Universidade do Minho

Escola de Engenharia

Departamento de Informática

Master Course in Computing Engineering

André Alexandre Wang Liu

3D Application Debugging

Master dissertation

Supervised by: António José Borba Ramires Fernandes,

Braga, December 21, 2014

ACKNOWLEDGEMENTS

I would like to sincerely and gratefully thank my adviser António José Borba Ramires Fernandes for all the guidance, patience and understanding in my studies at Universidade do Minho. His interest and help in this thesis was crucial and without it I would not have accomplished this work.

I would also like to thank all members of Departamento de Informática for all the knowledge I accumulated during this course.

I would like to thank all my colleagues in AgroSocial for sticking with me and giving me time to make the necessary studies while I worked along in our collaborative project.

I also would like to thank all the colleagues I met during my years in Universidade do Minho for all help provided during my student years.

Finally I would like to thank my family for supporting my life here in Campus Gualtar for it is their hard work that I could sustain myself so far and it is their hard work that I have reached so far.

ABSTRACT

It is rare for a bugless program to exist, this includes 3D applications with their respective shaders. In particular shaders are harder to debug than common applications, since they are loaded to the GPU and executed in thousands of smaller threads simultaneously. It is not easy to obtain variables values, the application state and it is hard to detect what causes errors. That's why it is necessary to study and develop debugging environments for these applications.

OpenGL in particular has many open source debuggers. A study about the features and usability of Bugle, Apitrace, GLIntercept, glslDevil and VOGL is documented, hopefully helping the reader to select the best tool for his needs. An analysis of the inner workings of each of these tools was also performed. Furthermore, the appendices allow the reader to use this document as a user manual.

Being a debugger for an API that is constantly evolving is not an easy task, hence, the issue of upgradeability is highly relevant. This study examines how each tool copes with OpenGL's evolution, in particular how each tool deals with new extensions and OpenGL versions.

Also a study on commercial debuggers from known companies such as AMD and NVIDIA was performed. While it is expected that these debuggers are more capable in general than their open source counterparts, this potential can only be fully explored in the respective graphics hardware. On the downside it is not possible to alter these debuggers and integrate them in another application.

The goal of this section is not only to analyse the potential of these proprietary tools, but also to understand the real value of the open source debuggers.

Nau 3D engine, developed at Universidade do Minho, is an engine which renders projects written with xml. Being capable of combining rasterization (OpenGL) and ray-tracing (NVIDIA's Optix) in multipass projects, it is a complex application that greatly benefits from having as much debugging features as possible.

Adding debugger features could help immensely all who work with the engine, helping both the engine developers to discover bugs in the source code, and the engine users to find bugs in their own projects.

With the knowledge gathered by studying OpenGL debuggers, several debugging features were implemented in *Nau* 3D engine. Some of these features are configurable, and maintaining the spirit of the original project, this configuration is also written in XML.

RESUMO

É muito raro existir um programa sem *bugs*, incluindo aplicações 3D com *shaders*. Os *shaders* em particular são mais complicados que as aplicações "comuns", uma vez os *shaders* são carregados para a placa gráfica e são executados em milhares de pequenas *threads* em simultâneo. Não é fácil obter valores de variáveis, estado da aplicação, e descobrir causas de erros. Consequentemente é importante utilizar *debuggers* especializados para este tipo de ambientes.

O OpenGL em particular tem vários *debuggers open-source*. O estudo realizado sobre a usabilidade e versatilidade deste *debuggers* é documentado nesta tese. Os d*ebuggers open-source* analisados são: Bugle, Apitrace, GLIntercept, glslDevil e VOGL. É também descrito o processo de instalação e utilização em apêndices permitindo que este documento também possa ser usado como um manual de utilizador.

Um estudo breve do código dos *debuggers* mencionados é necessário para entender as bases necessárias para *debug* de uma solução OpenGL. Este estudo também é necessário para compreender a questão da actualização destes projectos para acompanhar a evolução do OpenGL.

Também um estudo dos *debuggers* comerciais actuais de empresas conhecidas como AMD e NVIDIA é efectuado de forma a conhecer os debuggers comerciais actuais, o procedimento da documentação terá algumas semelhanças com os *debuggers open source*. Este estudo tem também como objectivo permitir avaliar sobre o valor real dos projectos *open-source*.

Nau é um motor 3D, desenvolvido na Universidade do Minho, para OpenGL que faz renderização de projectos escritos em XML. A possibilidade de ter um debugger interno pode ajudar imensamente todos os que desejem trabalhar com este motor, permitindo aos programadores do motor perceber possíveis *bugs* a acontecerem dentro do motor e também aos utilizadores do motor encontrarem *bugs* dos seus projectos.

Com base no estudo sobres os *debuggers open-source* OpenGL, são implementadas funcionalidades de *debugging* para o motor 3D *Nau*. Mantendo o espírito original do projecto, estas funcionalidades poderão ser configuras em XML.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Part I

INTRODUCTORY MATERIAL

INTRODUCTION

Bugs are known by anyone who writes computer programs. Even the common user will acquire such concept simply by using software.

Because of such bug omnipresence debugging tools are essential to help the programmer. A properly used debugger can save many hours of hard work.

In particular, 3D application are hard to debug, since there is code running in two separate processors, CPU and GPU. Graphic programs, a shader pipeline, are processed and executed in the GPU processor with a distinct memory space, and in multiple threads, making them particularly hard to debug. Furthermore, bugs can also appear in the pipeline construction itself.

That separates 3D application debuggers from standard debugging tools such as the already inbuilt debuggers in popular known IDE, such as Visual Studio.

This thesis presents a study on debugging tools, both open source and proprietary. It also shows how the integration of an open source debugger in a 3D application can bring added value to developers and users.

1.1 CONTEXTUALIZATION

3D applications are prone to error for a number of reasons. Probably the most common reason is the mathematics behind 3D graphics which is complex and hard to trace in a multi stream processor environment such as the GPU. This may require the user to output partial results to output buffers and latter inspect them. This inspection is not straightforward to achieve in a regular debugger since these buffers live in the GPU memory space, and it is up to the programmer to retrieve them. The multi stream nature of the GPU also makes it harder to debug a particular instance of a shader.

Another reason lays on the drivers themselves. While the specification is unique, it is a known fact that there are significant differences between the implementations from the two main hardware makers. For instance, currently, in some uniform block configurations, NVIDIA (driver version 344.11) reports block sizes that don't match the values reported by AMD (version 14.9). Furthermore, not all features described in the specification are implemented. The same applies to OpenGL extensions, with different drivers having different degrees of implementation completeness. The third issue relates to the silent way drivers deal with many errors. When an error occurs usually life goes on in the

application. Furthermore, it is up to the programmer to retrieve the compilation and linkage logs, and check for errors during execution.

Recently, OpenGL has came up with an extension dedicated to debugging [Khr14a]. The goal is to provide the user with feedback when invalid operations are performed. Although a step in the right direction, it is still far from perfect. Not all problematic situations are covered by this mechanism, and the debug messages provided by the different hardware manufacturers are far from helpful in most cases.

Each of the hardware vendors provides a debugging tool, at least for Windows operating system. NVIDIA released NSight [NVIa], and AMD has CodeXL [AMD13]. Both debuggers can work integrated with Visual Studio. While the list of features is impressive, including shader code tracing for NSight, and GPU memory inspection, and despite being supported by large corporations, these tools are not up to date with the latest version of OpenGL. CodeXL claims to support OpenGL 4.3 while NSight only supports OpenGL 4.2. Furthermore, NVIDIA Optimus equipped laptops do not to take advantage of the full list of features available in NSight, namely shader code tracing.

Open source tools on the other hand are not as powerful as they don't allow shader tracing and are not as complete regarding profiling capabilities. However, currently some are up to date with the latest OpenGL version and extensions making them useful for users who want to explore the latest features. It is also possible to integrate these tools within an application. Being open source allows for the required customizations to be performed. Another benefit is that these debuggers are not restricted to a particular hardware vendor.

1.2 MOTIVATION

As mentioned before debugging 3D applications is particularly hard. Hence, 3D debuggers are highly desirable tools. However, although open source debuggers have existed for some time there is no work detailing their features, weakness, and performing a comparison between them. This thesis attempts to fill that void. This thesis also covers commercial debuggers from NVIDIA and AMD for completeness and to provide a measure of the real value of their open source counterparts.

The following open source debuggers, mentioned in the OpenGL wiki [Ope12], shall be covered in this work: Bugle[MB07b], Apitrace[FJea13], GLIntercept[Tre13], GLSLDevil [KS10] (renamed as GLSL-Debugger [HX13]). An addition to the wiki's list could be the more recent Valve's OpenGL debugger, VOGL, which is also covered in this work. All these debuggers are being actively maintained.

The study of the open source debuggers will also provide insight on their inner workings.

Each of the debuggers has a rich set of features, though none of them as the superset of features. Bugle is a tool for OpenGL debugging implemented as a wrapper for Unix like systems and has its own GUI. Apitrace is capable of debugging on different platforms and different 3D APIs, although in this thesis the focus is on OpenGL. GLIntercept is simple and easy to use, and it is the easiest to

expand due to its plugin architecture. GLSL-Debugger has one of the most user friendly GUI. VOGL has features very similar to Apitrace, it also has a GUI that can rival GLSL-Debugger.

The second goal of this thesis is to create a debugger for *Nau* 3D engine [Rama]. *Nau* 3D engine is a 3D rendering engine developed at Universidade do Minho which allows hybrid rendering, i.e. it is capable of performing rasterization, using OpenGL, and ray tracing, using NVIDIA's Optix and Optix Prime [NVIb]. *Nau*'s Projects are created in XML files. However, the projects themselves, being highly flexible, can have bugs which will eventually require debugging, thus the objective of this thesis. On the other hand, implementing a debugger within *Nau* will also mean to implement a debugger that debugs the 3D engine itself helping the engine developers.

In this work an open source debugger, GLIntercept, was integrated with *Nau* and its GUI producing a richer debugging environment that will assist both *Nau* developers and users.

1.3 DOCUMENT STRUCTURE

The second chapter of this thesis will mainly focus on the analysis of the mentioned debuggers. The open source debuggers will also feature a sub section dealing with upgradeability regarding OpenGL versions.

Afterwards, in chapter 2.8, a comparison between the mentioned debuggers is made in form of tables, these tables pretty much serves as a conclusion for the study of the state of art. Those who want to either pick a debugger to use or work on a debugger should read this table. This will also allow a feature overview so we can view all the existing features a debugger has while seeing which debugger can complement a missing feature from another debugger.

Once all the analysis and experimentation is done the integration of a debugger for *Nau* begins.

The original debugger will have to be changed in ways that will allow *Nau* to interact with the debugger, this will force some tinkering with the debuggers code as a new adapted version of the original. *Nau* comes with a basic GUI, Composer, which uses *Nau* in order to render XML projects, this composer will receive additional features to accommodate the new debugging features, these additions shall be conscious of the debugging studies. These modifications will be detailed in chapter 3 and *Nau*'s new debugging features will described in chapter 4.

Conclusion and future work are detailed in chapter 5.

The installation method and usage procedures of all five open source debuggers these are documented in the appendices.

<div align="right">

2

</div>

STATE OF THE ART

In these study seven different debuggers for OpenGL were selected, five open source and two commercial debuggers. Four of the open source debuggers were chosen because they are mentioned in the official OpenGL wiki, aside from being the most popular among the community. VOGL is an exception, chosen due to its relation to Valve. The list of the selected debuggers is:

- Bugle (sec 2.1) - Open source

- Apitrace (sec 2.2) - Open source

- GLIntercept (sec 2.3) - Open source

- GLSLDevil (sec 2.4) - Open source

- VOGL (sec 2.5) - Open source

- CodeXL (sec 2.6) - Proprietary

- NSight (sec 2.7) - Proprietary

The focus of this work is on OpenGL debugging, thus it will barely mention other features outside of this context. Topics like DirectX, CUDA and OpenCL will not be dealt with unless it is also mentioned alongside OpenGL.

Most of these tools also provide some degree of profiling information, and this will be detailed for each debugger.

In the following sections each of the debuggers will be analysed. To conclude this chapter a comparison is performed in sec. 2.8.

2.1 BUGLE

Bugle [MB07b] is mostly used by Linux operative systems. For Windows it is recommended to use MinGW but a word of caution is given regarding Windows installation in the official website [MB07a] "it is significantly trickier than on UNIX-like systems, and currently only recommended for experts".

This work tested a successfully compiled Bugle in a Ubuntu 13.10 gnome operative system, this section shall report the analysis and experiments with Bugle.

Bugle's configuration is based on filters. A filter is a set of actions used to extract, manage, or print information. Filters are chained together, like a production line from a factory, the product being debug or profile information.

Bugle comes with several filters. Filters can be configured during the chain but no method other than changing the C code directly can create new filters.

There is also a GUI which allows for easier user experience. The GUI also enables the option of step by step to trace resources such as viewing textures, buffers or shaders.

The main list of features is as follows:

- Configure and show meta-data statistics (sec. A.2.1.1 and A.2.1.2);

- Log all GL calls (sec. A.2.1.3);

- Identify API errors (sec. A.2.1.4) ;

- Use OpenGL extension KHR_Debug (sec. A.2.1.5);

- Change the applications context attributes (sec. A.2.1.6);

- View OpenGL version and extensions used on the application (sec. A.2.1.7);

- Output a compilable C file (sec. A.2.1.8);

- Screenshot or video capture (sec. A.2.1.9 and A.2.1.10);

- Force front buffer output (sec. A.2.1.11);

- Switch to wireframe visualization (sec. A.2.1.12).

Bugle, as all the other open source debuggers, is also capable of logging OpenGL calls. This is extremely useful in complex applications such as 3D engines which are generic by design, and therefore have highly parametrized code paths.

The identification of incorrect parametrization of API calls is also a common feature. OpenGL parameters can be constants and it is common for a programmer to use the a value which is not on the list of accepted constants for a particular function. Furthermore, as in OpenGL all objects have a numeric ID, it is also easy to use invalid IDs. All these errors can be caught by these Bugle.

The usage of KHR_Debug is a welcome addition to Bugle. This extension replaces the old `glGetError` and provides far more information when an error occurs. However, Bugle's usage of KHR_Debug is limited to the display/storage of such information, not taking any advantage of the callback mechanism provided by the extension.

