Graphics Hardware (2008)
David Luebke and John D. Owens (Editors)

Total Recall: A Debugging Framework for GPUs

Ahmad Sharif and Hsien-Hsin S. Lee

School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332

Abstract

GPUs have transformed from simple fixed-function processors to powerful, programmable stream processors and are continuing to evolve. Programming these massively parallel GPUs, however, is very different from programming a sequential CPU. Lack of native support for debugging coupled with the parallelism in the GPU makes program development for the GPU a non-trivial task. As GPU programs grow in complexity because of scaling in maximum allowed program size and increased demand in terms of realism, debugging GPU code is becoming a major timesink for content developers. In addition to more complex shaders, applications are using multi-pass effects in order to create more convincing reality. In this paper, we present a debugging framework that can be employed to debug complex code running on the GPU in an efficient manner. By observing the API calls of the application that are made to the 3D runtime, the framework can keep track of the program's state in memory. Upon the programmer's request, it is able to capture and deterministically replay the stream of instructions that caused the final write to a pixel of interest. This execution stream includes writes to intermediate render targets and spans across shader boundaries. The stream of instructions can then be replayed on the CPU via emulation and the programmer can debug the straight-line code with ease. We also present a hardware-friendly scheme that can be used to accelerate the debugging process for long-chain multi-pass effects.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: GPUs, Debugging

1. Introduction

In less than two decades, graphics hardware has evolved from 2D accelerators, fixed-function 3D accelerators, to today's massively parallel programmable GPUs. Current GPUs on the market have nearly billion transistor budgets and are used for general purpose computation in addition to rendering 3D graphics. As GPUs are becoming more and more programmable, their complexity and cost to program them in terms of time and effort are also increasing. This heavy cost to program GPUs is one of the biggest man-hour sink associated with creating shaders as well as porting sequential CPU code to the GPUs. The cost of programming GPUs can be mainly attributed to the lack of native debugging support and the parallel nature of the primitive processing on the GPUs.

To tackle the challenge of debugging code on the GPU, several techniques have been proposed. These techniques can be divided into two broad categories: (a) instrumenting shader code followed by outputting an intermediate value to the bound render target for the programmer to inspect, or (b) emulating all primitives sent to the GPU on the CPU. Iterative deepening is an example of the first technique. Only part of the shader is executed on the GPU and the intermediate values are dumped to the render target. These intermediate values are then displayed as colors on the screen, or read back by the CPU and printed for the programmer to view. However, current manifestations of GPU debuggers using iterative deepening cannot debug across render target switches. The latter mentioned technique, emulation of the GPU execution on the CPU, does vertex, geometry and pixel shading of the entire scene on the CPU. By emulating all the work on the CPU, the programmer can easily put in breakpoints and execute the shader step-by-step. However, emulation can be slow, especially when the number of primitives or the resolution is large, or the shader is complex.

This paper introduces *Total Recall*, a debugging framework that content-publishers can use to rapidly identify bugs in the code running on the GPU. Our debugging framework allows programmers to insert conditional breakpoints inside shaders to enable error checking during development. When a breakpoint is hit, a complete execution history of the pixel that hit the breakpoint can be obtained from the debugger runtime for emulation on the CPU. The programmer can then execute the relevant code step-by-step and can observe all intermediate results. Note that this technique does not suffer the performance loss of full emulation, as only the programmer-specified primitives are selected for emulation on the CPU. By letting the CPU handle the complex task of step-by-step execution with watches on variables, we can simplify and speed up the debugging process.

Our main contributions are two-fold:

1. We present a debugging framework that allows the capture and deterministic replay of the entire execution stream of a pixel of interest. Note that this stream may contain writes to dynamic textures that were the sources of the pixel of interest. This execution stream, once obtained, can be replayed step-by-step on the CPU, as desired by the programmer via emulation.
2. By tracking finer buffer dependencies, as explained in Section 4, we can accelerate the debugging process for multi-pass shaders. Hardware modification is required for this acceleration technique.

The rest of this paper is organized as follows. First, our motivation for our technique is presented in Section 2. The next section, Section 3, describes the GPU programming model and describes the challenges associated with debugging GPU code. Section 4 then explains our proposed technique and delves into its implementation. Section 5 describes our experimental framework for implementing our debugger framework for Direct3D 9. Finally Section 8 concludes.

