Using OpenGL State History for Graphics Debugging

Bryce van Dyk, Christof Lutteroth, Gerald Weber, Burkhard Wünsche
Department of Computer Science
University of Auckland
Private Bag 92019, Auckland 1142
New Zealand
bvan036@auckland.ac.nz, {christof, gerald, burkhard}@cs.auckland.ac.nz

ABSTRACT

To fulfill the unique debugging requirements of graphics programming, specialized tools are needed to aid in the debugging process. Modern graphics debuggers allow developers to inspect the current graphics state of a running application, and influence their control flow. However, they do not make maximum use of information about previous graphics states, despite the possible utility of this information in debugging. We propose GLDebug, an OpenGL debugger with novel features for using historical information to assist with graphics debugging. GLDebug provides the ability to capture and recall OpenGL state and function call information. Developers can retrace the graphics state history of OpenGL applications and compare different recorded states, which may come from different applications. State differences are made clearly visible, so that the source of state-based errors can be tracked down more easily. GLDebug was evaluated in a user study, with promising results: the participants found the tool helped them when working on four different OpenGL debugging tasks. All participants commented favorably on the support for tracking and analyzing state history. The results indicate that historical information is useful for graphics debugging, and that debuggers supporting such information can improve debugging efficacy.

Keywords
debugging, state history, function call history

1 INTRODUCTION

Computer graphics is applied in a vast number of fields such as entertainment, medicine, and computer-aided design. With so many applications for computer graphics, there is a demand for tools that assist programmers with the analysis and debugging of graphics code. However, general purpose debuggers do not cater to the specific needs of graphics programmers.

The need for dedicated tools stems from the unique paradigms used in graphics programming, as well as limitations due to the graphics hardware. For example, when programming with OpenGL, programmers must manage the state of OpenGL, treating OpenGL as a state machine. General purpose debuggers do not offer the ability to monitor this state – a useful feature that graphics debuggers should offer. Similarly, general purpose debuggers cannot help inspect the internal state of the graphics hardware – not in the same way they do for programs running on the CPU. There are also important differences in the types of data being dealt with: graphics debuggers must consider objects such as textures and matrices, which are of particular importance in graphics programming.

Various graphics debuggers have been introduced over the last decade by commercial vendors, open-source developers and researchers. These debuggers address the problems of inspecting the internals of graphics hardware, controlling the execution flow of graphics code, and profiling it. However, their focus is on giving developers access to the current state of the graphics hardware only.

In this paper, we explore the idea of using historical information to assist with graphics debugging. We present a novel debugger, GLDebug, which provides the novel ability to capture and recall past OpenGL state and function call information. GLDebug allows developers to accumulate this historical information over time from multiple OpenGL applications, and compare it in a user interface that is similar to other history viewers. Users of GLDebug can retrace the graphics state history of OpenGL applications and compare different recorded states, making state changes clearly visible. This makes it easier, for example, to find defects in erroneous code when comparing it with working code. In particular, we are addressing the following research questions:

R1 How can graphics state history be supported in a debugger and presented to the user?

R2 In how far does the use of graphics state history facilitate debugging?

The ability to record and inspect graphics API states has been discussed in prior work [3, 5], so we only give a brief overview of this. In particular, we point out the various challenges and techniques involved in capturing the internals of graphics hardware. Then, we discuss in more detail how graphics state history can be stored, managed and presented to the user.

Previous works have not fully utilized and investigated historical information about OpenGL applications. We discuss how a graphics debugger can make this information easily available to assist in the OpenGL debugging process, addressing R1. In particular, we show how the use of historical information can be supported in a debugger's user interface, and motivate the features of GLDebug with specific use cases.

After completion of the GLDebug proof-of-concept prototype, a user study was conducted to evaluate the usefulness of the tool and address R2. This evaluation was fairly small in scale and scope, but seems to be the first of its kind: there is little or no research that attempts to evaluate the effectiveness of graphics debuggers.

Note that the results about the use of state history for debugging presented here are not only applicable to OpenGL. Our implementation is based solely on OpenGL, but other low-level graphics APIs such as DirectX are conceptually very similar. As a consequence, the contributions of this research can also be applied to other graphics APIs.

Section 2 summarizes the requirements of graphics debugging in general, and for using state history specifically. Section 3 gives an overview of related work. Section 4 introduces GLDebug and elaborates its design, including the user interface for making OpenGL state history easily accessible to developers. Section 5 details key areas of GLDebug's implementation. Section 6 explains some of the debugging use cases that can be addressed with GLDebug. Section 7 presents the results of the user study. Section 8 concludes the paper and points out some future work.

2 REQUIREMENTS

Common features of graphics debuggers include *state tracking*, *logging of graphics commands*, and the *inspection of buffers*. These features are widely used in modern graphics debuggers. In this project, we are also looking at novel features regarding the use of graphics state history, such as *logging of graphics states* and *comparison of graphics states*. In the following sections, we will describe all these features as requirements of graphics debuggers.