The usage of the above extension requires an OpenGL debug context, hence it is required to ensure that such a context is created. Bugle allows the override of the context attributes, therefore solving this issue.

Bugle has a unique feature, it is capable of creating a C compilable file with the code to recreate the sequence of OpenGL commands. This file can be used as a replay of the original application, much like a video, but with code, instead of images.

The ability to take screenshots is also a feature that can be found in other debuggers. This allows to compare images generated with different algorithms, thereby providing debugging information to the shader programmer. The possibility to also create videos is an added bonus.

Forcing front buffer output provides the possibility to take screenshots of partial results, i.e. before swap buffers is called. Note, however, that this feature does not work with render targets, only with the default frame buffer.

The wireframe visualization feature is useful to inspect the models triangulation.

2.1.1 *Filter and Statistics Configuration*

Bugle is a debugger which works based on chains of filters, each `chain` is a set of filters which are executed during the debugging of the application. The chains are customizable by the user to adapt into different debugging requirements.

The filters are methods to either extract or output information, for instance the following filter `filterset stats_basic` is a filter which extracts basic information from the application such as the number of frames and time elapsed. However, this `filterset` does not output any information, it is only an information extraction filter. To display the information gathered by `filterset showstats`, filter `stats_basic` is required.

Some filters themselves can be configured, for example `showstats` mentioned before can be configured to show different statistics.

Bugle stores the configuration of which statistics to gather is a separate file, allowing not only to specify atomic attributes such as the number of frames elapsed but also to build new statistics as functions of these atomic attributes.

For instance, a new attribute for frames per second can be defined as

```
"frames per second" = d("frames") / d("seconds")
{
    precision 1
    label "fps"
}
```

Bugle allows some basic arithmetic operations such as sum, subtraction, division and multiplication, to combine atomic attributes. The options inside the brackets are related to the display of the new

7

statistic attribute. This means that whenever `showstats` uses `show frames per second` it'll output `fps` as shown in figure 1.
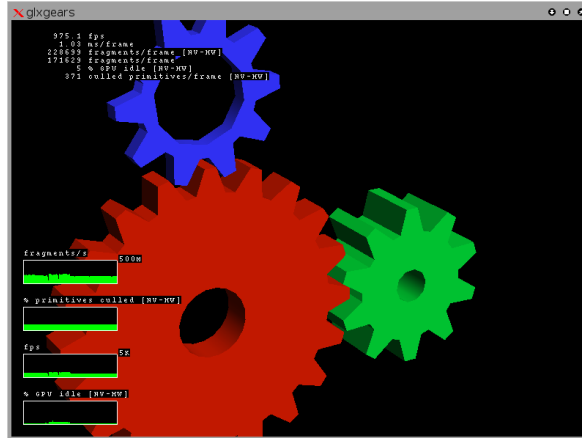
Figure 1.: Bugle showstats example.

An example of a chain with two filter sets is now provided:

```
chain override
{
    filterset extoverride
    {
        disable "GL_EXT_framebuffer_object"
        disable "GL_EXT_framebuffer_blit"
    }
    filterset contextattribs
    {
        major 3
        minor 3
        flag "debug"
        flag "robust"
        profile "core"
        profile "!compatibility" #disables compatibility
    }
}
```

As can be seen in the example above, the chain contains two filters. The first causes two extensions to be disabled, whereas the second alters the attributes of the OpenGL context, effectively overriding the application's own defined context.

2.1.2 *Graphic User Interface*

Bugle can be ran in a command line or with a GUI. The GUI provides additional debugging features besides *filter chains* such as:

- Viewing the GL state, it is possible to check which states changed by checking "show changed states", this is shown in figure 2;

- Step by step debugging, and breakpoints on certain GL function are also possible;

- Viewing resources such as shaders and texture;

- Viewing buffers and framebuffers. These resources are updated according to the current state bugle is paused (from step by step or breakpoints);

- Additional error logs, for instance there is the shader error log in the shader tab.

The GUI allows to pause the application and check its state. Breakpoints serve as an extension of step by step debugging and it helps searching or hunting for certain functions or parts of the rendering where an anomaly may occur.

The exposure of buffers and shader information to the user provides very valuable information to the user. Buffers can be the indirect output of a shader, and their inspection allows the programmer to check if the shader is output the correct data.

The shader log provides compilation and linkage information which details where things go wrong, much like the errors of a standard C compiler and linker.
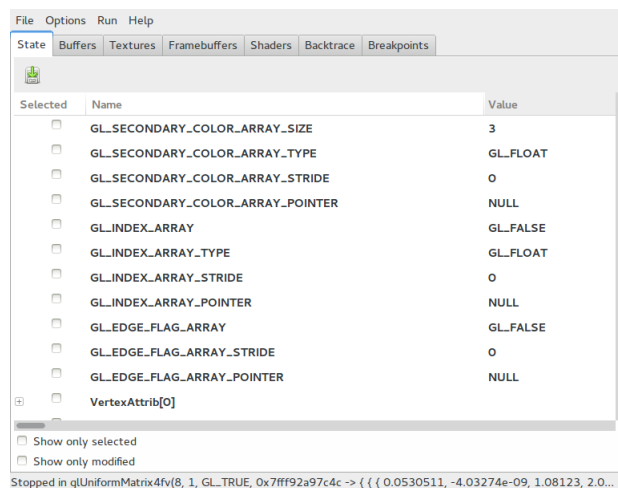
Figure 2.: Bugle state tab.

9
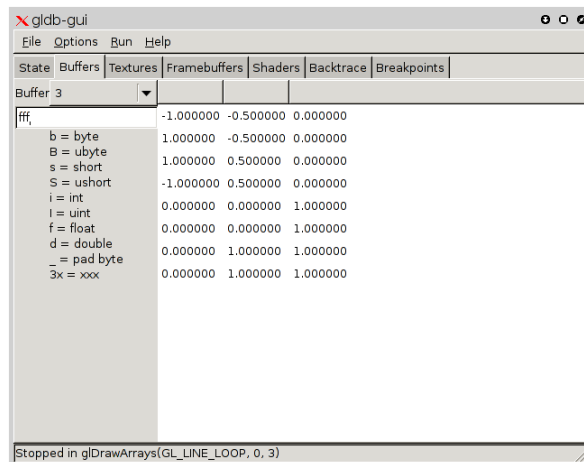
Figure 3.: Bugle buffer view.
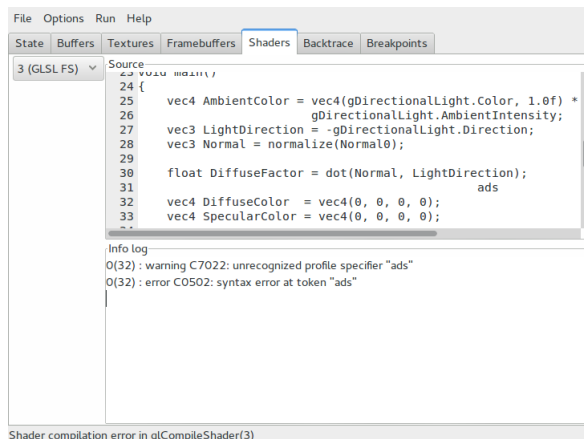
Figure 4.: Bugle shader error encountered.

2.1.3 *Inner Workings*

In Unix systems OpenGL debugging uses the wrapping method, the difference between Windows and Unix files is that Unix relies on *.so while Windows uses *.dll [Wik].

It is possible to link *.so files before loading applications. This is exactly what Bugle does. The gldb interface simplifies the process for the user as it will only add the `BUGLE_CHAIN=<chain>` `LD_PRELOAD=libbugle.so` before running the application. As can be seen `libbugle.so` is the wrapping library and `LD_PRELOAD` forces to use the specific library first. In windows systems it'll create a `opengl32.dll` instead.

This also means that GUI is optional, simply use `BUGLE_CHAIN=<chain>` `LD_PRELOAD=` `libbugle.so <target application>` and Bugle will initiate, the `<chain>` is the name of chain to be used (check *Filter Configuration* section).

2.1.4 *Maintenance*

In order to update Bugle with the latest OpenGL functions simply update the files that come within bugle's khronos-api folder. Bugle generates its files using the OpenGL libraries using the khronos-api XML files ([Khr14b]).

Once the new libraries are updated a rebuild is required. The building process is described in sec. A.1.

2.2 APITRACE

Apitrace [FJea13] has cross-platform compatibility allowing debugging for Linux, Windows, and Android.

The debugging has two stages. A logging stage where the application is started and Apitrace records every OpenGL command issued. The second stage provides the ability to provide the log, or trace, files.

This debugger has the ability to replay its trace files allowing the user to check and verify its current state including the resources and uniforms used for the current function. Using the replay it is possible to dump images to ffmpeg in order to create a video.

In short this debugger offers:

- Capability to create an OpenGL, OpenGL ES, Direct 3D and DirectDraw trace file;

- Replaying an OpenGL and OpenGL ES trace;

- Using the replay combined with ffmpeg to create a video;

- Inspection of the OpenGL state when retracing;

- Viewing and editing trace files;

- Creating screenshots and videos based on the replay.

Apitrace is a debugger that needs to be run in command line when tracing. It will create a `.trace` file which contains all traced information.

To read a trace file it relies in its GUI. Its trace file is a binary file readable only by Apitrace which may be a downside considering it forces the user to use only its tools.

Replaying is Apitrace's strongest feature. Replaying is independent from the target application, hence it is possible to store many different trace files for replaying based on the same application.

Trace files can be edited, allowing to change shader code, OpenGL function parameters, and add or remove functions.

An issue with trace files is their size, as these can get extremely large. Apitrace allows trimming but extreme care is required, for instance to avoid removing important initialization sections.

When using the GUI Apitrace behaves like the other debuggers, showing logs for function calls, allowing to examine the state(figure 5), textures, shader code, uniform variables.
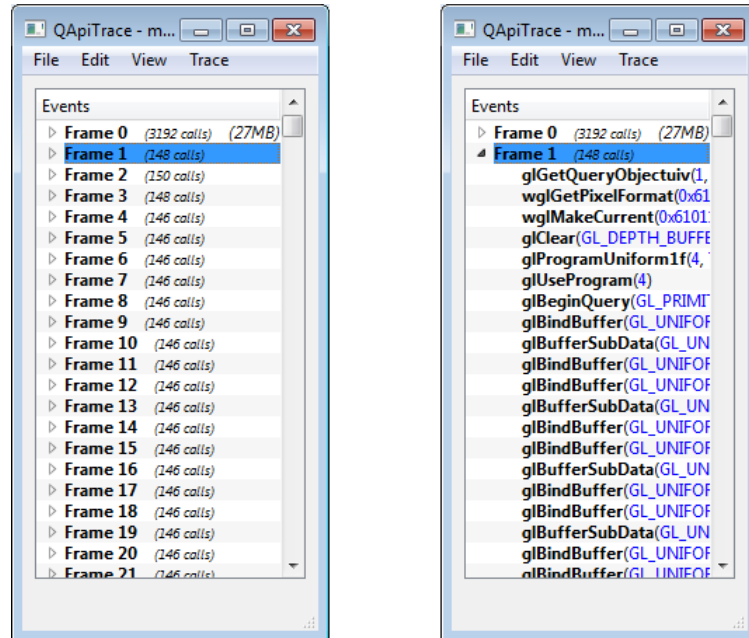
Figure 5.: On the left shows how Apitrace splits the log per each frame, on the right it has a frame node expanded showing its function log.

Not only it is possible to manually edit the GL calls for the next replay it is also possible to lookup the state on the current function shown in 6. This is very useful since it shows the GL state, uniforms and shaders. Doing so allows the user to find if everything is in place before an OpenGL call.
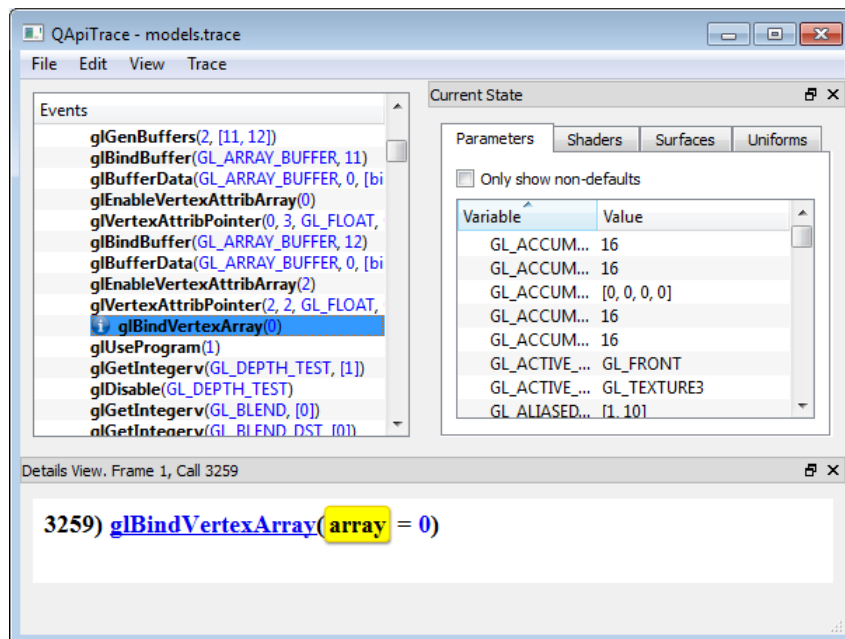
Figure 6.: Looking up the state before changing the program, it allows to see the current program's parameters, shaders, buffers(surfaces tab) and uniforms.

Another worthwhile feature of Apitrace is profiling. Apitrace can measure the CPU and GPU times for each OpenGL call, as well as report the number of pixels that were written in draw calls.

2.2.1 *Inner Workings*

O Apitrace for windows works using the usual DLLinjection method. Apitrace raw code is coded with a combination of C code and Python scripts, the python scripts are used to generate the final C code solution.

This application isn't based on one single executable, it has multiples executables and also carries python scripts to make additional and more advanced commands.

Making a qt and a script based initial solution allows less code to be written, it does create a more complex build method, however it also allows cross-platform compatibility.