2. Motivation

Most programmers are used to debugging sequential code using breakpoints, watchpoints, or printf statements. Breakpoints and watchpoints are the primary mechanism employed by debuggers such as gdb, windbg, etc., to pause or intercept the program state so the programmer can inspect variables at certain interesting events. Printf statements, a less elegant approach, are used to dump out intermediate values of program variables for the programmer to figure out how the program computed its final output. Unfortunately, neither breakpoints nor printf statements are natively available on GPUs. Emulation on the CPU can offer both of these, but emulated code runs at a much slower speed than native execution on the GPU. Another issue with debugging GPU code is that GPUs are inherently parallel machines and human programmers are not used to thinking in parallel.

For debugging code that is not executing correctly, programmers are usually interested in watching or dumping intermediate values of a certain computation which they suspect is not working as expected. For graphics applications,

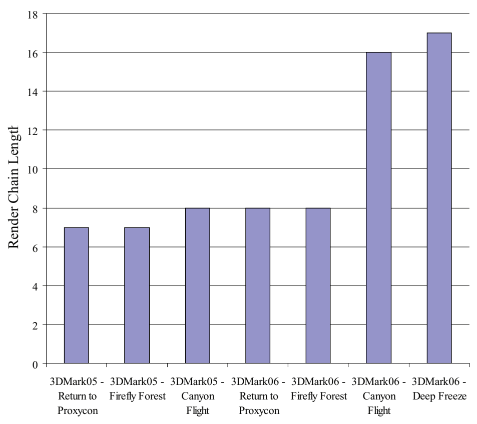


Figure 1: *Comparison of different benchmarks' maximum render chain length. This corresponds to the maximum number of dynamic textures data has to pass through before it shows up as a pixel on the final screen.*

this can be done on the GPU by compiling and running a shader whose final output is an intermediate value of the programmer's interest. Various programs, e.g., GLSLDevil [SKE07] and Shadesmith [PS03] for OpenGL, were introduced to intercept, modify and recompile shaders automatically.

However, current GPU debuggers are unable to debug the entire history of a pixel that went through several passes including render target switches. The programmer might be able to go through one shader step-by-step, but no framework exists, to the best of our knowledge, that allows the programmer to step through the entire execution stream that led to the final write of the pixel of interest.

A piece of data can pass through an entire *render chain* before appearing as a pixel on the final render target. Each shader transforms the input texture and geometry data into render target pixel values. Figure 1 shows the render chain length for several 3D benchmarks. As can be seen, a pixel value on the final screen can come from a chain of up to 17 textures in length for a benchmark (data passed through a chain of 17 dynamic textures before it appeared on the screen). Our debugger framework allows the programmer to step through all 17 shaders in sequence which makes programming multi-pass shaders much more intuitive and easier to debug.

3. GPU Programming Model and Architecture

To the graphics programmer, the GPU exposes itself as a z-buffer based primitive processing engine. Primitives like triangles, lines, etc., are processed by the GPU and end up showing on the render target as pixel values. Modern graphics runtimes like Direct3D and OpenGL provide the ability to perform three programmable operations on the primitives in the form of vertex, geometry and pixel shaders. Vertex shaders handle transformation and lighting of input vertices. Geometry shaders provide a way to amplify or destroy geometry in a programmable fashion while pixel shaders compute the final color of the pixels of the current render target.

The GPU does not define any ordering in which primitives will pass through this pipeline. To ensure correct results, the programmer enables z-culling with which geometry hidden behind other geometry will not be displayed on the render target. Because of this ordering constraint relaxation, the GPU can be architected to achieve a large amount of parallelism for graphics workloads: primitives can be processed in parallel with respect to each other. However, this parallelism comes at a cost: since most programmers are used to writing straight-line sequential CPU code, development and debugging on the GPU becomes a non-trivial task to perform.

