for shader debugging can be divided into two basic approaches: software emulation and shader code instrumentation. The former uses a software implementation of the rendering pipeline in order to emulate the execution of shader programs according to the specification of the shading language. This allows for direct control of program execution and provides access to arbitrary data content. The main drawback of this approach is that debugging is not performed on the target hardware and results may therefore not correspond to actual hardware values. In contrast, shader code instrumentation will provide true hardware values, but access to the data is complicated. Since there is no direct access to the individual processing elements, the only possibility to get data back from the GPU is to read back the final result of a shader invocation, i.e. vertex attributes or pixel color. Accordingly, it is common practice in shader development to use printf-style visual debugging by manually rewriting the (fragment) shader to return the value of interest as its final output and to interpret the resulting images. However, in the context of multi-pass algorithms involving offscreen render targets or if debugging full floating-point precision data, the direct display of intermediate results is often not sufficient and a fair amount of code changes to the host program is required to permit readback of rendering results to main memory. This is even worse for vertex and geometry processing units, as they do not output directly displayable content and debugging those shaders may require additional changes in subsequent stages of the rendering pipeline. Furthermore, this manual Edit&Continue style of debugging is tedious and error prone.

An efficient shader debugging tool therefore must be able to automatically instrument the shader code and host program in an application-transparent manner. In particular, it has to work without explicit code changes, i.e. does not require modification and re-compilation of the host program.

In this paper we present a generic, minimal intrusive, and light-weight solution for debugging OpenGL Shading Language programs directly on the target hardware, that operates completely application-transparent. It supports the full GLSL 1.2 specification for vertex and fragment shaders including the recently introduced extension for geometry shaders [NVI06]. Shader debugging is performed on a per draw call level and allows singlestepping and the inspection of arbitrary variable content.

2. Related Work

Basic requirements for a versatile GPU debugger have been defined by Purcell [Pur05] and more recently by Owens et al. [OLG*07]. They conclude that a practical tool for GPU debugging should provide essentially the same basic features as a traditional debugger. These include variable watches, program break points, and singlestep execution of the GPU programs. Additional requirements arise from a graphics programming point of view. A GPU debugger should use the actual target hardware and not a software emulation, there should be little to no interference with the GPU state, and last it should maintain a certain degree of interactivity of the debugged host program in order to allow user interaction.

Currently available debuggers for graphics applications for the major graphics APIs, i.e. OpenGL and Direct3D, can be roughly divided into two distinct groups: graphics calls and state debuggers, and dedicated shader debuggers.

The first group includes Microsoft's Direct3D profiling and debugging tool PIX [Mic07] and a number of OpenGL state machine debuggers, namely spyGLass [Mag02], BuGLe [Mer04], GLIntercept [Tre04] and the commercial gDEBugger [Gra04]. While all of these tools provide the ability of API call tracing and logging as well as breakpointing (PIX and gDEBugger further allow to display various performance counters and other profiling information), only the PIX tool provides the possibility of shader debugging. However, for shader debugging PIX relies on the software emulation of the Direct3D reference rasterizer, i.e. no actual hardware values are debugged. In case of OpenGL some of the above mentioned solutions provide Edit&Continue shader editing, but currently no tool that we know of features a full-fledged shader debugger.

On the other hand there are solutions that are specifically geared towards shader debugging. The Apple OpenGL Shader Builder [App02] provides mechanisms to develop and debug ARB vertex and fragment shaders in a closed environment. The first tool to automate fragment shader debugging in the context of the target application was Shadesmith [PS03]. It introduced the so-called interactive deepening method, a technique for automatically generating a sequence of truncated debug programs for singlestep execution of ARB assembly fragment programs without flow control. Using Shadesmith requires slight source code modifications of the host program. Furthermore, its frame level debugging approach did not allow for debugging multi-pass algorithms.