A study of Apitrace's code concludes that the tracing works according to the following diagram in figure 7:
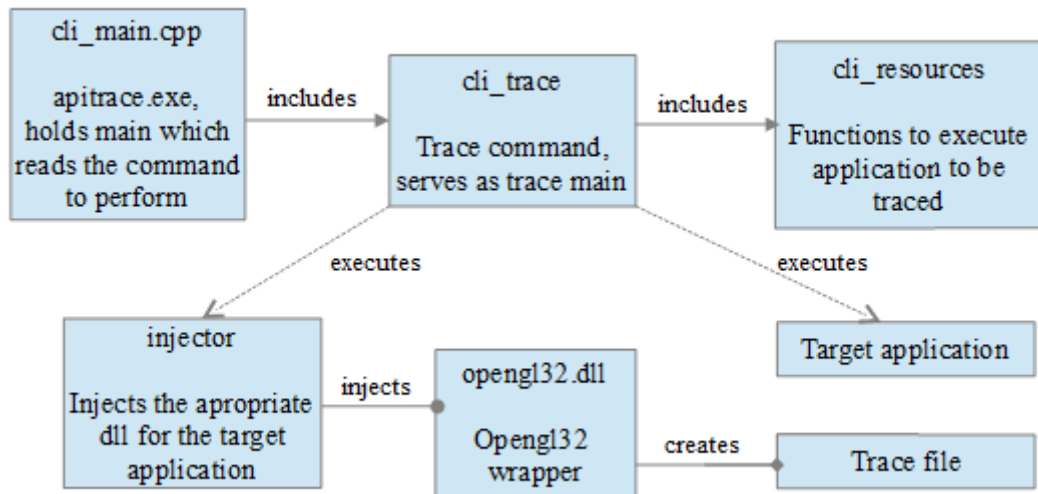
Figure 7.: Apitrace inner workings diagram - trace action

Apitrace works with the common DLL injection method, it uses an injection application to make the injection work, it does require an previously prepared wrapper, the wrapper itself does the logging. Apitrace also allows to trace direct3D as well since it contain other wrappers beside OpenGL.

2.2.2 *Maintenance*

The information regarding OpenGL functions and parameters is stored in python scripts. When a new version of OpenGL comes out, or even a new extension, these scripts must be updated.

These scripts are created by another script: `glspec.py`. This is done by using `make -b` command on the `specs\scripts` folder, this command will download all the current OpenGL specs, it'll also generate `glapi.py`, `glxapi.py`, `wglapi.py`, `glparams.py`, `wglenum.py` and `eglenum.py`.

However the process is not fully automatic, requiring manual editing. An outside user, i.e. outside of the development team of Apitrace may find it hard to do the update on his own. Nevertheless it should be feasible to add a few functions manually if necessary.

2.3 GLINTERCEPT

GLIntercept [Tre13] is an OpenGL debugger for windows, it has an official windows installer which contains a fully updated debugger up to OpenGL 4.5.

This is the only debugger that does not have a GUI. It relies on manually edited text files when it comes to configuration and utilization.

This debugger is unique due to the fact that it allows the inclusion of plugin libraries. The source code has its own plug-in solution to help creation of plugins. By using this capability it is possible for other programmers to extend the debugger, without having to alter the source code, nor recompile it.

The debugger itself is capable of tracing and XML logging, it can also capture frames and resources. If on one hand this is the debugger with less features, it is also the one with the nicest architecture regarding future expansions, thanks to the inclusion of plugins.

GLIntercept without any plugins only performs logging and some basic level of error checking. Plugins allow to add another wrap to OpenGL functions, allowing code to be executed before and after the functions.

The original GLIntercept also comes with additional plugins that allows:
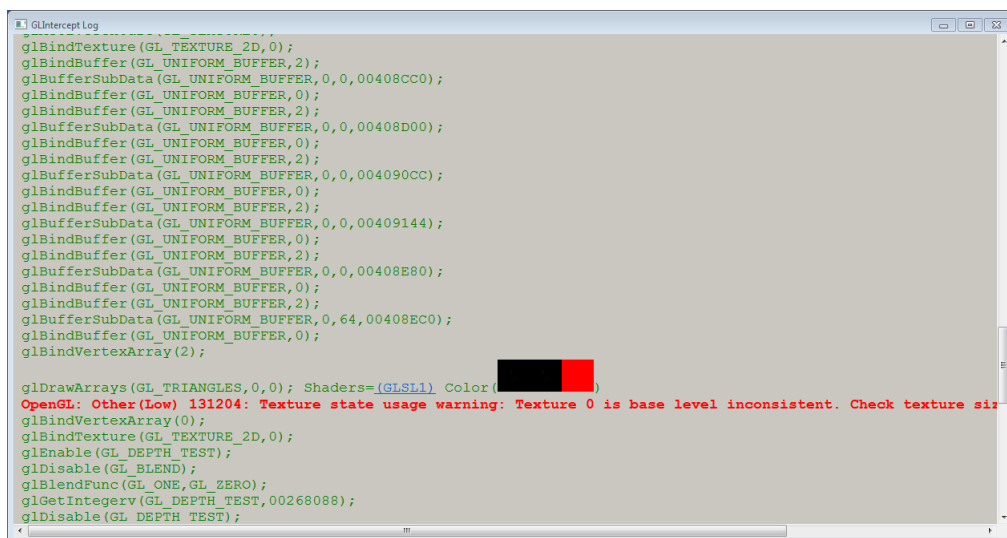
- ARB_ debug_ output Logging;

- Extension override;

- Shader Editor;

- Frame pinging (updating the frame);

- Write function statistics.

2.3.1 *Logging*

GLIntercept most basic feature is logging. As mentioned before this is what GLIntercept does without plugins. The basic log GLIntercept, saved in a file, looks like:

```
(...)
glUniform4fv(5,1,[0.597000,-0.390000,0.700000,0.000000])
glUniformMatrix4fv(6,1,false,[-0.006048,0.002009,0.003809,0.000000,
0.000000,0.007302,-0.002488,0.000000,0.005158,0.002355,0.004466,
0.000000,0.705268,0.841264,0.479696,1.000000])
glUniform1iv(7,1,[1])
glUniform1iv(8,1,[2])
glUniform1iv(9,1,[0])
glBindVertexArray(10)
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,13)
glDrawElements(GL_TRIANGLES,1518,GL_UNSIGNED_INT,00000000)
(...)
```

GLIntercept can log per frame. The nicest feature of GLIntercept logs is the ability to output content enhanced logs in XML format. An example of a XML log can be seen in figure 8.

Figure 8.: GLIntercept XML output.

As seen in figure 8 the XML log can display thumbnails of textures, and links to where shader source code is stored. Note that this is not a link to the original source code, but to the source code captured by GLIntercept.

At each render call, for instance `glDrawArrays` it can also display the currently bound shaders and textures.

Basic error checking can also be configured in GLIntercept. This mechanism is based on `glGetError` function, and it has the option of issuing a breakpoint when an error occurs, or just report the error.

Another useful feature of the log is the ability to save the contents of the frame buffer. When this option is enabled the user can configure to save the pre/post/diff frame buffer, including the color, depth and stencil buffers. As an added bonus users can create a color index table to help the visualization of the stencil buffer.

An interesting option is the "LogFlush". When an application crashes there is a risk that the log in the file is not fully saved. Enabling this option guarantees that every call is recorded.

GLIntercept does provide a rich list of logging features, which together with the plugin architecture makes it a very interesting option.

2.3.2 *Plugins*

As mentioned before, GLIntercept only performs logging by default. Any other addition to GLIntercept can be made through plugins. Plugins essentially add another wrapper within GLIntercept's wrapper. For instance, the statistics plugin works by counting every function that passes through the wrapper and outputting the results at the end of the application.

It is also possible to override the current OpenGL state within a plugin.

GLIntercept has plugins placed on a separate Visual Studio solution, the solution should have several projects, each project is a DLL plugin for GLIntercept.

When observing all solutions it can found that they usually have the following pattern:

- Header folder with the following files:

    - `CommonErrorLog.h` so the tools know how to log errors;

    - `InterceptPluginInterface.h` where the connection between the code and output DLL is made;

    - `<Plugin Name>.def` (Sometimes in source folder instead) always with the exact same functions, used to define the DLL;

    - (optional) `config.ini` usually an example of how to activate the plugin.

- Source common files with the following files:

    - `ConfigParser.<cpp/h>` to parse the configuration files;

    - (optional) `InputUtils.<cpp/h>` required to override the application's input;

    - (optional) `MiscUtils.<cpp/h>` has a few additional functions which may come in handy;

    - (optional) `NetworkUtils.<cpp/h>` when network communication is required;

    - (optional) `ReferenceCount.h` which provides reference counting class.

- And also in the source folder it can be found `<Plugin Name>.cpp` which acts as a main function and includes `<PluginCommon.cpp>` required for DLL exportation.

These functions do not require modification, they are shared between all plugins, the exceptions are the ones dependent on the plugin's name, the .def file has the exact same contents between all plugins.

It is possible to create additional plugins for GLIntercept and its source code provides the workspace to do so, use the `GLI Plugins` solution to view the plugins source code and it is possible to use the already existing plugins as reference.

The already existing test plugin is a perfect base for a new plugin, its important to check the other plugins to import additional headers and code such as `InputUtils` to capture input.

In order to understand plugin creation the `CustomGetUniforms` was created to intercept uniform data, one of the first tasks was to reuse one of the already existing projects and rename it to the new plugin's name, editing the project's properties is crucial.

Since it is intended to only print uniform information with a press of keys `InputUtils` is necessary.

The plugin starts with at the `CreateFunctionLogPlugin` which can be copied from an already existing plugin and simply re-adapted, this is followed by a new customized class based on

`InterceptPluginInterface` again re-adapted from already existing code, here this class is named as `GetUniforms`.

Where a real change is required starts from the class constructor, in `GetUniforms` it was necessary to add all functions to intercept with `gliCallBacks->RegisterGLFunction(<string>)` while using `"*"` as a string value would allow all functions unfortunately it does not allow partial wildcards such as `"glProgramUniform*"` thus requiring a function iterating through the different variations. The class deconstructor can be left empty.

In `GLFunctionPre` is where the `InterceptPluginInterface` intercepts all registered functions, using the function `accessArgs.Get(<pointer>)` it'll fetch and store the value in `<pointer>` which can be an integer or a pointer to an array, in this case it was necessary to check which type of function was intercepted in case the uniform value was float, integer , etc.

Once the value type of an argument is known it can be used as a simple integer or a pointer to an memory location, however if it points to a part of memory it will only become valid on `GLFunctionPost`, sometimes it is wise to use a pointer which is held from `GLFunctionPre` to `GLFunctionPost`.

The plugin `CustomGetUniforms` in order to print only at the end of the frame it is important to use `GLFrameEndPost`, here it will use `inputSys.IsAllKeyDown(<keys>)` to check if the required set of keys have been pressed, in such case it'll print the uniform data if valid. `<keys>` is the set of keys necessary to be pressed and it is a vector of unsigned integers (`vector<uint>`), it is possible to use the configuration to read the desired set of keys.

In `CustomGetUniforms` it is used `GetKeyCodes("CaptureKeys",parser,captureKeys)` to read the keys, `GetKeyCodes` is a function copied from the existing camera plugin, the parser is the `ConfigParser` and `captureKeys` the vector which is also used as `<keys>` mentioned before. This way it is possible to add additional parameters in the GLIntercept's config file to change the keys necessary to print the uniforms.

2.3.3 *Inner Workings*

GLIntercept in order to wrap has to record all functions during initialization, this is done by reading OpenGL's header files with its own parser and recording them in the `FunctionTable`.

When a plugin asks for a specific function to wrap it'll also add a function to the `FunctionTable`, the `FunctionTable` serves as a handler and intercepts functions.

`BuiltinFunction.h` is the header of all basic OpenGL 1.* functions which are mandatory for the wrapper.

GLIntercept also needs to know where the original `opengl32.dll` is, when the function is called GLIntercept will always dynamically redirect OpenGL functions to the main DLL allowing functions to be called even when the header is unknown.

2.3.4 *Maintenance*

GLIntercept can log all OpenGL functions. However, there is no automatic way of understanding the function parameters unless some configuration is provided. GLIntercept requires header configuration files with the function signatures and constants to provide this information. In case the function information is not provided it'll replace the parameters information with `"???"`. Currently, GLIntercept is up to date with OpenGL 4.5.

Should a new OpenGL version come out, it is feasible to partially update the header configuration files manually by adding a few functions and constants.

For a full update, GLIntercept has a script to use the egl, gl, glx and wgl XML files from khronos spec repository at https://cvs.khronos.org/svn/repos/ogl/trunk/doc/registry/public/api/. It is possible to use the script to create an almost complete GLIntercept header configuration files. A small amount of code tuning is still required due to the fact that OpenGL has constants defined with the same value.

First place the headers in the `3rdParty\HeaderGen` folder and execute `XMLGenGLI.py` script. This should create the headers automatically, however, tuning is still required.

In an attempt to transform GLEW's header into an appropriate GLIntercept's header several steps of careful tuning were required.

There are a few pointers that should be noted when making the changes:

- Some types have to be replaced with similar types for example `GLclampf` becomes `GLfloat`;

- Many defines must be enclosed within an enum, for example `GL_ZERO` has to be enclosed within `enum EnumName{ GL_ZERO 0x0,};`;

- `GLEnum` types will need to point at the enum which should be enclosed with (if it requires `GLZERO` using the previous example it will become `GLEnum[EnumName];`

- `GLbitfield` behaves like `GLEnum`, however conversion to `GLEnum` is unnecessary, it does uses the defined enums;

- Some enums like `GL_ZERO` and `GL_FALSE` need to be enclosed in different sets of enums, using the previous example it'd have `enum EnumNameBools{ GL_FALSE 0x0, GL_TRUE 0x1};`, this way there won't be conflict between `GL_ZERO` and `GL_FALSE`, however in order to recognize as `GL_FALSE` it'll need to be changed to `GLEnum[EnumNameBools];`

- `APIENTRY` and `GLAPI` will need to be removed from the header file;

- It is best to split the header files according to OpenGL version, this way it becomes easier to track the version changes while relying on the already existing headers.