4. Total Recall Approach

Step-by-step shader execution can be done by either: (a) stepping through all threads one instruction at a time, as iterative deepening performs, or (b) selecting one thread and stepping through it. Although stepping through all the threads has its value, debugging a single thread at a time offers several advantages over debugging all pixels at once:

1. Programmers, especially those having background in sequential CPU programming, are better at debugging sequential code than parallel code. Thus it would be easier for a programmer to follow just one execution stream rather than many, especially if different threads follow different paths (thread divergence).
2. Unlike multiple threads, a single thread will only have one set of variables that it works on. For a programmer with a background in CPU debugging, a single set of variables is easier to keep track of than multiple sets.
3. It might be the case that the programmer is interested in only one or a few bad pixels, in which case this debugging model fits perfectly in this situation.

Current methods like Microsoft's PIX [Wal07] and GLSLDevil [SKE07] allow the programmer to view all writes to the pixel of interest. The programmer can step then through the shader that caused each write. Applications, however, can use multiple passes or render-to-texture in which a write is made to an off-screen buffer, which is then read back in a subsequent draw call to write to the final render target. Popular techniques like Shadow Mapping [Wil78] and deferred shading [DWS*88] use multiple pass rendering to create convincing graphics effects realtime. A complete history of the pixel should incorporate all writes that were made to off-screen buffers that led to the final write of the current pixel. To the best of our knowledge, no current GPU debugger offers this. We propose a debugging framework that allows the programmer to recreate the entire execution history of the pixel that they might be interested in. This execution history can then be replayed on the CPU and the programmer can view all intermediate values via emulation. Notice that this is different from emulating the shader invocations on all the pixels on the CPU like Visual Studio .NET does for Direct3D shaders. Only the shader invocations that the programmer is interested in are captured and emulated on the CPU. This is also different from instrumenting the shader and running it on the GPU using iterative deepening as we selectively emulate code entirely on the CPU after obtaining the input values of the relevant shaders.

Our proposed debugging framework allows the following:

1. Conditional breakpoint statements inside shaders with break in program execution when the condition is satisfied.
2. Capture of the entire execution history of a particular pixel of interest. This history encompasses writes done to intermediate textures that were subsequently read by the shader that did the write to the final render target.

While this can be done entirely in software, as a library above the 3D runtime, we propose hardware support to do this more efficiently and quickly.

Some of the elements of our debugging framework are provided in the sub-sections below. These sub-sections will identify the "book-keeping" and data capture that is required by the debugging environment to debug shaders using our proposed technique.

4.1. Conditional Breakpoints

Conditional breakpoints are breakpoints that get hit upon the true evaluation of a certain condition. Once the breakpoint is hit, the state of the program needs to be presented to the programmer. Current GPU hardware does not support breakpoints natively. In order to capture the render state at the breakpoint, the debugger runtime needs to first identify the exact draw call which was made when the breakpoint was hit.

A simple way of doing the aforementioned is to bind an additional render target to the GPU state and write to it upon meeting a certain condition specified by the programmer as shown in Figure 2. This can be done fairly easily because conditional writes are supported by the modern GPU hardware. GPUs also support hardware counters to count the number of pixels drawn in a set of draw calls. After every draw call, the debugger runtime can read back this counter via a query to see whether this counter is non-zero. For example, in the Direct3D runtime, an object of the OcclusionQuery class can be used to perform the query on the hardware counter [Mic]. If the hardware counter is non-zero, the debug render target can be read back and searched to see which pixels hit the breakpoint. If multiple pixels are found, they can be presented to the user to select one to view its

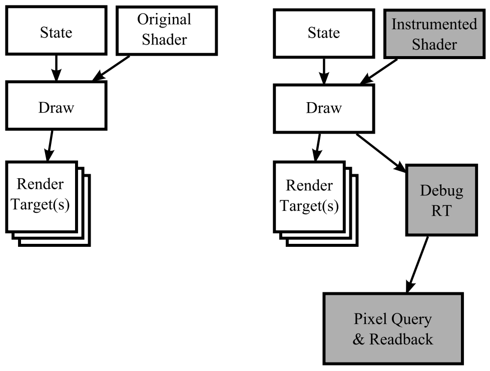


Figure 2: *A simple way of finding whether a breakpoint condition was met or not. A debug render target is bound to the render state and the shader instrumented with a conditional write to this render target. After the draw call, this debug render target is queried for number of pixels drawn. If the value is non-zero, the breakpoint condition was met in the draw call.*

execution stream. Presently, occlusion queries cannot track pixels written to individual render targets separately. Therefore, the shader has to be modified to remove the writes to the regular render targets when the debug render target is bound. Once it is known that the breakpoint was not hit, the original shader can be restored and the execution can continue like normal.

