BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2024
http://excel.fit.vutbr.cz

GLSL Shader Debugging Toolkit

Ondřej Sabela*

Abstract

This paper presents a solution for shader program inspection and debugging. The goal is to provide a workflow similar to typical debuggers like GDB or LLDB, while also displaying information about the graphics API's pipeline state. A shared library hooking mechanism is employed to establish a connection with the target application. All function calls to the graphics API are intercepted, the important parameters are logged and used to build a snapshot of the internal graphics pipeline state. Source codes are organized in a configurable hybrid (both physical and virtual) file system. As the user performs debugging actions (e.g. source code stepping, making modifications, breakpoints), the code is rewritten, instrumented, and recompiled in the background, transparently to the user. Emphasis was placed on ensuring the solution is flexible, hardware-independent and stand-alone. The complete solution, *Deshader*, comprises a shared library, a *runner* application and an extension for Visual Studio Code, which is integrated into the library.

*xsabel03@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

The term "debugging" has become synonymous with "stepping through source code" in modern parlance. While there are some proprietary GPU debuggers for **general compute** programs, they often lack support for primarily **graphics** APIs like OpenGL or Vulkan. Until recently, Microsoft's PIX was the only notable exception, offering shader debugging since the early versions of DirectX. This may stem from the fact that standardizing debugging features is not a primary focus of the Khronos group. The most widely used open-source tool for graphics debugging, RenderDoc, was the first to support debugging for the OpenGL Shading Language, GLSL, under Vulkan, thanks to its author who proposed the `SPV_KHR_non_semantic_info` [1] and the related `NonSemantic.Shader.DebugInfo.100` [2] extensions for Vulkan and SPIR-V.

While other solutions have existed in the past (NVIDIA FX Composer, Strengert [3], Hilgart [4]), all of them have been abandoned. Their failure can be attributed to certain design incompletenesses and compromises, such as dependency on a custom compiler, overly isolated environment, and support for only some shader stages. Both the historical and contemporary solutions are generally based on one of three very different implementation approaches:

1.1 Instruction Level Debugging

NVIDIA's CUDA-GDB, AMD's ROCgdb and Intel® Distribution for GDB all leverage deep knowledge of their instruction set architecture, enabling them to create a genuine execution environment similar to traditional debuggers, providing the most detailed and deep information (e.g. occupancy) and support heterogeneous (CPU and GPU simultaneously) debugging. However, proprietary instruction level debuggers lack the support for graphical APIs, although it is practically possible (a proof-of-concept can be found in these blog posts [5]). Another approach is to **simulate** the program execution.

Microsoft offers PIX [6] (formerly Visual Studio Graphics Diagnostics) for DirectX high-level shaders (HLSL) and low-level DXBC/DXIL pseudo-assembly. RenderDoc [7] and ShadeRed [8] also make use of a virtual machine for SPIR-V pseudo-assembly execution. The simulation approach can be sensitive to differences between the virtual and physical devices, requiring continuous updates to remain synchronized with the API and extensions. There also have been criticisms regarding performance, the absence of heterogeneous debugging, and the limitations of the simulated environment. However, both methods have an advantage of full control over program execution and state.

1.2 Source Code Instrumentation

This concept essentially involves "rewriting the shaders to accept and output additional information", as demonstrated in academic works such as [3], [4] (both on GLSL source level) and Purcell [9] (on ARB assembly level). Instrumentation doesn't require knowledge or even re-implementation of the instruction-level model and can be largely hardware-agnostic. However, the implied extensive modifications to the shader code and graphics pipeline must be made with precision to avoid any unexpected behavior, which can be difficult to reliably detect. Nevertheless, this is the most flexible concept, which is why it was chosen as appropriate for Deshader. Additionally, it addresses the current gap in up-to-date solutions of this type.

2. Features of a Shader Debugger

I have defined several **fundamental features**: source code stepping, breakpoints, conditional breakpoints, logging (printf-debugging) – as can bee seen on Figure 3a; variable watching, pipeline state inspection, edit and continue. To achieve the flexibility goal, additional **secondary features** are described in the following paragraphs. These features help address specific challenges inherent to GPUs and OpenGL, the supported graphics API.

2.1 An Easy Connection and Control Interface

Controlling Deshader from the host application must be done in a way that does not cause any unintended behavior when Deshader is not plugged in. For example, the OpenGL specification [10] includes the `glDebugMessageInsert` function, which can be hooked and used as a communication channel. Additionally, the GLSL specification [11] features the `#pragma` directive, which is "not subject to preprocessor macro expansion" and can be safely ignored by the compiler. As summarized in Figure 1a, commands can thus be accepted via:

1. the `glDebugMessageInsert` function – when the ID parameter matches `0xDE5ADE[X]`,
2. `#pragma deshader` directives directly in GLSL source code, or
3. a side channel for external GUI – Deshader implements WebSocket and HTTP protocols.