2.1 General Requirements of Graphics Debugging

State tracking is a functionality allowing a user to track, view, and potentially alter the state of the underlying graphics system. OpenGL is generally known to be a state machine. How this machine is configured controls many aspects of how a command to the machine is processed. Bugs can easily be introduced by having the machine configured incorrectly [9].

As an example, consider a situation where a programmer is using a third-party library that makes changes to OpenGL state. Unfortunately, the programmer is not aware of these changes and thus subsequent OpenGL calls made by the program are not behaving as expected. But even if the programmer suspects this to be the cause, they still have to track down which part of the state is being altered.

In the above example, being able to inspect state is very helpful. The simple act of seeing what the state is and comparing that against what is expected saves the programmer from having to recompile code with debug instructions inserted to inspect state, or worse yet, from having to expend time learning that the bug is even related to OpenGL state. There are also instances of complex state interaction, where it is useful to be able to inspect several state variables at once. Presenting state information in a clear and easily navigable way facilitates this.

Command logging or **call logging** refers to a debugger logging commands being issued to the graphics API, and making the log visible to the user. This feature is useful as a reference, in a similar way to viewing OpenGL state: it helps verify that the *actual* behavior of the program is the same as the desired behavior. For example, this helps to make sure that a certain function is indeed being called, or that a certain argument to a function is correct.

Another useful, though rarer, aspect of this feature is being able to replay the commands that are logged. By doing this one can recreate a scene step by step, seeing the effect that each command has (visually and/or in the graphics state information). However, implementing this functionality is technically much more difficult than just logging calls.

Inspection of buffers is the ability of a debugger to query OpenGL for information contained in buffers belonging to the program being debugged, and then to expose this information in various ways to a user. Buffers can be used to store a variety of things, but the common inspection case is buffers storing texture (image) data. That said, support for inspecting other types of buffers exists in some debuggers, e.g. for buffers containing shader input data such as vertices. The way data is exposed can be visual or numeric, with different representations being appropriate depending on the buffer contents.

For example, a debugger could retrieve and allow inspection of the depth buffer, which helps determine if an object is being culled by the Z-test. Another use case is the inspection of an off-screen texture that is being rendered to, a common technique in deferred shading/rendering [11]. Being able to visually inspect such a texture may be invaluable in seeing that the rendered image is as intended.

Shader debugging is functionality helping with shader bugs, which is becoming more and more important with the prevalence of shaders in modern graphics programming. Special support for shader debugging is necessary because of the shader pipeline being opaque: while input and output can be observed, what happens inside the pipeline is difficult or impossible to observe, making bugs that occur in the pipeline very difficult to diagnose and resolve. One of the popular shader debugging techniques is to instrument shaders so that additional information is output [14], allowing a programmer to read back the values of variables during shader execution – information that is normally inaccessible. Another technique is that of emulating the shader pipeline in software [13], allowing for much greater visibility and enabling identified requirements such as step-through debugging of shaders.

2.2 Requirements for Using State History

Our work here seeks to extend upon the ability of tracking the current state of an OpenGL program, by **tracking the state over the life** of such a program. This is similar to state tracking, with the additional requirement that captured information is persistent and is always available for recall. This contrasts with systems that only allow for viewing of the current state of a program, where previously captured information is not stored. Such concepts have been explored in the context of general purpose debuggers [10, 12], but have only been vaguely suggested for graphics debuggers [5].

In addition to tracking OpenGL state over the program execution, we also look at providing a means by which users can **compare the captured information** in a way that assists with debugging. It is important to report captured information to the users in a fashion that enables quick comparison of different states in order to facilitate the debugging process.

As an example of the above two requirements, a user should be able to record states from an OpenGL application that is running smoothly. When a bug is encountered, the user should be able to recall the state from when the program was running correctly, and compare that to the current, buggy state. The GUI should allow for a comparison such that the user is able to identify problematic states (if any).

3 RELATED WORK

3.1 Enabling Technology for Graphics Debugging

There are several technologies that enable and support graphics debugging, although they are not debuggers themselves. For example, there are systems available that aid in the capture of calls made to OpenGL, or that allow for querying of the state of OpenGL. A number of debuggers, including GLDebug, are built upon such systems.

Chromium [8] is a system for the manipulation of OpenGL command streams. Chromium uses a client-server model, with streams of commands being dispatched by clients to one or more servers from which the streams may be passed onto other servers. Each server can inspect and, if needed, modify the stream sent to it. Chromium can also be leveraged to manipulate the command streams, thus it is possible to alter the behavior of a program. These features are immediately useful in that they allow for both state tracking and command logging. However, Chromium is no longer being developed, leading to compatibility issues with recent versions of OpenGL.

BuGLe[1] is a toolkit designed to aid in the debugging of OpenGL applications. BuGLe makes use of filters that are used to intercept some or all OpenGL calls. Once a call is intercepted, it can be inspected, and modifications can also be made before the call is passed on to OpenGL. In contrast to Chromium, BuGLe is still being developed, so it has much better compatibility with more recent versions of OpenGL.