If the condition intended by the programmer is to break on a certain pixel (by selecting, or otherwise specifying the coordinates of it), the debugger can isolate the draw call that writes to that pixel by doing the following. The debugger runtime creates a depth buffer (of the same resolution as the render target) that has all pixel values set to the lowest depth except for the pixel of interest. This depth buffer can then be bound to the render state and the draw call made. If the draw call wrote a value to the pixel of interest, a query will return non-zero number of pixels drawn. In this way, the precise draw call that modifies the pixel of interest can be obtained.

Once we know the draw call that resulted in the breakpoint being hit, we need to extract the render state and all shader inputs to deterministically replay the shader on the CPU, where it can be executed step-by-step for analysis. The render state can be obtained by simply querying the 3D runtime on top of which the debugger runtime is running. For example, the Direct3D 9 device can be queried for the current bound render targets, textures, etc., by using function calls like ID3DDevice9::GetRenderTarget() in the ID3DDevice9 interface.

4.2. Obtaining the Shader Inputs

Shaders on the GPU have input values coming from the previous stages of the pipeline that are used to look up textures or perform other computations. These live-in values are needed to faithfully and deterministically replay the GPU execution stream that led to the breakpoint. Current GPU hardware supports gather, but no scatter. This means that for obtaining all input variables for a shader, we can use the following approach: create a render target whose bytes per pixel is equal to or bigger than the size of the live-in values of the shader times the number of pixels. The debugger runtime can then bind this render target to the render state and create a shader that packs all shader input values and outputs them to this render target (pass-through shader). The storage requirement of this debug render target can be reduced by a multi-pass approach: at every pass, a certain number of inputs of the shader are dumped to a smaller debug render target and read back.

4.3. Buffer Dependencies

As mentioned earlier, the goal of this debugging methodology is to provide the programmer with the entire execution history of a particular pixel. This consists of a stream of dynamic instructions that were executed on the GPU that finally led to the conditional breakpoint. The debugger runtime therefore has to keep track of all bound input textures to see whether they were a render target sometime before. A *buffer dependency graph* can be built by the debugger runtime by intercepting all requests to set the render target and checking whether the render target is a texture level or not. Once the render target is detected as a texture, the current shaders can be parsed and inspected to see if they are doing any look-ups from the currently bound textures. If so, those textures are added as parents to this render target. Figure 3 shows a buffer dependency graph for an application. In total 5 draw calls were made, each to a separate render target. The value of a pixel in the final render target depends on the entire chain of rendered textures as well as the input textures (and the geometry/state at each draw call).

Once a breakpoint is hit, the debugger runtime can walk through the buffer dependency graph and figure out exactly how to construct the execution stream that led to the breakpoint being hit.

4.4. Full Execution History

Execution history can be obtained by the following iterative scheme once the breakpoint has been detected as hit:

1. First, obtain the live-in values of the pixel that hit the breakpoint. This can be done by using the methods described in the sub-sections above. These live-in values of the pixel contain the texture coordinates of all its parents in the buffer dependency graph. If the parent textures are all static, meaning they have no parents in the buffer dependency graph, then can go to step 4.
   If the input bound textures are dynamic, go through each of the dynamic textures and program a breakpoint at the