For instance in `glext.h` it has:

```
#define GL_PIXEL_MODE_BIT 0x0020
//...
#define GL_COMPUTE_SHADER_BIT 0x0020
```

And in the header configuration files it has

```
//gli1_1 include file
enum Mask_Attributes {
    ...
  GL_PIXEL_MODE_BIT                  = 0x0020,
  ...
};
//gli4_3 include file
enum Mask_ShaderProgramStages {

  GL_COMPUTE_SHADER_BIT              = 0x0020,

};
```

The conversion of functions also requires tuning to take the different `enums` into account. For instance, consider the function `glUseProgramStages`:

```
//glext.h
GLAPI void APIENTRY glUseProgramStages (GLuint pipeline,
    GLbitfield stages, GLuint program);

//gli4_4 include file
void glUseProgramStages(GLuint pipeline,
    GLbitfield[Mask_ShaderProgramStages] stages,
    GLuint program);
```

On the positive side, these updates do not require a rebuild of GLIntercept.

2.4   GLSLDEVIL/GLSL-DEBUGGER

glslDevil [KS10] originally by Magnus Strengert, Thomas Klein, and Thomas Ertl [SKE07] is now renamed as GLSL-Debugger, [HX13] it is a debugger with a GUI similar to commercial debuggers. This debugger is still alive and being updated however it still requires improvement.

From the current analysis this debugger allows some control on the debugging such as freezing the debugged application to view its statistics and resuming until it hits a certain function.

As practical its interface may be the shader debugger is not reliable except for showing which shader is active, it may end with much less features compared to the other debuggers. Its uses are the following:

- GL calls trace;

- Step-by-step OpenGL debugging;

- Buffer visualization;

- Shader debugging (if the current function is an OpenGL *debuggable draw call*;

- Restart shader debugging without changing the windows items;

- Step-by-step shader debugging;

- List shader variables with name, type and current value;

- Different windows according to the appropriate shader.

In the Windows version of GLSL-Debugger a `glsldebug.dll` wrapper is created

Whenever GLSL-Debugger debugs an application it'll create a log file as well, this GLSL log file format is according to the following example:

```
1395652361 - INFO: ORIG_GL: glGetActiveAttrib (0x7fded7f79600)
1395652361 - INFO: ORIG_GL: glGetError (0x7fded7f76b40)
1395652361 - INFO: ORIG_GL: glGetAttribLocation (0x7fded7f79620)
1395652361 - INFO: ORIG_GL: glGetError (0x7fded7f76b40)
1395652361 - INFO: SAVED ATTRIB: Position size=1 type=GL_FLOAT_VEC3,
location=0
1395652361 - INFO: ORIG_GL: glGetActiveAttrib (0x7fded7f79600)
1395652361 - INFO: ORIG_GL: glGetError (0x7fded7f76b40)
1395652361 - INFO: ORIG_GL: glGetAttribLocation (0x7fded7f79620)
1395652361 - INFO: ORIG_GL: glGetError (0x7fded7f76b40)
1395652361 - INFO: SAVED ATTRIB: TexCoord size=1 type=GL_FLOAT_VEC2,
location=1
1395652361 - INFO: ORIG_GL: glGetProgramiv (0x7fded7f794c0)
1395652361 - INFO: ORIG_GL: glGetProgramiv (0x7fded7f794c0)
1395652361 - INFO: ORIG_GL: glGetProgramiv (0x7fded7f794c0)
1395652361 - INFO: ORIG_GL: glGetError (0x7fded7f76b40)
...
```

GLSL does not have the best automatic log system compared to the other debuggers, but the GUI is the main focus of the debugger.

2.4.1 *Graphic User Interface*

GLSL-Debugger's GUI is capable of showing log in real-time unlike most debuggers. As show in figure 9 it shows both trace and statistics, these are updated as long the application runs, this is useful as it allows the user to see which functions are being called, the rate they are being called and the amount of functions so far.
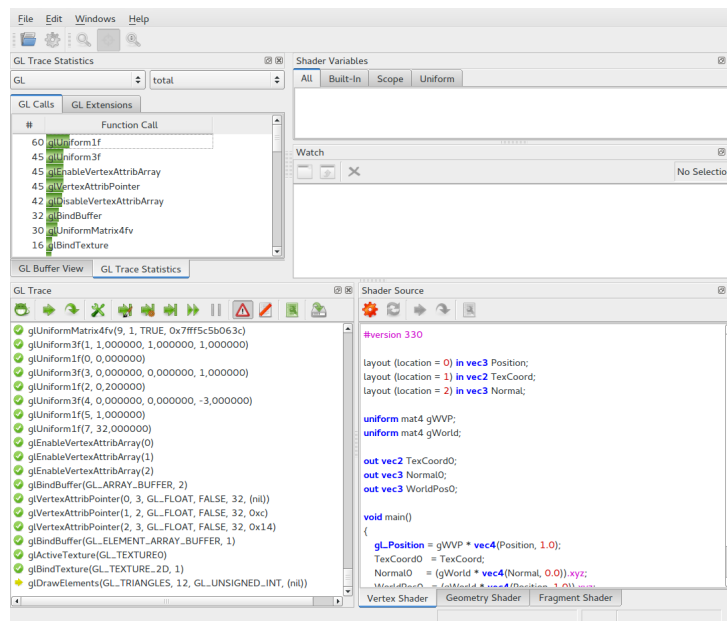
Figure 9.: GLSL-Debugger featuring trace statistics and the vertex shader.

The GUI allows to simulate the trace of shader code whenever a draw call is the next function to be called, in order to do so the application must be paused. Currently only GL 2.0 shaders can be reliably debugged while mesa limits the version to GL 3.0. This debugging is shown in figure 10 where the frag color is being debugged, the shader variables shift according to the debugger. Shader variables are only visible when debugging.
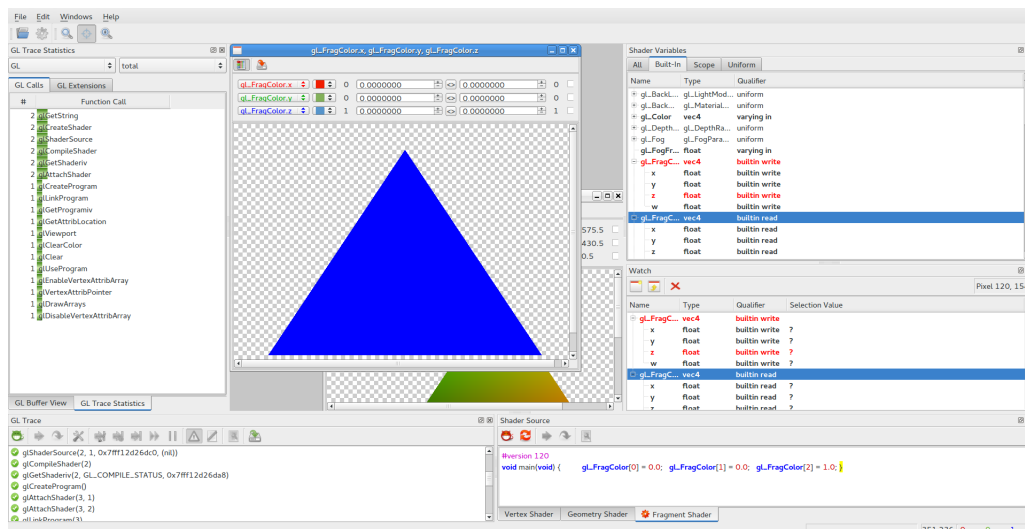
Figure 10.: GLSL fragment color viewer.

GLSL Shader debug code is in the debug lib files, in order to debug the shader it'll parse all shader data such as uniforms and attributes into a `ShaderProgram` struct and run them in a new thread. It fetches uniform data using OpenGL functions such as `glGetIntegerv` or `glGetProgramiv` from the original OpenGL functions.

Unfortunately the whole process still needs further development since shader debugging is unreliably and limited to mesa headers (GL version 3.3 as of November 8th, 2014).

2.4.2 *Maintenance*

GLSL updates its library functions by relying on khrono's OpenGL headers, so in order to update its library it needs to have the current headers on `glsldb/GL` replaced and remake the build. For instance it'll need to use `glext.h`[1] to substitute the current GLSL glext file, other files from the same repository are `wglext.h` and `glxext.h. WinGDI.h`[2]. Finally `gl.h` and `glx.h` belongs to mesa repository [3]. Even so there is a good chance that updating will cause issues with enumerates.

2.5 VOGL

VOGL [Val] is an open source OpenGL debugger released this year, originally authored by RAD Game Tools and Valve Corporation. This debugger should have both Linux and Windows cross-platform compatibility. Just like Apitrace cmake and Qt is required.

The main goal of this debugger is efficient replay. Valve's team realized that going through a replay of Apitrace implied the sequential execution of the code up to the desired break point. The team wanted to avoid this as performing sequential access in a game can take a long time. Valve's team are designing a solution to be able to randomly access a given frame. This implies that the state bookkeeping will be far more complex, yet for large applications this may be a very useful feature.

As a Valve debugger this debugger can capture steam games. Note that currently this debugger is only in alpha stages, those interested should pay attention to its development and growth. According to wikipedia [Com] this debugger has the following high level goals:

- Free and open-source

- Steam integration

- Vendor and driver version neutral

- No special app builds needed

- Frame capturing, full stream tracing, trace trimming

---

1 http://www.khronos.org/registry/

2 http://www.csee.umbc.edu/ squire/download/WinGDI.h

3 https://github.com/freedreno/mesa/tree/master/include/GL

- Optimized replayer

- OpenGL usage validation

- Regression testing, benchmarking

- Robust API support: OpenGL v3/4.x, core or compatibility contexts

- UI to edit captures, inspect state, diff snapshots, control tracing

2.5.1 *Functions and GUI*

Just like the other wrapper based solutions it is necessary to copy the DLL wrapper to the target application. Once copied it can be debugged via command line using a GUI. VOGL can view its log's with its own GUI which is organized very similarly to Apitrace. it is also capable of looking up the state with replaying just like Apitrace, this can be seen in figure 11.
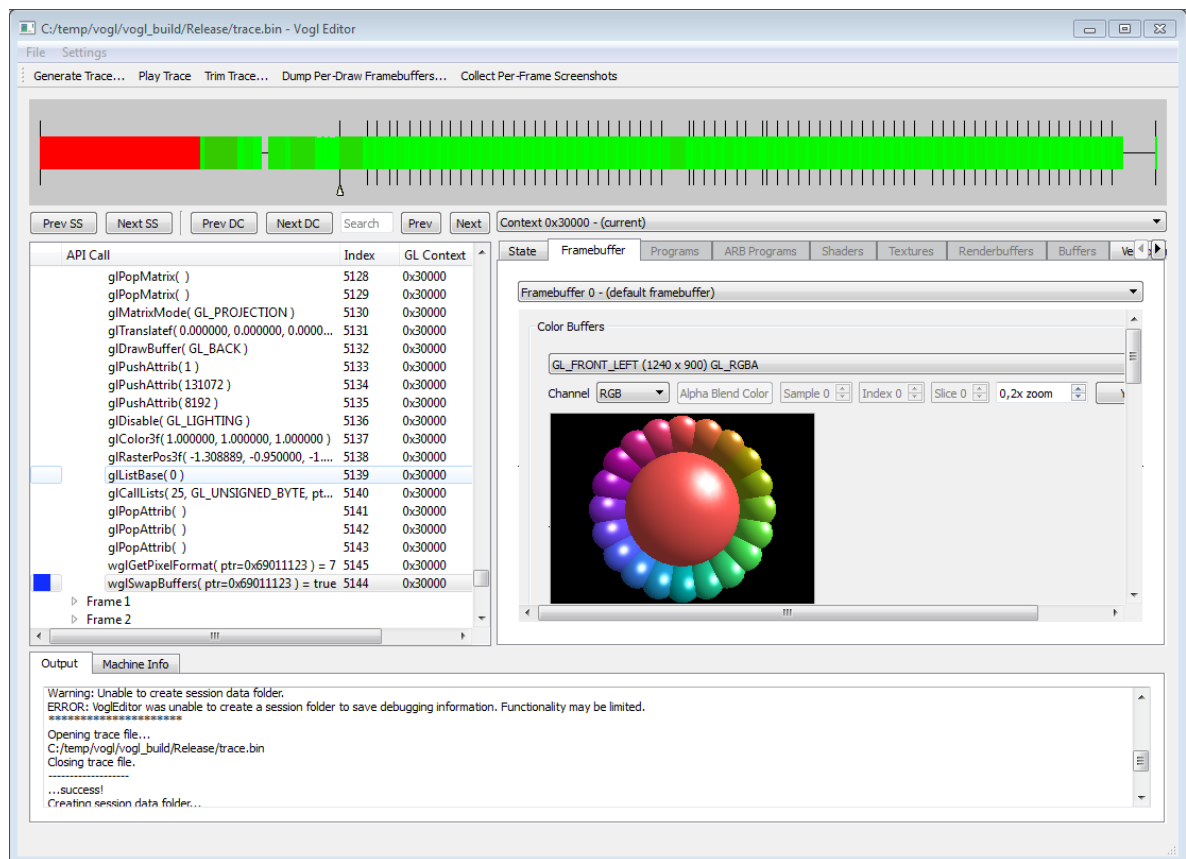
Figure 11.: VOGL editor after snapshot.

VOGL store trace files in `.bin` format, it is also possible to convert them to `.json` format, in `.json` format the frames are separated per `.json` file. When replaying a `.json` trace it is possible

to edit the file during replay, the next time the frame is replayed the changes should be in effect unless it is cached (which requires restarting the replay).

The replay can actually start with interactive mode which allows the user to pause the replay midway. What it actually does is to stop at a certain frame. For example in order to jump the frame backwards it'll replay from the beginning until it reaches the previous frame.

VOGL is capable of benchmarking a trace file, by doing so it'll attempt to run the trace as fast as it can while offering the benchmark results at the end of the replay.

The GUI should show a timeline in the top area, that's the functions timeline, it points at the current selected function within the timeline.

2.5.2 *Maintenance*

VOGL like other debuggers rely on khronos XML spec files, those spec files are stored within `glspec` folder, during compilation of the VOGL's Visual Studio solution It'll automatically build and use `voglgen32.exe` which in turn reads the XML spec files to create `.inc` files.