3.2 Graphics Debugging

The most actively developed graphics debuggers at present are commercial products, such as PIX[2] and Nsight[3]. There are also several academic projects in this area [7, 13], of which two major contributions are described below. However, little active research appears to be occurring in this area at the moment.

gDEBugger[4][5] was one of the first commercial graphics debuggers to become widely available in 2004. It demonstrated many of the features seen in modern graphics debuggers, such as all of the general features discussed in Section 2. Furthermore, all these features were accessible through a GUI. The contribution of gDEBugger is in its pioneering of graphics debuggers in the commercial space, as well as offering many

[1] http://sourceforge.net/projects/bugle/

[2] http://msdn.microsoft.com/en-us/library/ee663275%28v=vs.85%29.aspx

[3] http://www.nvidia.com/object/nsight.html

[4] http://developer.amd.com/tools/gDEBugger/Pages/default.aspx

[5] http://www.gremedy.com/

features incorporated with a GUI. gDEBugger development has been discontinued as it became part of another debugger, CodeXL, which is discussed below.

Microsoft PIX is a commercial graphics debugger for use with DirectX on Windows as part of the Xbox 360 development kit, which is actively maintained by Microsoft. It is one of the few tools available for DirectX debugging. A notable feature of PIX is its ability to capture all of the commands used to create an image (a frame), and then replay these commands step by step on demand.

nVidia Nsight and **AMD CodeXL**[6] are further examples of modern commercial debuggers. These tools provide many of the features mentioned in Section 2, including newer features for shader debugging, similar to those seen in GLSLDevil (see below). While they are available free of charge, their usage is limited to their developer's respective hardware.

There are tools that log calls made to graphics APIs such as OpenGL, e.g. **glintercept**[7] and **apitrace**[8]. These tools log the API function calls made by an application to a file, and allow users to inspect this log, e.g. for profiling. Some of these tools (e.g. apitrace) also allow users to replay the log files and inspect the current graphics state during replay, similar to a graphics debugger.

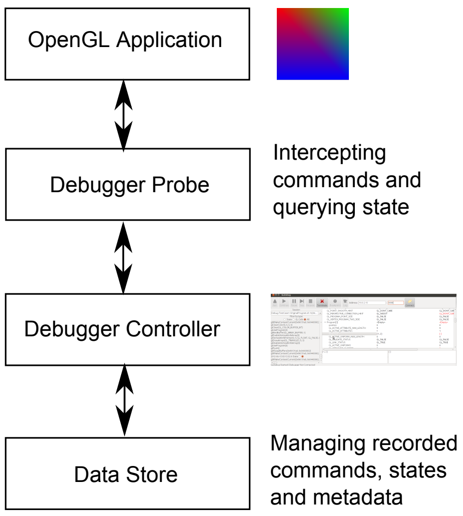
GLSLDevil[9] [14] is a tool specifically aimed at debugging the shader pipeline of OpenGL applications. GLSLDevil provides novel features in that it automatically instruments OpenGL shader code. The instrumented code then outputs extra information that can be used for debugging. GLSLDevil uses a GUI to present this information to users, showing the values of the variables used in a shader. It also supports some visualizations of those values, e.g. as images.

Apart from command logging and playback, historical information is not supported in any of the currently available debugging tools. A possible reason for this is that the storage and computation requirements make it non-trivial [10, 12]. Furthermore, the current research on graphics debugging exhibits a lack of evaluations of graphics debugging tools and their use in practice, which may make it an uncertain area to prioritize for development.

3.3 Debugging using History

The concept of recording the state of a program throughout its execution has been proposed for general-purpose debugging [10, 12]. The research in this area speculates that the ability to step back through a programs trace aids the user in certain debugging tasks. For example, such debuggers can help when a bug is found that is tied to a variable with an incorrect value. In this scenario, the debugger can be used to step backwards in time and find at what point the value deviated from appropriate values. Graphics debugging has some similarities to such general-purpose debugging scenarios: bugs often originate from some unintended state change [9], which is identified by inspecting the execution flow. However, the state machine aspect of graphics debugging is typically much stronger, with a reliance of outputs on a complex state and different types of potential bugs. Also, the technology involved in graphics debugging is different.

GQL (graphics query language) was created along with a debugging system by Duca et al. [5]. Similar to GLDebug, it enables tracking and logging the state and calls made by an OpenGL program over the course of execution. However, the historical information is only made available through an SQL-like language (GQL) that users have to learn, and there is no direct support for comparing states and highlighting of state differences. It is known that efficient use of a query language such as SQL depends strongly on individual ability and the user interface [4], hence it is questionable whether a textual query language such GQL can adequately support day-to-day graphics debugging tasks. GQL was not evaluated empirically to see if users find this approach effective or user friendly.


OpenGL Application
Debugger Probe
Intercepting commands and querying state
Debugger Controller
Data Store
Managing recorded commands, states and metadata
Figure 1: Architectural overview of GLDebug.

4 DESIGN

GLDebug is designed based on several high-level components, as shown in the architecture diagram in Figure 1. The *OpenGL application* is the program being debugged. It is executed on top of the *debugger probe*,

[6] http://developer.amd.com/tools/heterogeneous-computing/codexl/
[7] http://code.google.com/p/glintercept/
[8] http://apitrace.github.io/
[9] http://cumbia.informatik.uni-stuttgart.de/glsldevil/

which is a library that intercepts the OpenGL calls made by the application. Intercepting these calls makes it possible to capture information about the calls themselves as well as other data that can be inspected while the intercepting library has control.