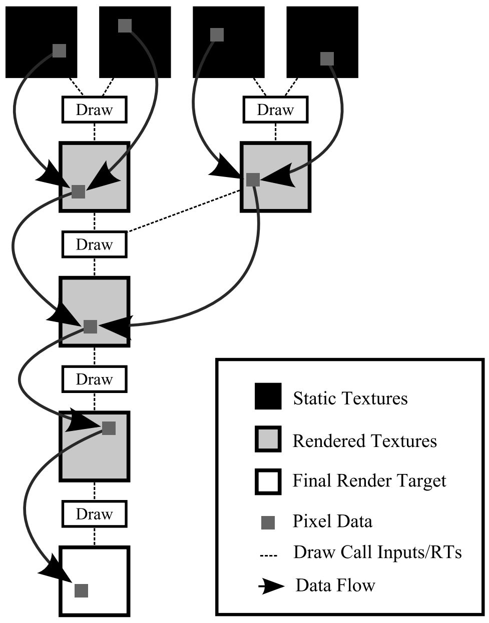
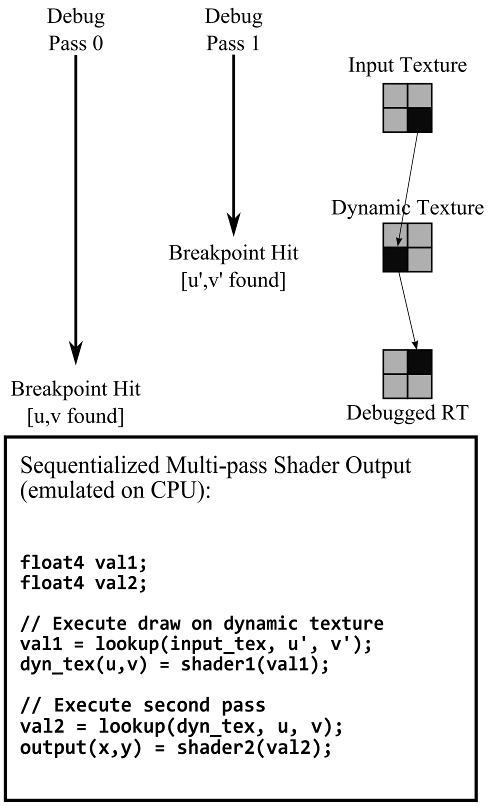


Figure 3: *Example buffer dependency graph. At each draw call the currently bound shader takes input from the input textures as well as the geometry and produces an output render target. Note that any pixel value in the final render target is dependent on pixel values of the previous passes' render targets.*



Sequentialized Multi-pass Shader Output
(emulated on CPU):

```
float4 val1;
float4 val2;

// Execute draw on dynamic texture
val1 = lookup(input_tex, u', v');
dyn_tex(u,v) = shader1(val1);

// Execute second pass
val2 = lookup(dyn_tex, u, v);
output(x,y) = shader2(val2);
```

Figure 4: *In debug pass 0, the render target under debug hits a breakpoint and its input values (u, v) are found. Next, a breakpoint is inserted at those coordinates in the dynamic texture. The same draw calls are made and we obtain the input of the dynamic texture (u', v') in the second pass. Finally the full execution stream is reconstructed and emulated for debug.*

coordinates obtained from the live-in values. If the dynamic texture is looked-up at various coordinates in the pixel shader, breakpoints will have to be set at all these locations.
If the input dynamic texture supports texture filtering upon texture access, breakpoints will have to be set at multiple locations per set of texture input coordinates. This is done so that the emulation environment can do texture filtering to produce the same result as the GPU hardware.

2. Start from the beginning of the frame and keep making draw calls until we hit the breakpoint that was programmed in step 1. The debugger runtime has to store the initial state and all state and draw call information (as well as vertex/index buffer writes) in the current frame to replay these draw calls. This information can be stored in memory and typically does not take more than 100 MB per frame (for the 3DMark06 game tests that we tested).
3. The breakpoint was hit. Go to step 1 to read the shader inputs.
4. We now have all the relevant execution history that can be passed on to the emulator module. This history includes all writes to dynamic textures that led to the breakpoint that was programmed in by the developer.

An example debug session is shown in Figure 4. In this case, 2 passes are required in order to obtain the shader inputs that can be used for deterministic emulation on the CPU.

Once this execution history is obtained, we can emulate the entire stream on the CPU and allow the programmer to go forward and backward in time. Through emulation, the programmer should be able to view all intermediate results, set watchpoints, etc.. The emulator can compute the final expected output of the shader and match it against the pixel values obtained from the series of draw calls in the replay buffer for sanity check.