This means VOGL can be maintained by simply updating with the latest spec files as the rest of the process is automatic.

2.6 CODEXL

CodeXL [AMD13] is an AMD developed debugger, it carries capabilities specifically for AMD boards, it allows the user to check both the GPU and CPU states. CodeXL can also work as a Visual Studio extension, thus debugger offers services such as:

- Trace all API calls;

- Performance counters, includes kernel usage, number of instructions executed, etc.;

- Temporal kernel observation;

- Application summary;

- Kernel code disassembly;

- Shader Debugger;

- Remote application debugging.

CodeXL has its own IDE which may be used instead of Visual Studio, however it uses a different solution structure.

2.6.1 *Debug Mode*

Debug mode gives basic buffer and resources information, it works when a project pauses. When viewing the buffers it shows what is drawn on the buffer, same for loaded textures. VBO's have more data associated from "Image view" and "Data view", the data view showing the arrays in a table with its own display options.

2.6.2 *Profilling Mode*

There are different types of profiling, in order to get a report it is necessary to start and finish the application, afterwards a report file will be generated.

CPU time based profiling will report the functions performance, such as the number of times called and the percentage of all calls they represent.

In Visual Studio it will show the 5 hotspot functions and modules so allowing an user to check where there may be a bottleneck.

GPU application trace will provide the common trace call list, it will also give the statistic about the time spent on each function such as the maximum, minimum and average times.

The trace report will also include a timeline related to the application lifespan.

2.7 NSIGHT

Nsight [NVIa] is a debugger developed by NVIDIA, just like CodeXL it also extends Visual Studio but can also extend Eclipse. Unlike CodeXL it doesn't carry its own IDE. This document will focus on the Visual Studio extension.

Following features from NSight are expected:

- Application and System Trace with Call Stack Correlation;

- Trace and report SLI performance limiters;

- Graphics Shader Debugger;

- Graphics Frame Debugger;

- Frame Profiling;

- Frame Timing.

- Remote application debugging.

During usage some of the features don't work in the laptop, in order to use NSight to its fullest it should be run in a desktop. In fact, the most interesting option, shader tracing, does not work with Optimus equipped laptops.

2.7.1 *Graphics Debugging*

Nsight's graphics debugging mode is far more unique than any other debuggers, rather than giving a trace file or a report it overrides the application's output. Essentially it means that part of its GUI runs on the debugged application itself, and can also be used as standalone apart from the IDE.

It is easy to notice a performance graph on the left (on default) as shown in figure 12, to open the HUD show in figure 13 and 14 use `<ctrl-z>`. The HUD also reports graphical memory usage.
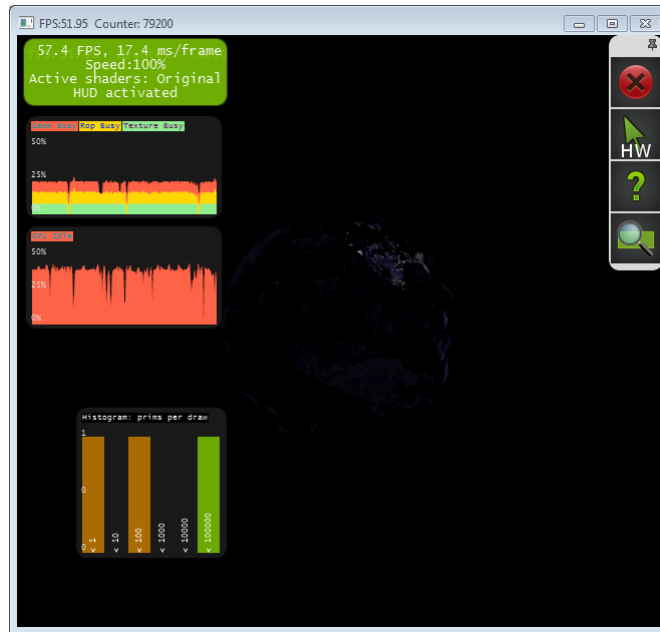
Figure 12.: Nsight HUD while application is running.

When the HUD is on a number of new options are available such as `<ctrl-w>` which forces wireframe to be active as shown in figure 14. When paused even more options are available as shown in figure 13, if an appropriate IDE is used it will synchronize and integrate some options on the IDE.
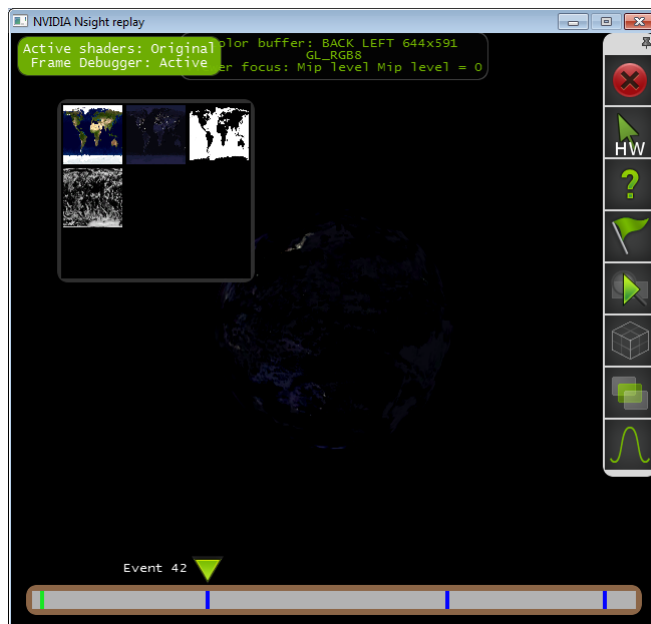
Figure 13.: Nsight HUD while application is paused.
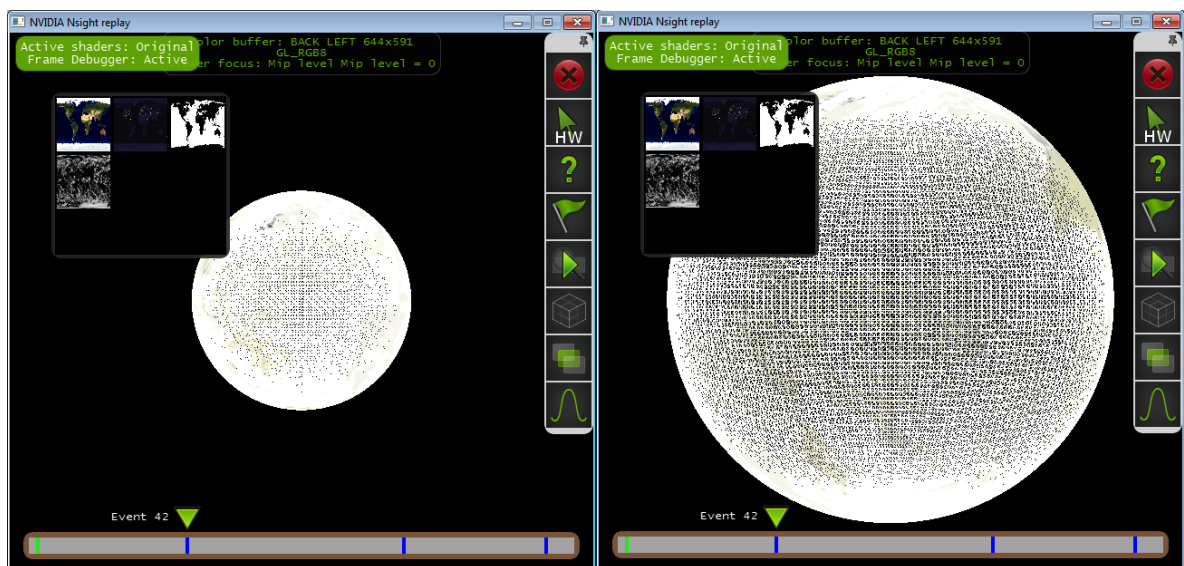
Figure 14.: Nsight HUD while application is paused with wireframe on.

The timeline shown on the bottom at the figures 13 and 14 is a timeline representing the draw calls on the current frame, there the state on each rendering stage can be viewed. This allows to understand better the sequence of commands for the current frame. The same timeline is synchronized with the IDE which allows to view the resources related to the rendering stage on an automatically opened tab called "API Inspector".

The "API Inspector" allows to view the used resources. An important feature is the program section in the menu on the left, allowing to view which working shaders are related to the draw call. Using the source link on the shader list will open a fetched shader file. Here break points can be defined as in a common Visual Studio debugging session. Shader variables are also accessible.

2.7.2 *Performance Analysis*

When using "Performance Analysis" it will display a wide variety of options to select. "Activity Type" will show a list of four items, unless it is required to trace other related processes "Trace Application" option will do.

"Trace Settings" starts unchecked, the program used had no useful information regarding CUDA, "DirectX, OpenCL and Tools Extensions, selecting those will not affect the debug result list (except for performance perhaps). Only System and OpenGL were need, Tools Extensions was checked just in case.

Afterwards, the Launch button causes the start of the analysis. It will generate a report when stopped (either Stop button or the application ends, "Trace Process Tree" will only create the report when the Stop button is clicked). The report will be split in several sections, the name of the section should provide enough understanding to its contents. The sections are the following:

- Common
    - Summary Report
    - Session Summary
    - Timeline

- OpenGL
    - OpenGL API Call Summary
    - OpenGL API Calls
    - OpenGL Draw Calls
    - OpenGL Frames
    - OpenGL Transfers

- System
    - Function Calls
    - GPU Devices
    - Modules
    - System Information

Among the five open source debuggers GLIntercept is the most simplest one. Its basic functions are limited to logging, although it is the most complete log of all. GL Intercept does not have any GUI. On the plus side, GLIntercept has the great advantage of allowing plugins to extend the wrapper. Plugins allow other programmers to extend GLIntercept without messing with its code, something that would be required when a new version of GLIntercept is released. This architecture, which is both simple and powerful makes it an ideal candidate to merge with a 3D project requiring constant debugging, such as a 3D engine.

Apitrace and VOGL are very similar to each other. Apitrace is older than VOGL, and the latter has an improved interface. Having a team supported by Valve is also a plus point for VOGL. However, at the present time, unlike VOGL, Apitrace is up to date with the latest OpenGL specifications. These two debuggers are the only ones showing the ability to replay. Even the two proprietary debuggers do not have this feature. Bugle has its own alpha version of replay with the compilable C file feature, but even when mature it will not match Apitrace and VOGL as these later debuggers have a GUI that assists in trimming and editing the trace files.

GLSL-Debugger and VOGL have the most complete interfaces, with GLSL being the only open source debugger, capable of simulating shader debugging. However, this feature is far beyond the latest OpenGL specifications meaning that is of little use except for some basic learning.

Bugle is very rich compared to the other debuggers, it has more features, shows more information and it is possible to configure it to tailor to the user needs. Among the five debuggers Bugle is currently the most complete debugger in many aspects. It does however, lack the ability to replay trace files like VOGL and Apitrace, it can't add additional plugins like GLIntercept and it doesn't have a shader debug simulator like GLSL-Debugger. Bugle ability to show statistics on runtime far surpasses the other open source debuggers.

CodeXL and NSight are actually very similar to each other, they both show very detailed profiling results, are capable of debugging shaders on their respective manufacturer's hardware and show detailed statistics and logs.

The proprietary debugger's major flaw is the fact that they are hardware limited including the fact that very old GPUs are incompatible with some features, and laptops equipped with Optimus are not allowed to do shader debugging with NSight.

There are a few differences between them. As shown NSight places a HUD on the application something that CodeXL doesn't have, this means that the usability and experience in using them should be different.

The major advantage for proprietary debuggers is shader debugging, particularly for NSight. However, this is limited hardware wise, and its not up to date (OpenGL 4.2 for NSight 4.2, and OpenGL 4.3 for CodeXL 1.5). Still, if the hardware matches the requirements, and no cutting edge feature is being used, this is a major plus.

The open source debuggers have features that are unique as well. The plugins from GLIntercept and replay ability from Apitrace and VOGL are two good examples.

A comparison between the open source debuggers highlighting the pros and cons of each tool is presented in table 1. A feature list for open source debuggers is presented in table 2. Table 3 presents similar information for CodeXL and NSight.

	Bugle	Apitrace	GLIntercept	GLSL-Debugger	VOGL
	Graphic User Interface;	Generation of organized trace files;	Easy to set up;	Graphic User Interface;	Graphic User Interface;
	Chain information filters;	Trace organized per frame;	it is unnecessary to install additional tools to build;	Easier for an inexperienced user;	Generation of organized trace files;
	Configure meta-data statistics;	Editing the trace file;	Existing configurations are easy to set up;	Viewing the shader source when debugging;	Trace organized per frame;
	Realtime statistics, including OpenGL call count, fps, etc;	Profilling a trace file;	Plugins may increase the potential for interested users;		Exporting frames to editable json;
		Trimming a trace file;		View shader variables contents (incomplete);	Trimming a trace file;
		Replaying the trace;	Viewing/Editing shader when debugging (can fail);	View resources when debugging;	Replaying the trace as long the trace file exists;
Pros	View Shader errors;	Replaying the trace until a chosen frame;	Overriding context attributes;	View OpenGL call count on realtime per frame;	Replaying the trace until a chosen frame;

	Overriding context attributes; Screenshot / Video capturing; Create eps vector graphics screenshots; Step by Step debugging; Automatic Maintenance	Creating videos with the trace file; View loaded uniforms when looking up the state with GUI; Cross-Platform compatibility.	Creation of XML files as a trace file; Configuration inside .ini files in a understandable C based code; Updating to a newer OpenGL requires only an additional include list; View total number of OpenGL calls at the end of trace.	View extension count on real-time per frame; Step by Step debugging Edit parameters on the next OpenGL function to call; Breakpoint at particular functions	Dumping buffer state and screenshots; Editing trace files while replaying; Conversion from Apitrace to VOGL; Cross-Platform compatibility Automatic Maintenance
Cons	Hard to install on windows;	Binary trace files	No GUI; No Linux version.	Dependency on Mesa library	Still in early alpha stages.

Table 1.: Open Source Applications Pros and Cons table

In table 2 the following tags are used::

- *yes* - means that the feature was found in the debugger;

- *no* - means that the feature was not found in the debugger;

- *bugged* - means a bug occurred during the using of the following feature;

- *incomplete* - means that the feature is not always compatible but the author is aware of it;

- *similar* - means that the exact feature is not present, but a similar feature can be used to replace it.