Traditional debuggers typically rely on operating system calls to attach to running processes, limiting the ability to connect multiple debuggers simultaneously. To enable heterogeneous computing, Deshader uses a library loader hooking method, as illustrated in Figure 1b. On Linux, injecting a library into a program can be accomplished using the `LD_PRELOAD` environment variable, or `DYLD_INSERT_LIBRARIES` on MacOS. On Windows, simply placing a DLL with the same name as the hooked DLL into the working directory is sufficient. The helper application `deshader-run` automates this process.

2.2 Organizing Source Code

Shaders are compiled from source strings on-the-fly and OpenGL doesn't know their physical origin. Each program consists of shader stages which can be composed of multiple shader objects. Shader objects can be reused across stages and programs, similar to shared C objects, resulting in sophisticated logical relations (see Figure 2b). Deshader employs a virtual file system that supports hardlinks and mapping points to physical storage. Users can control shader names using the interfaces shown in Figure 2a.

2.3 Thread Grouping and Selection

Pausing and stepping through one thread execution can yield different results than pausing and stepping through all shader threads simultaneously. Moreover, in OpenGL, an application can create multiple isolated or cooperating graphics contexts, each with its own shaders, which should pause and continue independently. Handling context switching in a thread-safe manner is essential. Deshader exposes command interface to strictly express the user's decision regarding thread selection and grouping.

3. Instrumentation Techniques

After "start debugging" request, Deshader enters *debugging loop* (see Figure 3b). The pipeline state is captured, and shaders are then processed by a syntax parser that outputs locations of all *statements* (where a step or breakpoint can be placed), *declarations* (data objects and functions), *variable scopes*, *function calls* and **flow control** (conditionals, loops) blocks, where threads can diverge, requiring additional checks (*guards*) for previous breakpoint hits. Stepping is implemented by placing a counter increment and conditional `return` after each statement, as illustrated in Figure 3c. Breakpoints are implemented similarly, but their *guard*s are only placed after function calls and flow control blocks (see Figure 3d).

Input parameters (selected thread, next step index) are straightforward to pass to shaders using uniforms. **Outputting** data (logging, reached step, stack trace) requires either the use of a *Shader Storage Buffer Object* and an atomic counter (as a cursor for writing into the buffer), or a *Renderbuffer* when the amount of output data is constant for each thread.

Acknowledgements

I would like to thank my supervisor Ing. Tomáš Milet, Ph.D. for insights and sharing his experience with not-so-well-documented behaviors and internals of OpenGL and GLSL, various recommendations about the instrumentation implementation, and keeping me patiently on track.

References

[1] Baldur Karlsson. SPV_KHR_non_semantic_info. https://github.com/KhronosGroup/SPIRV-Registry/blob/main/extensions/KHR/SPV_KHR_non_semantic_info.asciidoc, 2020. Accessed: 20-04-2024.

[2] Baldur Karlsson. SPIR-V NonSemantic Shader DebugInfo Instructions. https://github.com/KhronosGroup/SPIRV-Registry/blob/main/nonsemantic/NonSemantic.Shader.DebugInfo.100.asciidoc, 2022. Accessed: 20-04-2024.

[3] Magnus Strengert, Thomas Klein, and Thomas Ertl. A Hardware-Aware Debugger for the OpenGL Shading Language. In Mark Segal and Timo Aila, editors, *SIGGRAPH/Eurographics Workshop on Graphics Hardware*. The Eurographics Association, 2007.

[4] Mark Hilgart. Step-Through Debugging of GLSL Shaders. School of Computer Science, DePaul University, Chicago, USA, 2006.

[5] Marcell Kiss. Making an AMDGPU debugger. https://martty.github.io/posts/radbg_part_1/, January 2023. Accessed: 2024-01-12.

[6] Microsoft. HLSL Shader Debugger - Visual Studio. https://learn.microsoft.com/en-us/previous-versions/visualstudio/visual-studio-2017/debugger/graphics/hlsl-shader-debugger?view=vs-2017&viewFallbackFrom=vs-2017%5C#understanding-the-hlsl-debugger, November 2016. Accessed: 20-04-2024.

[7] RenderDoc – an open source portable graphics debugger. https://renderdoc.org/.

[8] ShadeRed – an open source shader IDE. https://shadered.org/.

[9] Timothy John Purcell. Debugging tools. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, page 114–es, New York, NY, USA, 2005. Association for Computing Machinery.

[10] Mark Segal and Kurt Akeley. The OpenGL® Graphics System: A Specification (Version 4.6 (Core Profile)). The Khronos Group Inc., May 2022.

[11] John Kessenich, Dave Baldwin, and Randi Rost. The OpenGL Shading Language, Version 4.60.8. The Khronos Group Inc., August 2023.