4.5. Challenges and Limitations

The emulated output could be different from the GPU output because of floating point format and multi-sampling issues. Typical GPUs implement a non-standard floating point format for speed and efficiency reasons. In order to emulate the GPU execution more faithfully, a hardware-vendor supplied functional emulator can be used to emulate the non-standard floating point format implemented in hardware. This can enable the emulation result to be accurately paired with the real result from execution on the GPU.

Hardware support for multi-sampling/super-sampling can also cause differences between the emulation and actual GPU execution. Since the atom of pixel execution exposed to software is a single pixel, our framework cannot insert "breakpoints" at sub-pixel locations in the dynamic textures. Thus our framework is currently limited to accurately debugging render targets that do not have hardware support for super-sampling.

Alpha blending can present another challenge to isolating the shader inputs that will be used for deterministic emulation. When alpha-blending is enabled and a draw call is made, a binary search is done to isolate primitives that cause a write to the pixel location of interest. The debugger runtime has to intercept the draw call and make modified draw calls to figure out which primitives cause a write to the pixel location. The debugger runtime can use the depth-buffer clear technique mentioned before to see if half the primitives of the original draw call end up writing to the pixel of interest. Both halves have to be checked recursively to obtain all primitives that write to the pixel of interest. In the worst case, all primitives of the draw call write to the same pixel location with transparency enabled. In this case, these primitives will have to be drawn one-by-one, and all shader inputs obtained this way.

4.6. Accelerating multi-pass debugging

The main loop explained in Section 4.4 can take some time to execute, especially if there is a long chain of buffer dependencies. In this case, hardware can be employed to accelerate this process. Each source and destination buffer can be divided into a uniform grid of blocks. The hardware can maintain a bitvector for each render target block that maintains information about which source blocks were used in writing to this destination block. From reading this bitvector, one can determine the source buffer blocks that contributed to the write of a render target block. If there is a lot of spatial coherence in the pixel shader, this bitvector will be sparse. In this case, in step 2) from the previous sub-section, we can make a less expensive draw call by only drawing a small portion of the dynamic buffer. This can be done by clearing only a few blocks of the z-buffer, so that all other blocks will not be shaded because of early-z reject. This acceleration technique requires hardware modifications.

PIX and others already have functionality to determine the primitive id from the pixel value. If the exact primitive covering the pixel of interest is known, the debugging environment needs to only process that primitive in order to perform deterministic replay. This can be done in a few ways:

1. Assigning each primitive an id value and tagging a separate buffer with the primitive id of the primitive. In this way, by looking up the value of the primitive id in the corresponding pixel of the primitive-buffer, the exact primitive that covers that pixel can be determined.
2. By only clearing the z-buffer of the pixel of interest and reading back the number of pixels rendered after every draw call, the exact draw call that writes to the pixel can be determined. Then, a binary search can be performed on the primitives inside the draw call using the same mechanism to figure out exactly which primitive caused a write to the pixel. A subset of the primitives in the draw call are sent to the GPU and the hardware counter is queried to see whether they got drawn to the pixel of interest.

5. Experimental Results

In order to provide a proof-of-concept debugger, we implemented a debugger runtime library for Direct3D 9. Direct3D 9 is a 3D runtime that exposes a common interface to applications and games using COM objects that want to utilize the graphics hardware. On the back-end, the runtime interacts with the driver code provided by hardware manufacturers. The Direct3D runtime handles the creation, binding and destruction of various resources like vertex and index buffers, shaders, devices, etc. The application interacts with this runtime via different interfaces like IDirect3DDevice9, etc.

For our debugger, we encapsulated the Direct3D 9 interfaces in our own spy dll to capture the behavior of the application to debug. All the application needs to do is to put our spy dll inside the working directory and it will get loaded when the Windows loader inspects the import table of the application. The architecture of our debugger is shown in Figure 5.

Our debugging library saves relevant per-frame information like writes to vertex buffers, creation and setting of pixel and vertex shaders, etc., to segments of memory called replay buffers. It does that so the programmer can revert back to a previous frame by the press of a button. Once the programmer is in the desired frame, a pixel to debug can be selected by clicking on it.