The probe feeds the information it gathers into the debugger *controller*, which provides the GUI for controlling the debugging process. Through the probe, the controller can request graphics state information and influence the control flow of the application being debugged. The controller is also used to present information about the application to the user, and in particular let the user access the graphics state history in a convenient way. To support state history, the controller stores graphics states, OpenGL commands and related information in a *data store*. The data store is queried whenever historical information is needed. In the following paragraphs, the components of GLDebug are described in more detail.

4.1 Probe

The probe is the component responsible for capturing data from the program being debugged, and feeding that data to the controller. It is a shared library that provides the same interface as OpenGL. When a program is run, the probe is linked instead of the default OpenGL library. This means that all calls that would normally be made to the OpenGL library are passed to the probe instead. The probe allows for arbitrary code to be executed once a call is intercepted, hence taking over program control and allowing for both inspection and modification of OpenGL calls. It can process commands sent to it from the controller, such as for pausing the application, and send data to the controller, such as graphics state data that is queried by executing additional OpenGL commands.

There are several benefits of having the probe as a separate component of the system. For example, GLDebug can run on a computer separate from the computer running the OpenGL application. This provides benefits in terms of being able to run the probe and debug OpenGL applications on systems with less power and/or storage, such as mobile devices. Also, the probe can be developed independently of the other components. The downside is that there is additional work involved in developing a communication protocol for the probe and the controller.

The probe is lightweight, does not perform much processing and does not impede the OpenGL application. It is important that the probe does not alter the behavior of the OpenGL application. Similar designs can be found in other debuggers, such as the GQL debugger mentioned in Section 3, which has a separate process that processes the data captured from an application.

4.2 Controller

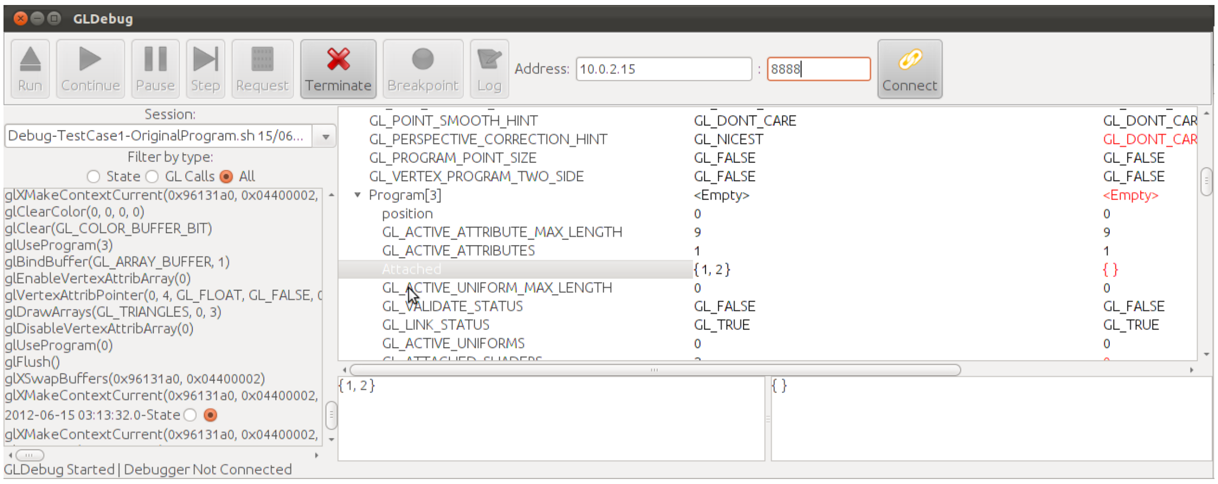
The controller is responsible for controlling the running OpenGL application and retrieving information about it through the probe. It is also responsible for storing the information in the data store, and making it accessible to the user through a GUI. Because of the distributed architecture of GLDebug, the controller and data store can be hosted on a more powerful system.

Figure 2 shows the controller GUI. The buttons at the top allow users to connect to a running probe and influence the control flow of the application being debugged, i.e. start, pause, stop and step through it. Furthermore, they allow users to set breakpoints on specific OpenGL functions, and request the graphics state from the application. The GUI also presents captured information back to the user. Graphics state information is presented in the right section of the window, below the top row of buttons. The table lists all OpenGL states variables with their values, and there is space at the bottom to show the value of a selected variable in more detail, i.e. in the case of longer state variables such as shader source code.

Note that the table on the right shows two graphics states, one in the left column and one in the right column. Differences in these two states are highlighted using color coding: unchanged variables are shown in black, variables with different values are highlighted in red, and if the values are the same but there has been a recorded state between the first and second state where the variables are not the same, then they are shown in purple. This allows users to quickly compare two graphics states. The states to compare are selected in the list on the left, which shows, among other information, the sequence of recorded states. The two columns of radio buttons are used to select the two states that are shown in the table on the right. The drop down list at the top lets users select different application sessions to view data from. So a user can select a state snapshot from one execution of a program, and compare it to another, or even compare state snapshots from two different programs.