Another approach that allows the analysis of the complete rendering pipeline including vertex and fragment programs was proposed by Duca et al. [DNB*05]. Their system is based on the Chromium [HHN*02] system to intercept and record the OpenGL command stream and stores data from the pipeline in a relational database. This allows subsequent queries for arbitrary elements of the captured OpenGL state using a specialized query language. Debugging high-level Cg shaders is facilitated by instrumenting shaders following an extented interactive deepening approach that allows for handling dynamic flow control. Up to now, the proposed system is the most complete solution for the analysis of OpenGL programs. The availability of the complete OpenGL stream offers ample opportunities for debugging and profiling OpenGL applications. Unfortunately, the systems was never made publicly available.

Due to the stream execution model of graphics hardware, debugging GPU programs is also closely related to debugging multi-threaded or distributed programs. There are a number of debugging tools available [The07, All07, Tot07] that extend the traditional sequential debugging paradigm to support the debugging of parallel tasks. While these solutions are well suited for debugging parallel programs in a typical development environment consisting of few dozens

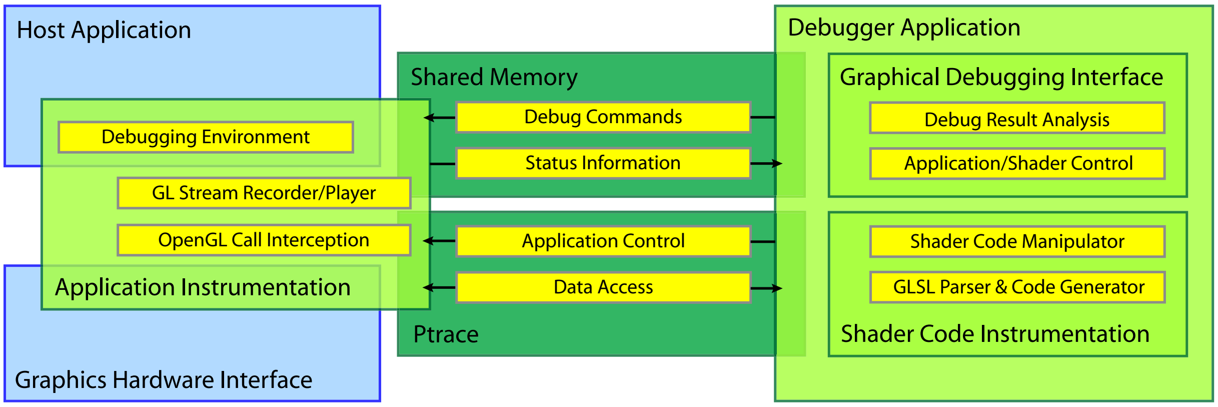


Figure 1: System architecture of our GLSL debugging environment.

of processors, their use is limited in the case of massively parallel execution with thousands of threads. Debugging a fragment shader program on a very small viewport already results in tens of thousands of independent threads. On the other hand, debugging a GPU program is much simpler with respect to the underlying execution model. Unlike a distributed parallel system, where inter-process communication and asynchronous events pose the major challenge for debugging, the independent execution model of the GPU makes things much easier.

3. System Overview

The proposed debugging architecture splits into two main components that communicate by means of exchanging essential debug information, e.g. debug commands, status information, and debug data.

The first componnets is a library running in the process space of the debugged host application, which enables us to instrument the host application for debugging GLSL shaders in arbitrary OpenGL programs without the need to recompile or even having the source code of the host program available. The concept is based on interactively intercepting all OpenGL calls evoked by the application during program execution with full access to all function parameters, thus enabling among other things the retrieval of shader source code. Depending on the current state of the debugger this allows for either running the host program unaltered with only little performance overhead or stopping and stepping through the execution of the target program on a per OpenGL function call level. The later is used to identify a single draw call as target operation for shader debugging. In addition, the instrumenting library establishes a debugging environment, i.e. float buffer objects, in the graphical context of the host application that permits not only the retrieval of the requested debug results but also assures that subsequent program execution is not affected by the debugging process.