- *Win* - is used to abbreviate Windows.

- *Unix* - means Unix like systems, such as Linux.

	Apitrace	Bugle	GLIntercept	GLSL-Debugger	VOGL
GUI	partial	yes	no	yes	yes
Trace step-by-step	no	yes	no	yes	no
Replay trace	yes	bugged[1]	no	no	yes
ARB_debug_output	yes	yes	yes	no	yes
Trace statistics	no	yes	yes	yes	yes
Runtime statistics	no	yes	no	yes	no
Video capturing	yes	yes	yes	no	no
Screenshooting	yes	yes	yes	similar	yes
Capture frame log	yes	similar	yes	similar	yes
Application profiling	yes	yes	no	no	yes
Shader information	yes	yes	yes	yes	untested
Uniform data reading	yes	yes	no	incomplete	untested
Force Wireframe	no	yes	yes	no	no
Plugin addition	no	no	yes	no	no
OS	Win/Unix	Win/Unix?	Windows	Win?/Unix	Win/Unix

Table 2.: Open Source Applications Feature table

Note (1): Bugle does not have true replay. The compilable C code can be considered as in the same category

A list of pros and cons for NSight and CodeXL is presented in table 3.

	CodeXL (partially tested)	Nsight
Pros	Visual Studio extension; Has its own IDE; Performance Analysis (CPU Profiling); View application resources; Step by step shader debugging (untested); Gives a function usage summary; Can create several report files; It can be downloaded for free without registration.	Visual Studio and Eclipse extension; Nvidia HUD is available outside the environment; Performance Analysis (CPU Profiling); View application resources per render stage; Step by step shader debugging using Visual Studio's break points (untested); Gives a function usage summary; HUD Integrated into the application with real time performance graph; HUD's timeline is synchronized with IDE; Can create several report files; Can list the modules/files used by the application.
Cons	Buggy if it isn't an AMD GPU; May cause errors on a Nvidia GPU.	Nvidia only; Lack of standalone IDE forces user to use one of the IDEs it extends; Requires Nvidia developer registration to use it (May take a day for approval); Significant number of features not available on Optimus based laptops.

Table 3.: Proprietary/Freeware Applications Pros and Cons table

Take notice that some features were written according to the documentation because the used hardware was not within the debugger's requirements.

Part II

DEBUGGING IN NAU

ADDING DEBUGGING FEATURES

Nau is an open-source 3D graphics engine that works with OpenGL as its base graphics API. Nau allows for multipass pipeline definition using XML project files. The material management system is very extensible and flexible allowing for complex pipelines, and enabling it to perform many graphical effects without the need to write a single line of code in the engine itself.

Nau works with Optix [NVIb], from NVIDIA, enabling hybrid rendering algorithms in pipelines that contain both rasterization and ray-tracing passes.

An embedded profiler detailing both CPU and GPU times provides helpful information to fine tune projects.

Composer is an application GUI that works on top of Nau's library and provides a simple GUI to explore the settings of the current project, allowing for shader recompilation in real time. It also provides information on materials, lights, cameras, and uniform variables.

Nau is continuously being updated to include new OpenGL features and to extend the XML project definition language. Although Nau already provides some debug information, both final users and developers would benefit from having extra debugging information available.

This chapter covers two approaches. First an open source debugger was integrated with Nau (sec. 3.1). This allows to use a set of features such as trace logging with minimal effort. Secondly, other debugging features were implemented in Nau itself (sec 3.2).

3.1 INTEGRATING A DEBUGGER WITH NAU

In order to have many debugger features without having to write the whole code it was decided to incorporate one of the studied open source debuggers. The selected debugger was GLIntercept because it provides the most hard missing functionality, function wrapping, and it does not have a GUI, which is not required. For all other debuggers, it would be necessary to separate the GUI from the logging functionality. All other functionality present in the other debuggers, such as buffer and texture inspection can be implemented more easily directly in Nau/Composer.

Another benefit of using GLIntercept is the ability to add plugins. If in the future some missing feature is required it is easier to write a plugin than to change the debugger code itself.

3.1.1 *Changes on GLIntercept*

Several changes were made on GLIntercept in order to integrate the library, first it was necessary to create a header (`ConfigDataExport.h`) which allows to edit the configuration settings outside of the original GLIntercept's functionalities, this results in a new header file that exports configuration editing functions.

Because the settings can now be edited on runtime several changes regarding removing and adding new settings also had to be made, for example plugins alter an internal function table, which implies methods to clean the function table in order to avoid crashing.

This is was done because the class `ExtensionFunction` indexes the functions to execute during wrapping in an array, if it is not properly cleaned it will suddenly execute an already erased function. The following code is an addition to clean this table:

```
//Plugins often add Overrides which should be removed if
//the plugin is removed
void ExtensionFunction::removeOverrides(){
    int removedExtensionsCount = 0;
    for (int i=overrideIndexList.size()-1;i>=0; i--){
        //Uses the list containing the plugin added overrides;
        //these must be deleted
        if (overrideMustDeleteList[i]){
            removedExtensionsCount++;
            //Removes override from function table
            functionTable->RemoveFunction(overrideIndexList[i]);
            //Removes the override from the real list
            //reorganizes the list (innefficient) but
            //it is fine since this function isn't used often
            for (int j=overrideCurrIndexList[i];
                 j<currExtensionIndex-1; j++){
                wrapperIndex[j]=wrapperIndex[j+1]-1;
                extFunctions[j]=extFunctions[j+1];
            }
        }
        else{
            //Fixes the pointer for the overrided function
            //if not deleted
            const FunctionData *foundFunc =
                functionTable->GetFunctionData(overrideIndexList[i]);
            *foundFunc->internalFunctionDataPtr =
```

```
                (void**)overrideCurrIndexList[i];
        }
    }
    currExtensionIndex -= removedExtensionsCount;
    overrideIndexList.clear();
    overrideCurrIndexList.clear();
    overrideMustDeleteList.clear();
}
```

A new function to restore the original loaded settings was also implemented, this seemed to be important in case the user wanted to reload or reset the configuration, considering how the configuration was stored all it required was a copy of the original loaded configuration stored in the driver and a current configuration.

The gliLog is no longer automatically created, it has to be started manually with a function on the target application. This allows Nau to not use GLIntercept at all, so the new features are not forced on the user and haze zero performance impact when disabled.

Also it was decided to disable any functionality to the wrapped functions until it is enabled by the application, this is to prevent any function to be logged when the application is supposedly paused, this is done by forcing the functions `LogFunctionPre` and `LogFunctionPost` to do nothing until active. This forces the necessity to use `gliSetIsGLIActive(true)` when releasing the paused state and `gliSetIsGLIActive(false)` when returning to the paused state. The debuggers `AddLoggerString` was exported as `gliInsertLogMessage` in order to allow Nau to add personal messages to the log file.

This is important since it allows Nau to define blocks of code, for instance for each frame, pass and others that may turn out to be helpful in the debugging task.

These additions means that the GLIntercept solution will need to be compiled as well, for more information regarding GLIntercept's installation check the appendices.

3.1.2 *Changes on Nau*

Nau will now interact with GLIntercept. A `gliConfig.ini` is necessary to load standard default configurations.

The `<debug>` tag now exists and is required in order to use the new features else it won't use the debugger at all. This means previous works won't use debugger, so the new features won't harm its performance.

On the code's side the new tags are all recorded on `projectloaderdebuglinker.cpp`. If a programmer wishes to add or remove tags for the debugger just edit this file. How tags work shall be mentioned on the usage section.

It is also highly recommended to install GLIntercept in order to have a fully organized directory with the basic GLIntercept necessities. The config file should be edited to fit the install directory.

3.1.3 *Changes on composer*

Nau comes with a solution called composer which reads Nau projects, a few debugging changes have been made in order to add debugging options.

The composer now is capable of pausing rendering, this causes the rendering result's to pause (it won't show the results at all which may look glitched), once it is pause it will attempt to load the txt logfile and split the log in frames, it'll also load program information which includes shaders and uniform.

Splitting the GLIntercept's simply requires to use `wglSwapBuffers` as a separator, this conclusion was achieved by analysing Apitrace's logs where all frames ended with the exact same function. Using the same method as GLIntercept's function statistics plugin a statistic regarding the function usage is generated.

Since GLIntercept supports single frame logging, the feature uses multiple folders which requires an additional library. The chosen library was *dirent*[Rö] due to its easier installation, practical use and compatibility. This does mean its include folders require an additional `dirent.h`, since dirent has all its functions embedded within the header file no new dll is required aside GLIntercept's wrapper.

3.2 ADDING OTHER DEBUGGING FEATURES

Besides incorporating an open source debugger into Nau, other debugging features were implemented. These features are useful for both Nau developers as well as the user who loads an XML file.

The list of these features is as follows:

- Buffer inspection;

- OpenGL State inspection;

- Shader uniforms inspection;

When paused, Composer will gather buffer data, by using OpenGL methods. It will first attempt to get all current VAO data and map the buffer information into a c++ map structure (which is actually a tree map). Once the VAO buffers are mapped it will map the remaining buffers, for instance shader storage buffers.

State inspection is a feature that is common to many open source debuggers. However, this brings an issue related to OpenGL evolution. Each new version may bring new enumerations to already existing state query functions, or even new functions. This implies that any hard coded solution will imply altering the source code in such scenario.

None of the studied debuggers provides any way of displaying new OpenGL state that doesn't imply adding/rewriting code, and rebuild.

Furthermore, OpenGL state is considerably large, with many items that may not be relevant for a particular application.

In here a solution is proposed that currently only goes halfway to the desired goal: fully configurable, no code editing state management. The solution is based on XML configuration files. These XML files detail the enumerands and the OpenGL functions that use them.

When Nau starts this file is loaded. When the application pauses, Nau will use the loaded information to call the appropriate functions and display the results in a window.

This solution deals only with part of the issue. First, if new enumerands are defined in new OpenGL versions, these can added to the XML file and no code editing is required, nor a rebuild. Secondly, the state to be displayed can be selected in the XML file. There is no need to display all state in every application.

The part that is yet to be solved is related to new OpenGL state query functions. Since there is no simple way to implement reflection in C++ this feature is not yet available. A scripting language can be an alternative solution for this problem, but the integration of scripting in Nau is future work.

A common error when developing new features in Nau is the mismatch between the uniforms we want to sent to the shader, and the uniforms we actually send to the shader. Nau provided only information about the former, and if no bugs are present in the source code, this feature is sufficient. When developing new features it is essential to know if these values match.

Information about the uniforms actually set in the shaders can be retrieved using OpenGL query functions. The information is extracted using methods from Lighthouse 3D's `vsGLInfoLib` [Ramb]. Nau is now capable of displaying both the information we want to send and the information actually sent.

A GUIDE TO NAU'S DEBUGGER

To activate Nau's debugger all is needed is the addition of the debugger tag on the project XML file, this section will demonstrate how most of these tags work. Often the mentioned value for the attribute will be some generic type, this means that the value should be of the same type within the quotation mark. The generic values are:

- `"bool"` so the value should either be `"true"` or `"false"`;

- `"uint"` means a a positive integer string like `"0"`, `"24"` and `"2000"`. `"-1"` is an example of invalid "uint";

- "string" is for text values so almost any value under quotes should be valid.

The project must have the debug tag like the following code, the `glilog` attribute is optional, when it is `"false"` it won't create the `glilog.txt` file:

```
<project>
    ...
    <debug glilog="bool">
        <functionlog>
            ... see functionlog section
        </functionlog>
        <errorchecking>
            ... see errorchecking section
        </errorchecking>
        <imagelog>
            ... see imagelog section
        </imagelog>
        <framelog>
            ... see framelog section
        </framelog>
        <timerlog>
            ... see timerlog section
```

```
        </timerlog>
        <plugins>
            <plugin>
                ... see plugins section
            </plugin>
            ...
        </plugins>
    </assets>
</project>
```

Most of the available options are very similar to the standard GLIntercept's config file, with a few exceptions.

4.1 FUNCTIONLOG

This will affect the created log file. When present a txt log will be created. While most tags are self explanatory it may be important to know that:

- When `<logmaxframeloggingenabled>` is enabled it will only log a certain number of frames according to `<logmaxnumlogframes>`;

- If `<logpath>` and `<logname>` are not set then the log file will be named by the default `gliConfig.ini` on the application folder.

```
        <functionlog>
            <enabled value="bool"/>
            <logmaxframeloggingenabled value="bool"/>
            <logmaxnumlogframes value="uint"/>
            <logpath value="string"/>
            <logname value="string"/>
        </functionlog>
```

4.2 LOGPERFRAME

This allows the user to create separate single frame log files, these log files are currently not connected to composers logfile.

```
        <logperframe>
            <logoneframeonly value="bool"/>
            <logframekeys>
```

```
            <item value="key"/>
            <item value="key"/>
            ...
        </logframekeys>
    </logperframe>
```

It should be noted that key means the key combination necessary to trigger the frame log snapshot, for example `<item value="ctrl"/><item value="f"/>` means that the user needs to press `<ctrl-f>` to create the snapshot.

4.3 ERRORCHECKING

Error checking allows the user to toggle whether to print errors on the `glilog` or not, when `"true"` the respective error will be logged.

```
<errorchecking>
    <errorgetopenglchecks value="bool"/>
    <errorthreadchecking value="bool"/>
    <errorbreakonerror value="bool"/>
    <errorlogonerror value="bool"/>
    <errorextendedlogerror value="bool"/>
    <errordebuggererrorlog value="bool"/>
</errorchecking>
```

The option `errorgetopenglchecks` causes a call to `glGetError` after each OpenGL call. Option is only useful for multi threading applications. This will report an error if a call is made on a thread that does not own the active render context. The remaining options will only have an effect if any of the two first options are active. Their behaviour is as follows:

- `errorbreakonerror`: issue a programmer debug breakpoint on an error. This allows to check the call stack on IDEs such as Visual Studio. Note the GLIntercept calls will also be present on the call stack, though;

- `errorlogonerror`: log errors on the function log;

- `errorextendedlogerror`: Resolve all parameters that were passed to the function to provide a full report on each function.