Finally, the controller can show users a list of function calls that were captured by the probe. As shown in the radio button group near the top-left, the user may select to see only states, only function calls, or both together. This allows the user to explore the history of all captured states and function calls over the lifetime of the application.

Originally, the GUI had a multi-tab design where separate tabs were used to control the probe, and to view and compare captured information. However, this design was discarded in favor of the current single-window design after initial user feedback. Users found a multi-tab window to be too cumbersome as it required a lot of switching between tabs.



Address: 10.0.2.15 : 8888

Session: Debug-TestCase1-OriginalProgram.sh 15/06...

Filter by type: State GL Calls [✓ All]

```
glXMakeContextCurrent(0x96131a0, 0x04400002,
glClearColor(0, 0, 0, 0)
glClear(GL_COLOR_BUFFER_BIT)
glUseProgram(3)
glBindBuffer(GL_ARRAY_BUFFER, 1)
glEnableVertexAttribArray(0)
glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0
glDrawArrays(GL_TRIANGLES, 0, 3)
glDisableVertexAttribArray(0)
glUseProgram(0)
glFlush()
glXSwapBuffers(0x96131a0, 0x04400002)
glXMakeContextCurrent(0x96131a0, 0x04400002,
2012-06-15 03:13:32.0-State
glXMakeContextCurrent(0x96131a0, 0x04400002,
```

GL_POINT_SMOOTH_HINT	GL_DONT_CARE	GL_DONT_CAR
GL_PERSPECTIVE_CORRECTION_HINT	GL_NICEST	GL_DONT_CAR
GL_PROGRAM_POINT_SIZE	GL_FALSE	GL_FALSE
GL_VERTEX_PROGRAM_TWO_SIDE	GL_FALSE	GL_FALSE
Program[3]	<Empty>	<Empty>
position	0	0
GL_ACTIVE_ATTRIBUTE_MAX_LENGTH	9	9
GL_ACTIVE_ATTRIBUTES	1	1
Attached	{1, 2}	{ }
GL_ACTIVE_UNIFORM_MAX_LENGTH	0	0
GL_VALIDATE_STATUS	GL_FALSE	GL_FALSE
GL_LINK_STATUS	GL_TRUE	GL_TRUE
GL_ACTIVE_UNIFORMS	0	0

{1, 2}

{ }

GLDebug Started | Debugger Not Connected

Figure 2: The GLDebug interface comparing two sets of captured state.

4.3 Data Store

The data store archives the OpenGL state information that is captured by the probe. This information consists mainly of state variables with names and values, with each graphics state containing hundreds of such variable-value pairs. Some variables are nested, i.e. they have child variables with values.

The data store also archives the function calls made by the OpenGL application that were logged by the probe. This includes the function name, parameter names and values, as well as the call order. Finally, the data store stores metadata about the logged information. This includes identifying information about the application being debugged and the debugging sessions, as well as timestamps for debugging sessions, states and function calls.

Our design makes use of a temporal database that performs delta encoding on stored information automatically. That is, when storing state information, it stores only the values of states that have actually changed. This means different states can be stored and recalled with minimal overhead, and comparison between the different states is somewhat simplified.

5 IMPLEMENTATION

GLDebug's probe was implemented using BuGLe (see Section 3.1) as a basis. The complexity and time requirements of implementing a debugger from scratch are significant. Using BuGLe as a basis greatly decreased the time required to develop the probe and implement the ability to capture OpenGL commands and state. However, BuGLe still had to be extended to meet the needs of GLDebug, e.g. with functionality for logging and sending information about OpenGL commands.

All communication between the probe and the controller is done through a single TCP connection. This allows the probe to run on the same system as the controller or on another system, as required. The communication is primarily initiated by the controller, issuing requests to the probe, such as those for state, or those to start or stop the execution of the OpenGL application. When the probe receives a command, it attempts to carry out that command and reply to the controller as necessary. There are some cases where the communication is initiated by the probe, e.g. the sending of logged function calls.

The data store was implemented using a temporal triple store called PDStore, which was developed in our working group in a separate project. PDStore's ability to recall previous database states makes it possible to access any of the previously stored OpenGL states. Per default, the controller uses PDStore as an embedded database, so both run in the same process. As with many database systems, it is also possible to connect the controller to a remote PDStore database.

The controller was implemented using Java, while the probe had to be coded in a lower-level language (in this case C) in order to be compiled into a shared library. This separation was helped by the fact that both components communicate over a remote interface based on TCP, as explained earlier.

The implementation of the probe was fairly demanding, even when considering the use of BuGLe as a basis. It required extending BuGLe for capturing extra information and sending extra data, which required a detailed understanding of BuGLe's internals. Furthermore, a deeper understanding of linking was required in order to make sure that BuGLe was linked instead of OpenGL. For a more in-depth view of GLDebug's implementation see [16].