The second component is the actual debugger application that can be further divided into two large modules. First, our GLSL shader code instrumentation performs automatic code manipulation. An OpenGL shading language parser is used to create an intermediate representation for a given shader that serves as basis for identifying the program execution order, variable scope determination, and the manipulation of shader source code in a syntactically and semantically correct manner. The compiler back-end is a GLSL code generator that reconstructs valid shader programs from the intermediate language. Second, a graphical debugging interface allows to control program execution of both the host application and the target shader, to select the draw call of interest from the OpenGL command stream, to specify debug requests for shader variable data, and to provide capable analysis and interaction methods for the generated debug results. Figure 1 gives a brief overview of the main components of our system and their interaction.

Based on these components the general control flow for debugging a GLSL shader program is as follows. For the draw call of interest the source code and execution environment of the currently bound GLSL shader is read back from the OpenGL state and passed to the shader code analysis module. We build an intermediate representation that serves as basis for scope determination, debug code insertion, and program control flow determination. Then, for each debug step in the shader program, an augmented debug shader is generated from the intermediate representation and inserted into the OpenGL state of the host application. Now the draw call is replayed and the debug result is read back and transfered to the debugger application.

4. Shader Code Analysis

In order to establish a basis for shader code instrumentation for a high-level shader language with dynamic flow control, it is necessary to fully analyze the syntactic structure of a shader program and to build an intermediate representation, i.e. a parse tree. This tree fully replicates the syntactic structure of the program string and provides the basis for automated code manipulation. In the context of Cg shaders this approach was sucessfully applied by Duca et al. [DNB*05] for debugging purposes. In our case we use a heavily updated and extended version of 3DLabs' OpenGL Shading Language Compiler Front-end [3Dl05].

The main focus of 3DLabs' original code is the rapid development of cross-platform compilers for low-level machine specific code generation. As the intended back-ends are highly vendor and hardware specific no actual implementation of a code generator is supplied. Applying this to shader debugging, the back-end is supposed to reconstruct valid GLSL shader code, which requires additional program information to be stored in the intermediate parse tree. Most importantly, this affects preprocessor directives, variable declarations as well as user defined `struct` data types. The former needs to be preserved since preprocessor statements expose direct compiler control that should affect not only the debug compilation process, but also succeeding compilation of the generated code. The later requires additional data to be stored, e.g. structure names, which is not necessary in the context of machine level code generation. To this end, the intermediate representation was extended to allow for syntactically valid and semantically equivalent reconstruction of a given shader in the compiler back-end.

In addition, the available system was updated to support current graphics hardware. Unfortunately, development for the original compiler front-end stopped with support for GLSL 1.10. Therefore, functionalities exposed by GLSL version 1.20, e.g. non-square matrices, handling of arrays as first class objects, etc., had to be integrated following the language specification. At last, we included the recent extensions for NVIDIA's G80 hardware, namely the `EXT_gpu_shader4` and `NV_geometry_shader4` specifications [NVI06]. To achieve compatibility for shader code that relies on vendor specific enhancements to GLSL, such as additional implicit type casts, we added support where changes were documented or perceived.

5. Shader Instrumentation

The requirements for shader code instrumentation in the context of automated debugging are threefold. First of all, manipulations to valid input code, i.e. shaders that comply with the GLSL specification and its extensions, must result in syntactically correct output, preserving the semantic structure of the input program in all parts that are not directly affected by the debugging process. In detail, required code additions must not induce any side effects to output registers, except for what is necessary to pass the target data values to the debugging environment. Particularly, with regard to fragment shaders, manipulation of the alpha and depth output are not permissible, as they influence the subsequent per-fragment tests. Furthermore, code manipulation should be minimal to assure a maximum degree of similarity to the input program, in particular with respect to hardware limitations such as nesting limits. Finally, the code instrumentation is required to allow debugging of any variable in scope at arbitrary code positions on an expression level.