- `errordebuggererrorlog`: Mirrors the contents of the error log in Visual Studio debugger window.

4.4 IMAGELOG

Image log determines if the logger will log images, the images should be logged on a separate sub-folder from the log. `<imagesavepng>`, `<imagesavetga>`, `<imagesavejpg>`, are the allowed save formats and `<imageicon>` determines the icon type.

```
<imagelog>
    <imagerendercallstatelog value="bool">
    <imagesavepng value="bool"/>
    <imagesavetga value="bool"/>
    <imagesavejpg value="bool"/>
    <imageflipxaxis value="bool"/>
    <imagecubemaptile value="bool"/>
    <imagesave1d value="bool"/>
    <imagesave2d value="bool"/>
    <imagesave3d value="bool"/>
    <imagesavecube value="bool"/>
    <imageicon>
        <imageiconsize value="uint"/>
        <imageiconformat value="png"/>
    </imageicon>
</imagelog>
```

The options are as follows:

- `imagerendercallstatelog`: if enable it will list the set of active textures for each render call;

- `imagesave...`: The allowed image formats. Only one format can be specified;

- `imageflipxaxis`: This option flips the texture vertically.

- `imagecubemaptile`: By default cube maps are saved as six individual images. This option creates a single image with six tiles.

- `imagesave...`: Determines which type of OpenGL textures are saved.

- `imageicon`: When present an icon will be saved.

- `imageiconsize`: The icon size in pixels

- `imageiconformat`: The format to save the icon

The icon tags are useful when combined with XML logging, as these are placed whenever a texture is referenced in a function.

4.5 FRAMELOG

This will save the frame buffer's pre/post/diff state on an additional `Frame` folder. While in GLIntercept's `gliConfig.ini` the pre/post/diff flags are one single attribute, here the they are 3 separate booleans (for example `ColorBufferLog` is now: `<frameprecolorsave>`, `<framepostcolorsave>`, `<framediffcolorsave>`).

```
<framelog>
    <frameimageformat value="string"/>
    <framestencilcolors>
        <item value="uint"/>
        <item value="uint"/>
        ...
    </frameStencilColors>
    <frameprecolorsave value="bool"/>
    <framepostcolorsave value="bool"/>
    <framediffcolorsave value="bool"/>
    <framepredepthsave value="bool"/>
    <framepostdepthsave value="bool"/>
    <framediffdepthsave value="bool"/>
    <frameprestencilsave value="bool"/>
    <framepoststencilsave value="bool"/>
    <framediffstencilsave value="bool"/>
    <frameicon>
        <frameiconsave value="bool"/>
        <frameiconsize value="uint"/>
        <frameiconimageformat value="png"/>
    </frameicon>
    <framemovie>
        <framemovieenabled value="bool"/>
        <framemoviewidth value="uint"/>
        <framemovieheight value="uint"/>
        <framemovierate value="uint"/>
        <frameMovieCodecs>
            <item value="string"/>
            <item value="string"/>
            ...
        </frameMovieCodecs>
    </framemovie>
```

```
            </framelog>
```

When present GLIntercept will save the framebuffer contents in the specified format (`frameimageformat` can be TGA, PNG or JPG).

The icon is useful for XML logging.

For each buffer (color, depth and stencil) there is the possibility of saving the pre/post/diff contents, i.e. the value of the buffer before, after, and the difference between both. Regarding the diff option, the pixels that differ between pre and post will be coloured green.

An interesting option is `framestencilcolors`. This allows the creation of a color table for the stencil buffer for easier visualization of its contents. An example to color pixels with stencil value of 0 with red, 1 with blue, and 2 with yellow could be done as follows:

```
<framestencilcolors>
        <item value="0"/>
        <item value="0xFF0000FF"/>
        <item value="1"/>
        <item value="0xFFFF0000"/>
        <item value="2"/>
        <item value="0xFF00FFFF"/>
    </frameStencilColors>
```

Note that the color values are coded in ABGR, not in RGBA.

GLIntercept also provides the option to create a movie. If multiple buffers are selected they are tiled.

4.6 TIMERLOG

Used to add a time on main log's function when a function takes more time than the set cutoff. The cutoff value is in microseconds.

```
        <timerlog>
            <enabled value="bool"/>
            <timerlogcutoff value="uint"/>
        </timerlog>
```

Note that this is only CPU time, hence its usefulness is limited.

4.7 PLUGINS

Adding a plugin is slightly different on Nau because of the configuration format. Some plugins can fit extra parameters (for example extension override). These extra parameters should be placed as

mentioned in the example, the format for these parameters is the same as the GLIntercept's config file.

Pay in mind that most GLIntercept plugins require the plugin name to match a certain name, for example in the GLIntercept's config file there is the following plugin:

```
OpenGLShaderEdit = ("GLShaderEdit/GLShaderEdit.dll")
```

It is possible to see which names are required in the standard `gliConfig.ini`, by converting this for Nau projects, it results in:

```
<plugin name="OpenGLShaderEdit" dll="GLShaderEdit/GLShaderEdit.dll"/>
```

```
            <plugins>
                <plugin name="name string" dll="dllpath string">
                    extraparameter1 = "extraparameter1 value";
                    extraparameter2 = "extraparameter2 value";
                    ...
                <plugin>
                ...
            </plugins>
```

4.8 RETRIEVING OPENGL STATE

Nau has a new function state implementation which allows the user to get OpenGL state information by reading a XML file, the following example can be used:

```
<?xml version="1.0" ?>
<methods>
<method>
        <enums>
            <enum value="0x0BE2" name="GL_BLEND"/>
            <enum value="0x8076" name="GL_COLOR_ARRAY"/>
        </enums>
        <function name="glGetBooleanv">
            <param type="GLenum"/>
            <param type="GLboolean*"/>
        </function>
    </method>
    <method>
        <enums>
            <enum value="0x8455" name="GL_FOG_COORD_ARRAY_STRIDE"
                alias="GL_FOG_COORDINATE_ARRAY_STRIDE"/>
```

```
            <enum value="0x0B70" name="GL_DEPTH_RANGE" length="2"/>
        </enums>
        <function name="glGetIntegerv">
            <param type="GLenum"/>
            <param type="GLint*"/>
        </function>
    </method>
</methods>
```

This XML file will list OpenGL state gathering methods, each method corresponds to one function, this does not mean that a error will occur if the same function is duplicated in a different method list. The most important part of the method is the enums list (and their name and values) and the function name.

Using the example it is possible to say that the enums `GL_BLEND` and `GL_COLOR_ARRAY` use the function `glGetBooleanv`, the values are necessary to get the real OpenGL enum. In the second method the enums will use `glGetIntegerv`, the alias is unused but it may prove useful for later additions, length means that the enum fetches two values instead of 1 (the default length), using a length higher than 1 returns a array instead.

Adding new functions requires to add the respective function to the OpenGL function names map within Nau's `head/src/nau/debug/state.cpp`.

As mentioned before, this is not the ideal solution since it requires code editing to add new functions.

4.9   USING COMPOSER

The composer now has an additional `Debug` menu, in order to enable the other options it is necessary to use `Pause`, once paused the composer will start reading the GLIntercept main log file (if there is any) and fetch program data. This may take a second or more depending on the amount of data.

Once paused the `Advanced Pass Controller`, `GLI Log`, `Program Info` and `Buffer Info` shall become available. The `Next Pass` allows the composer to render only next pass, for more extensive pass control the user should use `Advanced Pass Controller` shown in figure 15. The `GLI Log` is shown in the figure 16, this is partially inspired by Apitrace's `qapitrace` GUI, however Nau's log has special Nau messages indicating whether Nau's frame or pass started or ended.
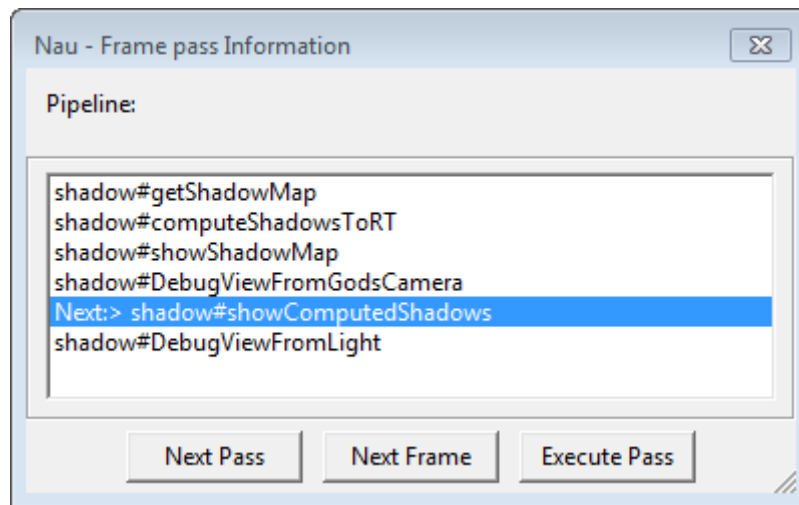
Figure 15.: Composer's Pass controller.
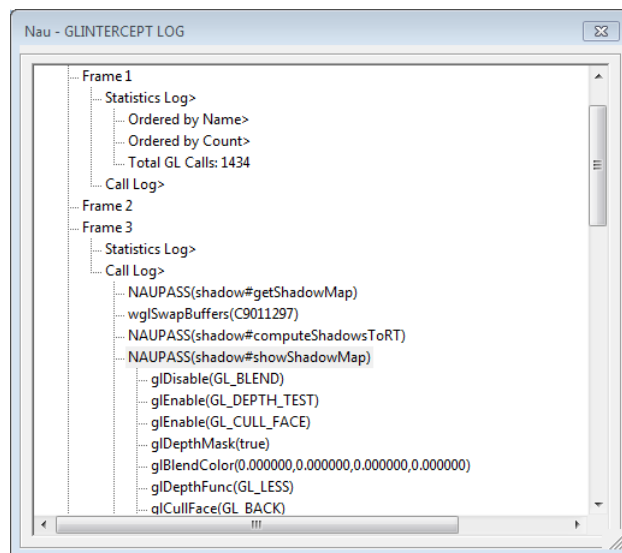
Figure 16.: Nau's GLIntercept log viewer.

`Program Info` in the figure 17 will give information from each program, uniforms are part of the `Program Info` but unfortunately only the latest uniform data will be recorded since uniform content isn't yet being logged. Take note that when a line ends with > as seen in the figure below it means it has a subnode.
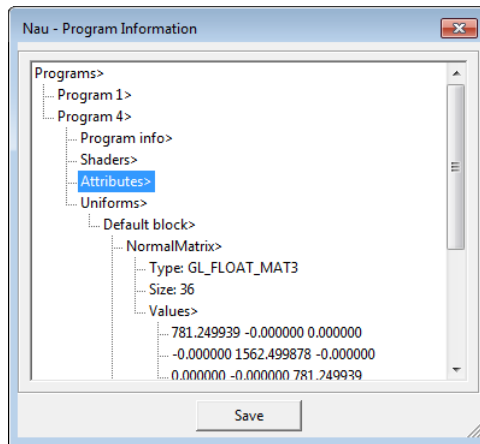
Figure 17.: Nau program information.

`Buffer Info` shows buffer information, the buffers are separated between VAO buffers and other buffers, VAO buffers have more detailed information and the types cannot be altered, the other buffers can change the type information as shown in figure 18 with a 6 types per line buffer.

Because some buffers are large (as shown in the figure 18, even with 6 cells per line it has over three thousand lines) they had to be paged for both visual and usability reasons since reading a buffer that big would take too long to load. The interface also has an option to save buffers to file.

In the figure 19 the VAO information can be seen, this information shows the VAO's element array and the corresponding buffer indexes which are in turn in the buffers page shown in figure 18;

Figure 18.: Nau buffer information.

Figure 19.: Nau VAO information.

<div align="right">

5

</div>

CONCLUSIONS AND FUTURE WORK

5.1 CONCLUSIONS

Open source debuggers are excellent tools that rival on some aspects with the proprietary debuggers from the graphics hardware manufacturers. This is true due to the fact that OpenGL is a very rich API, with a complete set of state querying functions.

Logging is performed thanks to wrapping of the original OpenGL library, thereby allowing code to be executed before and after each call. Most of the other features are based on OpenGL state query functions.

Apart from GLSL-debugger, all the open source debuggers are not capable of shader tracing. Even GLSL-Debugger is severely out of date due to its dependency on the Mesa library.

This limits proper shader debugging to the respective manufacturer's debugging tools such as CodeXL and Nsight, but event these tools have their limitations regarding the required hardware.

All these tools perform some sort of profiling, with the bare minimum being function call statistics. On the other hand Apitrace can measure GPU time for every function. Nevertheless, in this chapter the best tools are CodeXL and NSight, although Apitrace and VOGL are not too far apart.

It is interesting to note that at least Apitrace and GLIntercept are up to date with the latest OpenGL versions, while both CodeXL and NSight are still at least 2 minor versions behind.
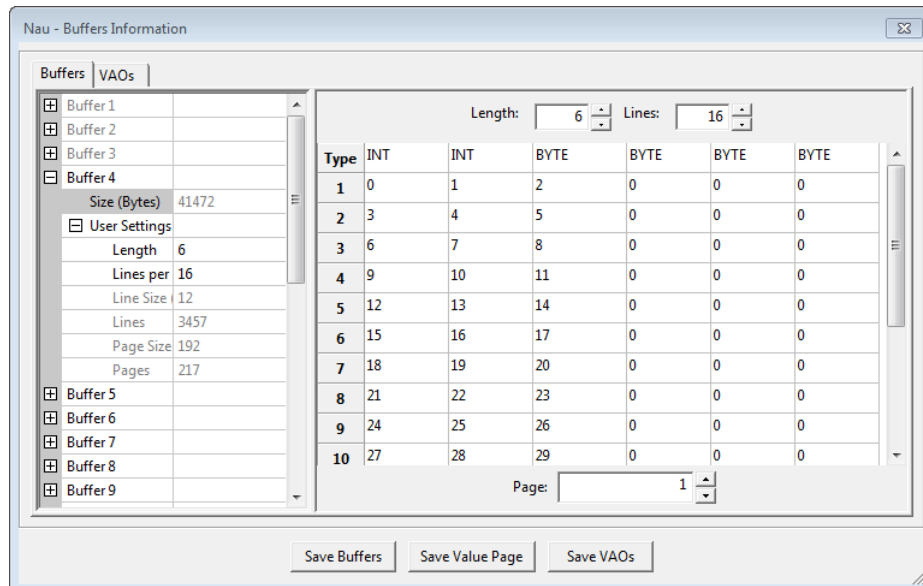
Integrating GLIntercept on Nau was a much more simple task than expected, GLIntercept's code itself is simple to read once the programmer knows where to start looking, all that was necessary to do was to export the configuration functions and to clean memory leaks. Then again the ease was all thanks to all the previous experience from reading all the necessary code.