6 USE CASES

In the following we describe important use cases for the use of state history during graphics debugging. We describe what kind of bug is involved in a use case, its significance in real-world graphics applications, and how state history can help to find the bug more easily.

6.1 Incorrect Graphics State

GLDebug is useful in cases where bugs are caused by incorrect OpenGL state, and particularly in cases where the state shifts from intended to unintended values (so-called ”snake in the grass” style bugs [10]). In cases where OpenGL is configured incorrectly, GLDebug makes it easier to view the values of state variables and thus find problems. However, GLDebug is of particular usefulness in the case where the state was configured correctly, and then shifts to an incorrect configuration. In such cases, being able to compare states can reveal not only the incorrect variable, but also shows in which state snapshot and at what time the problem occurred.

An example is a program that is rendering correctly, but then, through programming error, switches to an incorrect shader that results in a blank screen. In this scenario a comparison of the state captured when the program was performing correctly and incorrectly, respectively, would show that the shader source code is different. The user could then examine the source code from each of the different captured states, revealing that the incorrect code is being used in the error case.

Many bugs in OpenGL are caused by incorrect state [9], and being able to easily view OpenGL state is useful in diagnosing such bugs. These kinds of bugs can differ significantly in their severity depending on which and how many state variables are incorrectly set.

6.2 State Leakage

State leakage is a specific kind of incorrect configuration of state, where some code configures the state that then affects code elsewhere in a program. There are two major issues with this: first, the source of the issue is removed from the code where the problem occurs, making the bug hard to find. Secondly, the code causing the problem may not be available to the developer; for example, it may be part of a linked library. In addition to the above, while it may be apparent that a leak is happening, it is not always apparent which state variable(s) are being leaked and causing problems.

An example of this kind of bug is usage of an external library to draw a model using OpenGL. However, the library used has a bug in that it alters and does not reset the model-view matrix before returning control to the calling code. In this scenario, through no fault of the programmer using the library, bugs are introduced.

The use of external libraries is very common in graphics programming. There are numerous graphics libraries that build on OpenGL and other grahics APIs, and with the continuing developments in processing power and computer graphics techniques, many of these libraries are subject to continuous change. Particularly for larger projects, it is rare that a single developer knows all the code in which graphics state is changed. As a result, state leakage problems can happen fairly easily.

GLDebug aids in these circumstances by making state information from different points in the application readily accessible, allowing users to find where and which state variables are being leaked. Being able to capture a state snapshot before and after the state leak allows users to use the state comparison features of GLDebug to identify the variables that have changed, and locate problematic state changes.

6.3 Missing Error Handling

OpenGL produces its own kind of errors, which require their own kind of error handling code. Without this error handling code, many errors would pass silently. An example is a compilation failure of a shader – something that would silently fail without error checking code, and simply lead to an incorrect output.

Programs often lack sufficient error handling code [17], and sometimes such code is omitted altogether. This can be particularly dangerous if errors happen silently and can lead to later problems, which is often the case with OpenGL. When these errors go undiagnosed, only to lead to problems later, finding the place where the error actually occurs can be particularly time consuming.

GLDebug simplifies catching of errors raised by OpenGL, meaning that such errors can be discovered even if a programmer has omitted error handling code. GLDebug can pause the execution of the program when OpenGL raises an error, and display information about the error to the user. By being able to catch such errors when they occur, GLDebug reduces their impact.

7 EVALUATION

A user study was performed to evaluate GLDebug and investigate in how far the use of graphics state history actually facilitates debugging. Interestingly, there do not seem to be any published studies on the usability of graphics debuggers at the moment. Our user study provides some insight into graphics debugging in general, and assesses the efficacy of the support for state history in GLDebug. It also serves as a building block for future studies in that area.

7.1 Methodology

In this evaluation participants were asked to complete graphics debugging tasks with and without GLDebug. By letting them use GLDebug for some tasks but not for others, all participants got an impression of how useful GLDebug can be. A mixed-methods approach was used to collect data during this study:

- **Think-aloud protocol**: While working on the debugging tasks, participants were encouraged to speak out their thoughts aloud and make comments at any point.
- **Observations**: Participants were observed throughout the tasks, and significant observations were recorded.
- **System Usability Scale (SUS)**: After performing the debugging tasks, participants were asked to fill in the System Usability Scale [2] (a common usability questionnaire based on Likert-scales).
- **Likert-scale questions**: Five custom Likert-scale questions were used for evaluating specific features of GLDebug.
- **Open questions**: Open questions were used to ask what users liked and disliked about GLDebug, about improvements they could think of, and any other comments they may have.

Initially, also task completion times were recorded. However, this revealed one of the challenges when evaluating domain-specific tools for complex tasks, such as graphics debuggers: the performance of individual participants varied strongly, depending on how much graphics programming experience and programming skills they had, and other personal factors. This did not only introduce a lot of noise into the measurements, but also meant that some participants took an excessive amount of time to complete the tasks. Therefore, measurement of task completion times was abandoned after a few participants, and a maximum time of 15 minutes was allocated for each task. If a participant did not complete a task in the allocated time, the solution was presented and the participant could comment on it. To get meaningful results from quantitative measures such as task completion time, a lot of training would have to be incorporated into a study, or participants would have to be selected more carefully with regard to their graphics programming skills, to create a more homogeneous sample.