In contrast to interactive deepening, we do not terminate the program reconstruction and execution directly after the debug target, but always restore the complete input shader. This ensures that subsequent calculations that may affect per element tests, e.g. depth or alpha test, are still performed by the instrumented code. In order to output debug values, a newly inserted varying is used in case of vertex and geometry shaders, while for fragment programs it is necessary to use at least one color channel of a bound render target. As GLSL features read-write access to output registers, early manipulation of an output register at the requested debug position may introduce side effects to the execution of succeeding parts of the code. Therefore, we define a global variable to buffer the debug result until it is safe to write its content to a result register, i.e. until the program control flow terminates. For all newly added variables we assure unique names to avoid scope collisions by appending random suffixes, if necessary.

The additional code introduced to assign a requested variable to the debug register is in most cases directly added in front of the target statement by using the sequence `(,)` operator. This method proved to be the most flexible and generic. As the sequence operator can be used in place for any single expression, its operation order from left to right is well-defined, and the return type and value are defined by the right-most operand, which will all become relevant later on. An example for basic shader code instrumentation, with automatically added code marked in boldface, is shown here:

```
float dbgResult;
void main() {
  dbgResult = gl_Color.x,
    gl_FragColor = gl_Color * 2.0;
  gl_FragDepth = gl_FragColor.x;
  gl_FragColor.x = dbgResult;
}
```

5.1. Conditionals

In contrast to traditional debuggers for serial programs, not all threads necessarily need to follow the same program path. For inhomogeneous cases the user has to decide which branch should be followed. To be able to base this decision on current variable content, debugging needs to be performed after the evaluation of the conditional test itself, given that side effects of the test possibly cause the target variable to change. On the one hand, adding the required debug expression after the conditional statement using the sequence operator would result in a wrong evaluation of the test and thus breaks the semantic equivalence with respect to the original program. This is due to the fact that the rightmost operand determines the result value, which would be the newly added expression. On the other hand, code insertion can not be moved inside the branch body for conditionals that do not affect all elements, as it is the case for conditionals without an `else` branch.

As the usual way of inserting the debug code cannot be applied in this special case, the proposed solution uses a temporary, locally defined variable to buffer the result of the conditional test. Again, by utilizing the characteristics of the sequence operator, the conditional test is evaluated before the shader instrumentation while the resulting value of the used sequence is assured to equal the original conditional statement. This is illustrated in the following code fragment.

```
bool dbgCond;
if (dbgCond = ((i++) < 10),
    dbgResult = float(i), dbgCond) {
  i = 20;
}
```

5.2. Loops

Since GLSL exposes no built-in loop counters, debugging statements at a specific user defined iteration inside loops, i.e. statements for which the corresponding call stack holds elements enclosed by a loop structure, iteration requires the definition of a debug loop counter per nesting level. Having these counters defined at global scope allows for debugging a specific iteration even if the target is not directly an element of the loop body anymore, as it occurs by stepping into user-defined functions.

There are two alternatives for adding loop-aware debug code. One is inserting an `if` block that restricts its body to be evaluated only if the loop counter matches the requested iteration. The disadvantages of this approach are twofold. The `if` statement cannot be used in place for arbitrary expressions and insertion of debug code would increase the nesting level and may exceed the hardware given limit. Instead we propose to use the logical-AND (`&&`) operator that evaluates the right-hand operand only if the left-hand operand evaluates to true. An additional `true` expression is appended to the debug assignment to assure scalar Boolean return type. An equivalent construct is possible using the selection (`?:`) operator. Both do not increase the nesting level and are very likely to map to conditional write masks on graphics hardware, as indicated by NVIDIA's cgc compiler.

```
int dbgIter0;
...
dbgIter0 = 0;
for (i = 10; i > 0; i--, dbgIter0++) {
  (dbgIter0 == 5 &&
  (dbgResult = float(i), true)),
  f += f;
}
```

5.3. Function Calls

To debug a called function only at a single distinctive invocation, the complete function is duplicated and renamed. We use this approach in favor of adding conditional code to the function body, as changes are thus limited to the debug call stack and evaluation of non-debugged calls remain completely unaffected by additional code.