Once we receive input from the programmer, we replay the current frame (by playing commands from the replay buffer), but instead of running the pixel shader of the application, we modify the shader on-the-fly to dump the input variables of the shader to render targets as described before. Additional render targets are created and bound if the inputs to the shader are more than the current render targets allowed. If multi-pass shading was done in the current frame, our debugger runtime detects that and iteratively executes the captured draw calls to obtain the input variables of the very first shader. Once this "execution map" is created, it is dumped to a file for the programmer to view.

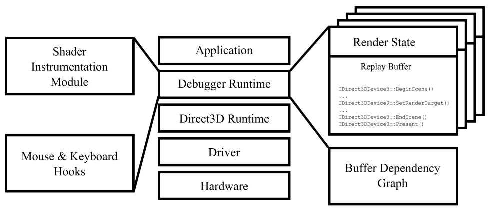

Shader Instrumentation Module
Mouse & Keyboard Hooks
Application
Debugger Runtime
Direct3D Runtime
Driver
Hardware
Render State
Replay Buffer
IDirect3DDevice9::BeginScene()
...
IDirect3DDevice9::SetRenderTarget()
...
IDirect3DDevice9::EndScene()
IDirect3DDevice9::Present()
Buffer Dependency Graph

Figure 5: *Architecture of our debugger framework. The debugger exports an interface identical to the Direct3D runtime and intercepts all Direct3D function calls made by the application. Replay buffers store per-frame information.*

6. Related Work

Single-threaded CPU debugging is a long-studied problem and effective solutions exist. Since the programmer is mostly interested in breaking on a special condition and inspecting intermediate values in registers and memory, a special breakpoint instruction (int 3 on x86 CPUs) is placed at the point of interest. When this breakpoint instruction is executed, execution jumps to an entry point in the debugger application and it displays the intermediate values in registers and memory. From here on, the programmer can execute instructions step-by-step to gain insight into the execution of the program. Conditional breakpoints are achieved by simply evaluating the desired condition and breaking execution if it is met.

A number of techniques have been proposed and implemented for GPU shader debugging. These techniques fall into two broad categories: the debugging environment intercepting shaders and modifying them on-the-fly, or emulating all shader invocations on the CPU. Shadesmith [PS03], Imdebug, Relational Debugging Engine [DNB*05] and GLSLDevil [SKE07] all use shader code instrumentation and readback after the draw call in order to debug shaders. .NET Shader Debugger uses the REF_RAST device and emulates shader code on the CPU. Starting from the June 2006 DirectX SDK version, PIX has a Pixel History feature which can be used to debug a single pixel by stepping through the vertex and pixel shaders that led to the write of that pixel.

Shadesmith [PS03] enables the user to step through shader code forward and backward. It does so by wrapping the actual OpenGL function calls in its own library and intercepting those function calls. When a draw call is made using some shader and the user wants to debug it, Shadesmith creates a temporary shader and binds that to the pipeline and makes the same draw call. This temporary shader executes a part of the original shader and outputs the intermediate result on to the render target. After the draw call, Shadesmith reads back the pixel values which contain the intermediate results. The user can step forwards and backwards and Shadesmith handles the creation and binding of the temporary shaders.

Microsoft's Visual Studio .NET can have conditional breakpoints inside shaders. It uses CPU emulation on a REF_RAST device to handle breakpoints. Every vertex and pixel is shaded on the CPU without discrimination. This emulation is slow, and becomes time-consuming especially at higher screen resolutions, high primitive count or longer shaders. Our technique does selective emulation of pixels that contribute to the pixel under debug.

gDebugger [TS05] is an OpenGL debugger that allows the programmer to visualize the OpenGL state before the draw call as well as after it. It does not allow stepping through shader code, but does offer a wide variety of performance visualizations.

The Relational Debugging engine [DNB*05], proposed by Duca et al, is implemented as a library for OpenGL which intercepts OpenGL function calls and stores the graphics state as a set of virtual tables. These virtual tables can then be accessed by an SQL-like query based language interface which is used to debug the application of interest. Like other debugging tools, they implement their debugger as a library over OpenGL's runtime, intercepting OpenGL state and shaders. The captured shaders are then instrumented and bound to the pipeline, yielding the desired results.