Each participant performed four debugging tasks: two with and two without GLDebug (i.e. using only text editor and compiler). Each task was performed by about half the participants with and the other half without GLDebug. The tasks were performed in the order presented below. The tasks were designed to each incorporate a single and unique bug. This helped us cast light on the utility of GLDebug for different kinds of bugs, and reduced any learning effects between the tasks that may have made tasks easier than usual. The tasks were modeled on real-world problems, but smaller in scale to allow for them to be solved in an appropriate timeframe. The four bugs involved were:

1. Incorrect graphics state: An incorrectly configured Z-buffer, resulting in an output with a polygon that has clipping issues.
2. State leakage: A call to an external library (for which the source code is not available) leaves texturing enabled, resulting textures being applied to polygons not intended to be textured.
3. Missing error handling: A shader is not compatible with the shader model of the VM being used for the test, so that the shader is not being compiled and used, and the resulting polygon not colored correctly.
4. Incorrect graphics state: An incorrectly configured model-view matrix that causes the output to be drawn progressively further and further away from the camera, instead of remaining static as desired.

Before undertaking the tasks, participants were given general training on the use of GLDebug, as well as a briefing on each task, in the form of instructional videos. Participants were encouraged to give verbal feedback during the tasks, and following completion of a task. Following completion of all the tasks, participants were given the questionnaire to complete.

7.2 Results and Discussion

There were 7 participants, all of whom were male Computer Science postgraduate students. All but one had completed at least one course on Computer Graphics and had some experience in using OpenGL. Some had more extensive OpenGL project experience (more than a year of OpenGL development). The participants varied widely in both their general programming experience and their OpenGL programming experience. As discussed previously, this variation prevented us from using performance measures to assess the utility of GLDebug, but did provide us with the perspectives of users with different skill levels.

The results of the study were generally positive, indicating that participants found GLDebug useful for the tasks. Participants indicated that they found GLDebug especially useful when there were conspicuous state differences, or when they had a clear idea of what state variables to inspect. The less experienced users in particular were sometimes not sure which state variables were related to an issue, so they found it difficult to identify the relevant state changes. Users indicated in both the Likert-scale and open-ended questions that they liked the ability to compose a view of state over the course of program execution. However, users indicated they would like more flexibility and automation in how state was captured. In summary, people found GLDebug useful when it was clear how they could leverage

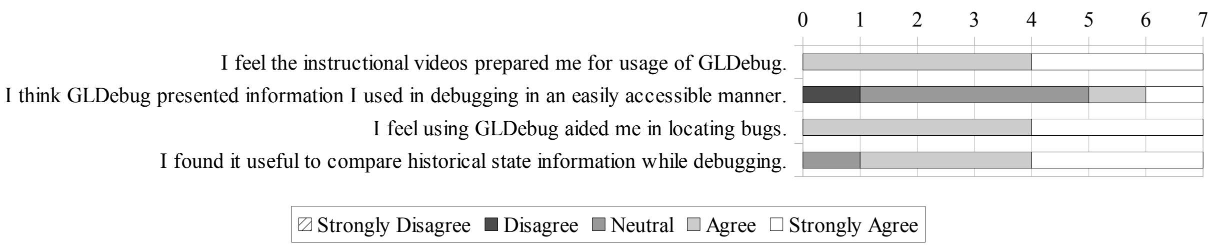


Figure 3: Participant responses to additional Likert-scale questions about the experience with GLDebug.

state information to debug a problem. Our results indicate that graphics state history can aid in debugging when users find information within that state that is clearly applicable to the problem at hand.

The average system usability scale score for GLDebug was 68.2, which is around average [1], and indicates that no serious usability issues are present. The participants suggested various improvements (see below), which helps to explain why the system got only an average score. For a research project such as GLDebug it is to be expected that it is not as polished and exhaustive in its functionality as a commercial product. The main point for this study was that debugging with state history was sufficiently supported.

Figure 3 shows the results of the additional Likert-scale questions. Q1 indicates that the instructional videos shown as training were perceived as sufficient. Q2 is in line with the results of the SUS, indicating no serious issue, but also indicating room for improvement in the presentation of information, which is discussed below. Q3 indicates that all users found GLDebug useful for debugging, which is a promising result for the prototype. Furthermore, Q4 shows that most users found the ability to compare captured state information useful.

The improvements suggested by the participants ranged from improvements to the GUI to thoughts on extra data that could be logged by the probe. Much of the feedback differed between the participants; for example, a common suggestion was making the GUI behave like an IDE the participant was familiar with. However, a strong majority of participants stated in the open question section that they wanted greater control over the ability to filter the information presented. Another desired feature mentioned in the open questions was the ability to automatically capture states based on certain conditions, such as each frame, or when a certain function call occurs. Filtering and conditional state capture would help to reduce the amount of information to that which is relevant for a specific bug. Participants also indicated a desire for functionality to show the original source code (if available) where an OpenGL call occurred, indicating the importance of putting the information provided by the debugger into proper context.