In addition to the function call itself already its parameters can induce side effects, which requires to place code after execution of all parameters, but still before the actual function call. This problem is similar to debugging conditionals, as it is described above. The key element is to buffer the rightmost `in` parameter and insert code after its evaluation. This is sufficient, as `inout` and `out` parameters need to be l-values in GLSL and thus cannot feature side effects.

```
void F(inout int p1, int p3, out int p4);
...
int dbgParam;
F(i, (dbgParam = (k += j),
      dbgResult = k, dbgParam), k);
```

6. Host Application Instrumentation

As already mentioned, instrumenting the shader code is only part of the proposed solution. Since graphics applications often employ a number of different shader programs, it has to be possible for the user to select the one of interest. Furthermore, since the execution of a shader program is triggered solely by rendering geometry, a shader debugger must also provide a method for selecting the draw call of interest. This means that at least part of the functionality of conventional OpenGL state machine debuggers is required. Most importantly, a possibility for interactively stepping through OpenGL calls as they are invoked by the application.

Acting as an interface between the host and the debugged shader, the application instrumentation must provide the means for replacing the original shader program by an instrumented one, to execute it, and to read back the debug results, thereby ensuring that the host application is oblivious of these changes. In short, the general process of debugging a shader for a given draw call is as follows:

1) Setup a debug environment (Sec. 6.1)
2) Record the target draw call (Sec. 6.2)
3) For each shader debugging request
    Inject instrumented shader in OpenGL state
    Replay draw call and read back debug result
4) Restore the original OpenGL state
5) Replay draw call to ensure correct continuation

For instrumenting the host application we use a combination of OpenGL command stream interception [BHH00] and the native `ptrace` debugger interface [Ins90] available on Linux systems. However, our approach is not limited to the Linux operating system. Other Unix systems as well as the MS Windows API provide equivalent functionality for monitoring and controlling the execution of processes from within a debugger. Ptrace is also used for transferring data between the address spaces of the host application and the graphical debugging environment. Additional communication between the debugger and the host application is handled through a shared memory segment (see Figure 1).

OpenGL command stream interception is realized by a shared library that provides function hooks, i.e. function definitions with the same signature, for all possible calls to the OpenGL and GLX library. These wrapper functions are automatically generated from the interface declarations provided by the OpenGL and GLX C header files. Thus, the system trivially supports all vendor-specific extensions known at compile time. Mapping the debug library into the process space of the host application through the preloading mechanism provided by the dynamic linking facility of the operating system causes its exported symbols to take precedence over symbols of libraries occurring later in symbol lookup scope and therefore allows us to intercept all calls to the original OpenGL implementation. Again, a similar mechanism exists for MS Windows, named DLL hooking [HB99].

Basically, each wrapper function is responsible for the following three tasks. First, it must provide the debugger tool with information about the function that is called, i.e. the function name and its parameters. Second, it has to be possible to call the original function and, eventually, communicate the potential result of the call or an error that might have occurred to the debugger. And last, it has to provide the means of performing an arbitrary number of additional debug operations. This functionality is realized by suspending the normal execution of the debugged program immediately after entering a wrapper function and switching to a special debug command execution mode. First, the name of the called function as well as the addresses of its function parameters and their respective types are stored in the common shared memory segment. Then, a simple event loop is entered. The debugger can now access the information provided by the wrapper function and issue debug commands. Among others this includes functionality for recording the current call for later playback, replaying a previously recorded OpenGL stream, retrieving the currently active GLSL shader program, injecting a new debug shader, or reading back the results of a shader debug step. Most importantly, this also includes calling the original OpenGL function to guarantee proper operation of the host application or evoking the function call with modified parameter values in order to facilitate debugging.

Besides the described functionality for OpenGL call stepping, the preload library offers an additional immediate execution mode that allows running the program without actually interrupting program execution until a stop command from the debugger is received. This provides the possibility of user interaction with the traced program.