GLSLDevil [SKE07] is an OpenGL debugger that allows the programmer to select intermediate values in the shader for display. An intercepting library understands the semantics put in by the programmer as debugging statements, rewrites the shaders accordingly and binds them to the pipeline. The library can then read back the debug channel of the pixel values to display intermediate variables. GLSLDevil can handle regular code, conditionals, loops and function calls in GLSL. Additionally, pixels can be selected by the user for individual debug.

PIX [Wal07], in 2006, introduced a utility for debugging individual pixels called Pixel History. This feature allows the programmer to go through the shader instruction-by-instruction for all the invocations that wrote to a particular pixel. PIX can also display the state of the Direct3D runtime as well as bound and unbound resources at each point in the program's lifetime. However, unlike our approach, PIX does not allow the programmer to view the full execution history of a pixel beyond a single shader invocation.

Our proposed technique is similar to other techniques proposed earlier as far as shader instrumentation is concerned. However, we only dump the input variables (live-ins) of the shader in order to deterministically replay the execution stream on the CPU. Once all the live-ins are determined, the CPU can do what it does best, i.e., go step-by-step and execute the captured stream. We also propose a hardware mechanism to speed up the capture of the execution stream.

7. Future Work

This work could be expanded in several directions in the future. Debug history could be enhanced by including vertex shaders, geometry shaders and stream out in the emulated stream of instructions. Similarly, one could think of a way to provide a complete history of a particle for a particle system simulated on the GPU, and emulate it on the CPU for debugging purposes.

With the introduction of Nvidia's CUDA [NVI08] and ATI's CTM [ATI08] technology, GPGPU programming is on the rise. However, other than emulation, there is currently no way to debug GPGPU code. This work could be extended to provide a framework that could help in debugging faulty output. The framework could allow for capture of the entire execution stream of instructions that led to a particular final output value. This stream could then be faithfully and deterministically replayed on the CPU in order to analyze program behavior and find bugs.

8. Conclusion

We provided the design and implementation of a framework that allows a shader programmer to sequentialize instructions from different shaders that caused the final write of a pixel of interest. This pixel of interest can be specified as input coordinates or a conditional breakpoint can be put inside the shader. The application is transparent to the debugger runtime and does not have to be recompiled to use it. After obtaining the sequentialized instructions, they can be deterministically emulated on the CPU step-by-step with ease. This allows the programmer to inspect all intermediate values as the execution progresses. Our framework is especially well-suited to multi-pass shading algorithms, as the programmer can view the entire execution stream at once regardless of render state changes (render target switches, shader changes, etc.). We think that by looking at the entire execution stream which causes the final write to a pixel, the programmer will gain valuable insight and will be able to correct unintentional program bugs.

9. Acknowledgment

This research was supported in part by NSF Grant CNS-0325536, a Department of Energy Early CAREER Award, and an NSF CAREER Award CNS-0644096.

References

[ATI08] ATI: ATI researcher relations - CTM document library. http://ati.amd.com/companyinfo/researcher/Documents.html.

[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 24*, 3 (Aug. 2005), 453–463.

[DWS*88] DEERING M. F., WINNER S., SCHEDIWY B., DUFFY C., HUNT N.: The triangle processor and normal vector shader: A vlsi system for high performance graphics. In *Computer Graphics (Proceedings of SIGGRAPH 88)* (Aug. 1988), pp. 21–30.

[Mic] MICROSOFT: Queries (direct3d 9). http://msdn2.microsoft.com/en-us/library/bb147308(vs.85).aspx.

[NVI08] NVIDIA: CUDA zone - resource for C developers of applications that solve computing problems. http://www.nvidia.com/object/cuda_home.html.

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

[SKE07] STRENGERT M., KLEIN T., ERTL T.: A hardware-aware debugger for the opengl shading language. In *Graphics Hardware 2007* (Aug. 2007), pp. 81–88.

[TS05] TEBEKA Y., SHAPIRA A.: Advanced opengl debugging and profiling with gdebugger. *Game Developer's Conference* (2005).

[Wal07] WALBOURN C.: Debugging direct3d 10 applications. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses* (New York, NY, USA, 2007), ACM, pp. 299–321.

[Wil78] WILLIAMS L.: Casting curved shadows on curved surfaces. In *Computer Graphics (Proceedings of SIGGRAPH 78)* (Aug. 1978), pp. 270–274.