Our study has some limitations: a small sample size, a lack of professional graphics developers among the sampled participants, and possible order effects. Small sample sizes are generally acceptable for qualitative usability studies, as experience shows that most usability problems can be identified even with few participants [6]. Furthermore, there is evidence that senior Computer Science postgraduate students as participants are a reasonable approximation of performing an experiment with software professionals [15]. Each task dealt with a different kind of bug to reduce learning between tasks, and training was given before undertaking the tasks to reduce the impact of learning. However, as all participants performed the tasks in the same order, it is possible that later tasks became easier. As the study was mostly qualitative, we do not consider this a severe problem. In conclusion, this study does provide evidence for the benefits of state history, but it should be validated with a larger sample taken from professional graphics programmers, or at least people with more extensive training and experience in graphics programming.

8 CONCLUSION

In this paper we investigated history-based graphics debugging – a practice that has remained largely unexplored in previous work. We illustrated how state history can be supported in a graphics debugger, and provided some empirical evidence for its utility. In summary, we have made the following contributions:

- The design and implementation of GLDebug, a graphics debugger with features for working with graphics state history.
- A discussion of use cases for history-based graphics debugging, and how they are supported by GLDebug.
- An evaluation of GLDebug, indicating that features for comparing historical states are useful.

Overall, historic state and call information seems to be useful for graphics debugging, and the evidence indicates that it would be a good idea to extend mainstream graphics debuggers with features similar to those of GLDebug. Our study also indicates that state history would be even more useful when combined with features for filtering it, to narrow down the flood of data to relevant states. Another potential way to improve the use of state history is a better visualization of historical information. Filtering functionality and visualization of information are known to play a role for general-purpose debugging, so it would be interesting to investigate how they can further improve the use of graphics state history. Another area of interest is expanding the ability to specify when to capture states, such as capturing after particular OpenGL functions, or after drawing a particular entity.

9 ACKNOWLEDGMENTS

We would like to acknowledge the following people for their contributions to the GLDebug project: Bruce Merry, Meng-Da Lin, Osama Sagar, Heinrich Strauss, and Chen Xiliang.

REFERENCES

[1] A. Bangor, P.T. Kortum, and J.T. Miller. An empirical evaluation of the system usability scale. *Intl. Journal of Human–Computer Interaction*, 24(6):574–594, 2008.

[2] J. Brooke. SUS-a quick and dirty usability scale. *Usability evaluation in industry*, 189:194, 1996.

[3] I. Buck, G. Humphreys, and P. Hanrahan. Tracking graphics state for networked rendering. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 87–95. ACM, 2000.

[4] Steven S Curl, Lorne Olfman, and John W Satzinger. An investigation of the roles of individual differences and user interface on database usability. *ACM SIGMIS Database*, 29(1):50–65, 1997.

[5] N. Duca, K. Niski, J. Bilodeau, M. Bolitho, Y. Chen, and J. Cohen. A relational debugging engine for the graphics pipeline. In *ACM Transactions on Graphics (TOG)*, volume 24, pages 453–463. ACM, 2005.

[6] Laura Faulkner. Beyond the five-user assumption: Benefits of increased sample sizes in usability testing. *Behavior Research Methods, Instruments, & Computers*, 35(3):379–383, 2003.

[7] Q. Hou, K. Zhou, and B. Guo. Debugging gpu stream programs through automatic dataflow recording and visualization. In *ACM Transactions on Graphics (TOG)*, volume 28, page 153. ACM, 2009.

[8] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. Kirchner, and J. Klosowski. Chromium: a stream-processing framework for interactive rendering on clusters. *ACM Transactions on Graphics*, 21(3):693–702, 2002.

[9] Mark J Kilgard. Avoiding 19 common opengl pitfalls. In *Game Developer's Conference, Proceedings*, 2000.

[10] Bil Lewis. Debugging backwards in time. *CoRR*, cs.SE/0310016, 2003.

[11] T. Möller, E. Haines, and N. Hoffman. *Real-time rendering*. AK Peters Ltd, 2008.

[12] Guillaume Pothier, Éric Tanter, and José Piquer. Scalable omniscient debugging. In *ACM SIGPLAN Notices*, volume 42, pages 535–552. ACM, 2007.

[13] Ahmad Sharif and Hsien-Hsin S Lee. Total recall: a debugging framework for gpus. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 13–20. Eurographics Association, 2008.

[14] M. Strengert, T. Klein, and T. Ertl. A hardware-aware debugger for the OpenGL shading language. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 81–88. Eurographics Association, 2007.

[15] Mikael Svahnberg, Aybüke Aurum, and Claes Wohlin. Using students as subjects - an empirical evaluation. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, ESEM '08, pages 288–290. ACM, 2008.

[16] Bryce Van Dyk. Using opengl state history for graphics debugging. Master's thesis, The University of Auckland, New Zealand, 2012.

[17] Westley Weimer and George C Necula. Finding and preventing run-time error handling mistakes. In *ACM SIGPLAN Notices*, volume 39, pages 419–431. ACM, 2004.