6.1. Debug environment

The setup of the debug environment depends on whether a fragment or a vertex or geometry shader is debugged. In case of vertex and geometry shaders vertex data, i.e. the values of varyings emitted by the respective GLSL shader, has to be captured. This is accomplished by using the recently introduced `NV_transform_feedback` [NVI06] extension that allows to capture vertex data prior to the clipping stage of the rendering pipeline and to store it into vertex buffer objects that can be subsequently mapped into main memory. In order to capture correct data from a vertex shader, a potentially active geometry shader has to be disabled. The current primitive mode for transform feedback is either given by the output primitive mode of the geometry shader being debugged or by the primitive mode specified for the draw call in question in case of a vertex shader.

Reading back results from fragment shaders is realized using the `EXT_framebuffer_object` extension which in contrast to transform feedback is available on all current graphics hardware. A single 32bit floating-point RGBA color attachment is used to capture debug values. Additionally a depth render buffer and if necessary a stencil buffer are attached to the frame buffer object. To assure correct results we have to take care of the per-fragment operations setup and to disable all parts of the imaging pipeline that may affect debug values. In order to correctly rebuild the target applications' behaviour, per-fragment tests should operate identically regardless whether they work on the original target or debug render buffer. For debugging purposes, however, direct control of these tests is desirable, e.g. disabling the depth/alpha test may be beneficial in certain cases. Therefore the user can choose whether to copy alpha, depth, and stencil from the currently bound framebuffer or to clear them to user defined values for each shader step. Furthermore, the possibility for enabling or disabling individual fragment tests and blending is provided.

All OpenGL state changes required to read back debug data are implemented in a way that is completely transparent for the host application, i.e. the OpenGL state when leaving the debug stage is the same as before entering it.

6.2. OpenGL Stream Recording and Playback

Debugging shader programs requires to repeatedly render the geometry that triggers their execution. Although it possible to record and replay an arbitrary stream of OpenGL commands [DNB*05], we restrict ourselves to track single draw calls. This means that either a single OpenGL function call, e.g. `glDrawArrays`, or an immediate mode stream of OpenGL commands delimited by `glBegin` and the corresponding `glEnd` call is recorded. Since an immediate mode stream may alter GL state, it is also necessary to save those parts of the state that may change inside such a block. Working on a per draw call level has several advantages. In comparison to relying on automatic or forced frame redraws, record and replay is much more fine-scaled and flexible, allowing for debugging animations and multi-pass algorithms.

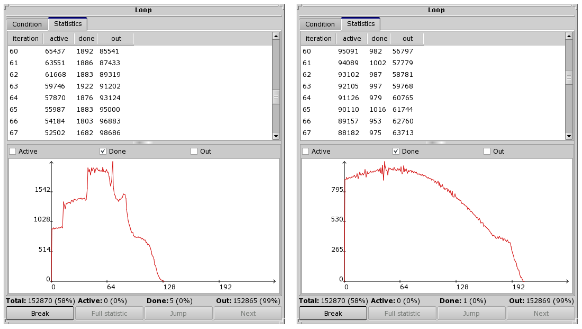
However, the draw call recording and playback method has its own pitfalls that have to be avoided. Rendering the same geometry multiple times invalidates OpenGL query objects and produces incorrect results if the application uses transform feedback mode, which results in incorrect behavior if program execution is continued after debugging a shader. However, these problems can be solved by taking special care of active query objects. In case of timer queries the solution is simple: We just ignore them, since the significance of timing results using an instrumented application program is questionable anyway. On the other hand, the result of occlusion or primitive queries might be crucial for the correct operation of an algorithm. The solution we propose is to keep track of active query objects. When entering the shader debug stage, active queries are terminated and their current values are saved. The queries are restarted using the same query object names when leaving the debug stage. Subsequent requests for `QUERY_RESULT` now must return the sum of saved and current query result. Dealing with active transform feedback mode is similar. Every time the host application calls `BeginTransformFeedbackNV`, it has to be checked whether a `TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN_NV` query is already active. If this is not the case, we start our own query. Thus, we can keep track of the number of primitives actually written to the buffer. Of course, queries started by the host have to be addressed as described before. An active transform feedback can now be terminated when entering the shader debug stage and it can be restarted using appropriate buffer offsets obtained from the primitive queries on exit.

7. Practical Considerations

Using the proposed system for debugging a shader program is similar to using a traditional source level debugger. However, there are some differences due to the special characteristics of graphics hardware.

7.1. Interactive Shader Debugging

From a users point of view the typical workflow of a debug session is split into two major tasks. First, by using a combination of interactively executing the target application and OpenGL call stepping, the draw call of interest for shader debugging is selected. For improved usability there is support for jumping to the next draw call, shader switch, or any user-specified OpenGL function as well as for optionally stopping program execution upon OpenGL errors. In addition to pure program flow inspection, it is possible to directly manipulate the OpenGL state machine by editing function call parameters, e.g. changing shader program uniforms. Next, the user selects a target shader and starts debugging by single stepping through the code. Due to the parallel nature of the



iteration	active	done	out
60	65437	1892	85541
61	63551	1886	87433
62	61668	1883	89319
63	59746	1922	91202
64	57870	1876	93124
65	55987	1883	95000
66	54184	1803	96883
67	52502	1682	98686

Total: 152870 (58%) **Active:** 0 (0%) **Done:** 5 (0%) **Out:** 152865 (99%)

iteration	active	done	out
60	95091	982	56797
61	94089	1002	57779
62	93102	987	58781
63	92105	997	59768
64	91126	979	60765
65	90110	1016	61744
66	89157	953	62760
67	88182	975	63713

Total: 152870 (58%) **Active:** 0 (0%) **Done:** 1 (0%) **Out:** 152869 (99%)

Figure 2: Using the loop analysis tool for comparing the efficiency of a volume rendering example with (left) and without (right) early ray termination. Note the higher number of fragments (y-axis) breaking out of the loop per iteration (x-axis) in the first case.

GPU, additional user interaction is required at program flow decisions, i.e. conditionals and loops. At each flow control instruction it is necessary to specify which execution path to follow. In case of conditionals whether to step into the true or the false branch or in case of loops if the loop should be left or another iteration should be performed. These decisions can be based on arbitrary debug data and are additionally supported by a per element evaluation and visual representation of the corresponding condition. At any time during the debug process the user can select variables from the current scope and add them to the list of active watch variables. The actual numerical values of watch variables can then be inspected on a per element level (vertex or fragment) or by using specialized visual debugging tools. For fragment data this currently includes an image viewer that maps floating-point fragment data to a color image. While this is the canonical solution for fragments, in case of vertex data such an inherent geometrical analogy does not necessarily exists. So far we offer two possibilities for inspection of vertex data: table views and scatter plots. Figure 3 and 4 on the color plate page show screenshots of typical debug sessions for fragment and geometry shaders.

7.2. Advanced Analysis

Combined instrumentation of the host application and the shader enables functionality that extends beyond traditional debugging. As an example, we implemented an in-depth loop profiling tool that summarizes statistical data for all iterations of a loop. This facilitates, for example, identifying the critical path of a shader. Figure 2 shows an analysis of the efficiency of early ray termination in a volume rendering application.

7.3. Limitations

Although the presented approach is very general, there are also some limitations. Most importantly, since shader instrumentation is done in a high-level language, it depends strongly on the correctness and reliability of the vendor-specific GLSL compiler and on the assumption that code manipulation, if performed in a semantically correct way, does not influence the program execution. In case of a driver or hardware bug, results are totally unpredictable, i.e. the bug may not manifest for the instrumented shader or, even worse, it will be triggered by the instrumentation in the first place. However, based on our practical experience current GLSL compilers seem to be quite mature.

Other limitations concern our solution for debugging vertex and geometry shaders. The necessary extensions are currently only supported on the NVIDIA G80 architecture. However, since Direct3D 10 requires similar functionality for streaming out vertex data, we expect all upcoming GPUs to support a comparable feature. Furthermore, it is not possible to debug vertex programs that operate on primitives generated by the OpenGL display lists mechanism. As there is no possibility of finding out what kind of primitives will be submitted by the execution of a display list. However, transform feedback mode requires specification of the output primitive type.

8. Conclusion

In this paper we presented a system for debugging GLSL programs that retrieves data directly from the hardware pipeline. Our solution allows to debug the complete programmable OpenGL shader pipeline, including vertex, geometry, and fragment shaders. Furthermore, it improves on the printf-style debugging approach still prevalent in today's shader development as it can not only be used for visual debugging but also for program analysis. The proposed system fits well into the shader development pipeline as it operates completely application-transparent and does not require any code changes or re-compilation of the host application. Our implementation of the system is available for download from the project web page.

Future work includes a MS Windows port of the application instrumentation sub-system, a concept of break points for the parallel execution model of shader programs, and more advanced program flow analysis, e.g. control flow of a single fragment/vertex, and statistics support, e.g. number texture lookups per sampler.

References

[3Dl05] 3DLABS CORPORATION: OpenGL Shading Language Compiler Front-end, 2005. 4

[All07] ALLINEA SOFTWARE: The Distributed Debugging Tool, 2007. 2

[App02] APPLE INC.: OpenGL Shader Builder, 2002. `http://developer.apple.com/`. 2

[BHH00] BUCK I., HUMPHREYS G., HANRAHAN P.: Tracking Graphics State for Networked Rendering. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware* (2000), pp. 87–95. 5

[DNB*05] DUCA N., NISKI K., BILODEAU J., BOLITHO M., CHEN Y., COHEN J.: A relational debugging engine for the graphics pipeline. *ACM Transactions on Graphics (Proc. of ACM SIGGRAPH '05) 24*, 3 (2005), 453–463. 2, 4, 6

[Gra04] GRAPHIC REMEDY: gDEBugger, 2004. `http://www.gremedy.com`. 2

[HB99] HUNT G., BRUBACHER D.: Detours: Binary interception of Win32 functions. *Proc. of the 3rd USENIX Windows NT Symposium* (1999), 135–143. 6

[HHN*02] HUMPHREYS G., HOUSTON M., NG R., FRANK R., AHERN S., KIRCHNER P. D., KLOSOWSKI J. T.: Chromium: a stream-processing framework for interactive rendering on clusters. *ACM Transactions on Graphics (Proc. of ACM SIGGRAPH '02) 21*, 3 (2002), 693–702. 2

[Ins90] INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.: IEEE Std 1003.1-1990, Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API), 1990. 5

[Mag02] MAGALLÓN M. E.: spyGLass, 2002. `http://spyglass.sourceforge.net`. 2

[Mer04] MERRY B.: BuGLe, 2004. `http://bugle.sf.net`. 2

[Mic07] MICROSOFT CORPORATION: PIX: The Direct3D profiling and debugging tool, DirectX 10 SDK, 2007. 2

[NVI06] NVIDIA CORPORATION: NVIDIA OpenGL Extension Specifications for the GeForce 8 Series Architecture (G8x), 2006. 2, 4, 6

[OLG*07] OWENS J. D., LUEBKE D., GOVINDARAJU N., HARRIS M., KRÜGER J., LEFOHN A. E., PURCELL T. J.: A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum 26*, 1 (2007), 80–113. 2

[PS03] PURCELL T. J., SEN P.: Shadesmith, 2003. `http://graphics.stanford.edu/projects/shadesmith/`. 2

[Pur05] PURCELL T.: GPGPU: general-purpose computation on graphics hardware: debugging tools. *ACM SIGGRAPH 2005 Course Notes* (2005). 2

[The07] THE PORTLAND GROUP, INC.: PGDBG Graphical Cluster Debugger, 2007. 2

[Tot07] TOTALVIEW TECHNOLOGIES, LLC.: The TotalView Debugger, 2007. 2

[Tre04] TREBILCO D.: GLIntercept, 2004. `http://glintercept.nutty.org`. 2