On Nau's side, due to the highly modular architecture of the 3D engine, it was painless to introduce all the new features. Although a substantial set of features have been implemented, much remains to be done to achieve a really helpful debugging environment.

The new additional features are actually very useful, it allows a user to integrate additional GLIntercept plugins increasing the range of possibilities for *Nau* itself, the logs should also help the project writer to check for mistakes. This is also true for the engine developers. Buffer and state inspection is also crucial for debugging, particularly for shaders that write to, or read from, buffers.

This project may have some good impact on *Nau*'s future development and community.

5.2 PROSPECT FOR FUTURE WORK

There is still a long road ahead for both getting the ideal debugger, or the ideal debugger features implemented in an application.

The conclusion of the state management process in Nau could include a scripting language. The scripting language would also allow the expandability of Nau without altering a single line of code in the engine. This is definitely an avenue to pursue.

BIBLIOGRAPHY

[AAWL14] António Ramires Fernandes Andre Alexandre Wang Liu. Open source debuggers and integration with a 3d engine. In Nuno Rodrigues Alexandrino Gonçalves, António Ramires Fernandes, editor, *Atas do 21º Encontro Português de Computação Gráfica*, pages 23–30, November 2014.

[AMD13] AMD. Codexl. CodeXL. AMD, November 2013. [Online; accessed 14 Noevember 2013].

[Com] Wikipedia Comunity. Vogl. Wikipedia. [Online; accessed 24 October 2014].

[FJea13] Fonseca, Alexander Monakov José, and et al. Apitrace. GitHub, November 2013. [Online; accessed 1 Noevember 2013].

[HX13] Hanson and Chris 'Xenon'. Glsl-debugger. Github, October 2013. [Online; accessed 14 Noevember 2013].

[Khr14a] Khronos. Khr_debug extension spec, March 2014.

[Khr14b] Khronos. Khronos opengl specs, March 2014.

[KS10] Thomas Klein and Magnus Strengert. Glsldevil - opengl glsl debugger. GlslDevil. Http://www.vis.uni-stuttgart.de, February 2010. [Online; accessed 15 Noevember 2013].

[MB07a] Merry and Bruce. Bugle user manual. BuGLe. OpenGL, 2007. [Online; accessed 1 November 2013].

[MB07b] Merry and Bruce. Opengl software development kit. BuGLe. OpenGL, 2007. [Online; accessed 1 November 2013].

[NVIa] NVIDIA. Nvidia nsight visual studio edition. NVIDIA Nsight Visual Studio Edition. NVIDIA, n.d. [Online; accessed 15 Noevember 2013].

[NVIb] NVIDIA. Nvidia optix ray tracing engine,. [Online - accessed 1 August 2014].

[Ope12] OpenGL.org. Debugging tools. OpenGL.org, March 2012. [Online; accessed 1 Noevember 2013].

[Rö] Toni Rönkkö. Dirent api for microsoft visual studio. Softagalleria. [Online; accessed 24 August 2014].

[Rama]   António Ramires. Nau - opengl + optix 3d engine. GitHub. [Online; accessed 8 June 2014].

[Ramb]   António Ramires. Vsglinfolib – very simple opengl information lib. Lighthouse3D. [Online; accessed 7 July 2014].

[SKE07]   Magnus Strengert, Thomas Klein, and Thomas Ertl. A hardware-aware debugger for the opengl shading language. In *Graphics Hardware*, pages 81–88, 2007.

[Tre13]   Damian Trebilco.   Glintercept - opengl call interceptor/logger.   Glintercept. Code.google.com, June 2013. [Online; accessed 1 Noevember 2013].

[Val]   Valve. vogl. Github. [Online; accessed 24 October 2014].

[Wik]   Wikipedia. Dll injection. DLL injection - Wikipedia the free encyclopedia. [Online; accessed 7 February 2014].

INDEX

Part III

APPENDICES

A

BUGLE

According to the documentation in order to use Bugle in Ubuntu it requires to check the following requirements:

- GCC 4.1 or higher;

- A C++ compiler with the POSIX regular expression functions (generally g++).

- OpenGL header files, including a recent version of glext.h. A version supporting at least OpenGL 2.0 is required. For OpenGL ES, the OpenGL ES and EGL header files are required;

- Perl 5;

- Python 2.7;

- Scons;

There are also secondary recommended requirements:

- GTK+ is needed for the GUI debugger.

- gtkglext, GLEW (1.6 or later) are needed for the texture and framebuffer viewers in the GUI debugger.

- GLUT is required by the test suite.

- FFmpeg allows the included libavcodec to be used for video capture.

Recent installations of Ubuntu usually have GCC, g++, Perl and Python. OpenGL libraries are still required to be installed and even if they aren't installed you should install it for the debugged programs, Scons also needs to be installed to build bugle:

```
sudo apt-get install freeglut3-dev libglew-dev libpangox-1.0-dev
sudo apt-get install  Scons
```

For the secondary installations a different approach is used, the latest version of GTK+ is hard to install, in fact a I failed to do so, GTK+ 2.24 works fine and is an already existing Ubuntu 13.10 package, in order to install it needs to:

```
sudo apt-get install libglib2.0-dev
```

The line above only if you don't have glib installed yet:

```
sudo apt-get install libatk-dev libatk-bridge2.0-dev libgtk2.0-dev
libpango-dev libpango1.0-dev libcairo-dev libgdk-pixbuf2.0-dev
```

After GTK+ is installed you can install gtkglext, the procedure is the same, download the tar file, extract and install with the same commands used to install GTK+. FFmpeg can be installed with:

```
sudo apt-get install ffmpeg
```

In case you are not satisfied with the current version of FFmpeg you consider changing the repository and installing a more recent version with the following commands:

```
sudo apt-get remove ffmpeg
sudo apt-get purge libav-tools
sudo add-apt-repository ppa:jon-severinsson/ffmpeg
sudo apt-get update
sudo apt-get dist-upgrade
sudo apt-get install ffmpeg
sudo apt-get install frei0r-plugins
sudo apt-get --purge autoremove
```

All it needs now is to install bugle, download the archive from the official site and extract it, I shall refer the folder you where the extracted contents are as `<root bugle directory>`, once extracted all you need to do is to build it with scons, this is done by doing:

```
cd <root bugle directory>
scons
sudo scons install
```

A build folder will be generated as a result of the first scons, the second will install it within the system.

Afterward you'll need to create a `.bugle` folder in you home directory where you'll place a `filters` and a `statistics` file. You can get both `.bugle` files in `<root bugle directory>` `/doc/examples`.

Bugle uses two configuration files, `filters` and `statistics`. To run bugle the provided `gldb-gui` on the target application, just run it inside the working directory and it'll run bugle's user interface, do not forget to change the chain in `Options-> Target`. The specified chain 'must exist inside `filters`.

A.2.1 *Filter Configuration*

Using Bugle requires configuring `filters` and `statistics`. This section will refer to the configuration of these two files and the results they produce. All filter chains exist inside `filters` file. For instance, an empty chain would be:

```
chain pass
{
}
```

All `filtersets` need to be placed inside a `chain`.

A.2.1.1 *Statistics configuration*

This file is used two determine statistics format, its existence is imperative whenever using stats outputs such as `logstats` and `showstats`,see section A.2.1.3. It can be configured to fetch the statistics in four forms:

- `d(<stat>)` - the difference in the stat (end - start)

- `a(<stat>)` - the average value of the stat

- `s(<stat>)` - the value at the start

- `e(<stat>)` - the value at the end

  `<stat>` can be one of the following values:

- `"frames"` - Requires `stats_basic`, current frame number;

- `"seconds"` - Requires `stats_basic`, current time in seconds;

- `"triangles"` - Requires `stats_primitives`, current number of triangles;

- `"batches"` - Requires `stats_primitives`, current number of batches of triangles;

- `"fragments"` - Requires `stats_fragments`, current number of fragments;

- `"calls:*"` - Requires `stats_calls`, current number of GL calls, the * can be the function name or left alone as a wild-card;

- `"calltimes:*"` - Requires `stats_calltimes`, current number of time spent on GL calls, the * can be the function name or left alone as a wild-card;

- `"calltimes:total"` - Requires `stats_calltimes`, current number of total time spent on GL calls;

Using these options it is possible to create the basic FPS counter, the existing example from Bugle's files is:

```
"frames per second" = d("frames") / d("seconds")
{
    precision 1
    label "fps"
}
```

As can be seen it works with simple math functions such as $FPS = \frac{frames}{seconds}$, using the possible stats options many different stats can be outputted. The example above simply needs to be copied and pasted into the `statistics` file in order to be available.

Another example of number of calls per frame would be:

```
"calls per frame" = d("calls:*") / d("calls:*") *
                    d("calls:*") / d("frames")
{
    precision 0
    label "*"
}
```

The `d("calls:*") / d("calls:*")` exists to cause a NaN when it is 0 so it won't be outputted and flood the statistics with zeros. This will list the number of times each function was called.

A.2.1.2 *Statistics filterset*

There is a group of `filtersets` which work together to output stats. In order to gather statistic relevant data the following `filtersets` are required:

- `filterset stats_basic`

- `filterset stats_primitives`

- `filterset stats_fragments`

- `filterset stats_calls`

- `filterset stats_calltimes`

The data each of these `filtersets` gathers is described in the statistics section. However, these filters alone will not produce any output, they exist only to gather data. In order to produce some output, the `filterset showstats` or `filterset logstats` are required, the former outputs in the application window while the latter outputs to a log-file, and both can be used at the same time.

To show and log FPS simultaneously it is possible to combine the examples from statistics section and create the following chain:

```
chain logandshowfps
{
    filterset stats_basic
    filterset showstats
    {
        show "frames per second"
    }
    filterset logstats
    {
        show "frames per second"
    }
    filterset log
    {
        filename "bugle.log"
    }
}
```

When this chain is used a FPS counter shall appear on the top left corner, it'll also create a `bugle.log` which logs the FPS every frame. As can be seen in the example above a `show` option is used on both `showstats` and `logstats`, this will write the `"frames per second"` from `statistics` file verbally.

`filterset showstats` can use additional options such as `graph` which draws a graph, `time` which decides the update rate, `key_accumulate` and `key_unaccumulate` which toggles the average statistics from the moment `key_accumulate` is pressed, `key_unaccumulate` undoes the `key_accumulate`'s effects.

The following chain will show a much more complete use of `showstats` also shown in figure 20:

```
chain showstats
{
```

```
filterset stats_basic
filterset stats_primitives
filterset stats_fragments
filterset stats_calls
filterset showstats
{
    show "frames per second"
    show "batches per frame"
    show "calls per frame"
    graph "triangles per second"
    graph "fragments per second"
}
}
```
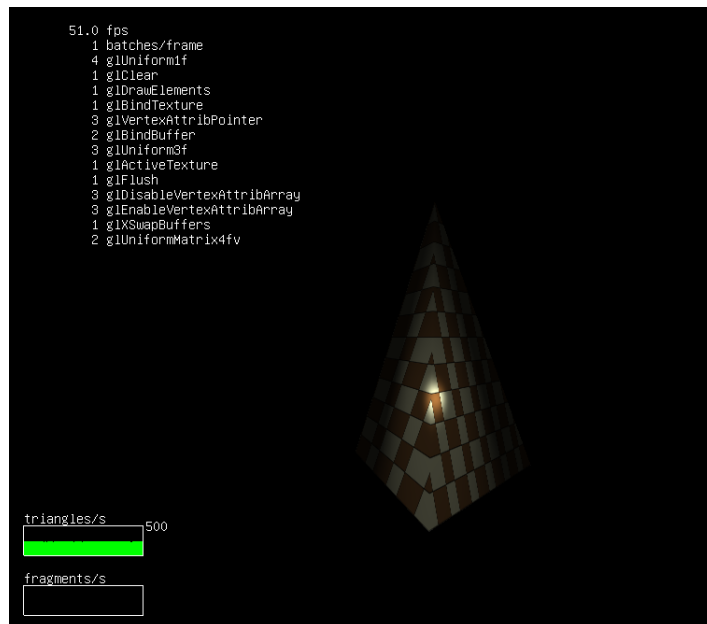

Figure 20.: Bugle showstats example.

A.2.1.3 *Trace and Log filterset*

The most common features of the open source debugger is to trace and log OpenGL calls. In order for Bugle to accomplish this two `filtersets` are required, the `filterset trace` and `filterset log`. The former captures data for the log and the latter specifies where to write. For example:

```
chain trace
```

```
{
    filterset trace
    filterset log
    {
        filename "bugle.log"
    }
}
```

The `filterset trace` does not require any additional options, the `filterset log` has more additional options beside `filename`:

- `filename` - Name of the output log file;

- `file_level`, `stdout_level` and `stderr_level` - Specifies the output level for log file, standard out and standard error respectively, its possible values are:
  - `0` - No logging;
  - `1` - Errors that usually lead to immediate termination like bad log file name or when using the `filterset showstats` without the appropriate stats `filtersets` when showing a stat (like attempting to show FPS without `filterset stats_basic`);
  - `2` - Warnings, usually the ones that affect Bugle's functions for example invalid file names when making screenshots prevents the usage of the filter itself, this results in a warning;
  - `3` - Notices, usually OpenGL errors or undefined behaviour, for example when `filterset checks` ignores an OpenGL error in the application;
  - `4` - Information from `filtersets` that generate logs such as `filterset trace` or `filterset showextensions`;
  - `5` - Bugle's debugging message, some `filtersets` output debugging messages, for example all `filtersets` which require input like the `filterset screenshot` generate input debug messages.

- `format` - Overrides default log format, the following printf-style escapes are allowed:
  - `%l` - Log level;
  - `%f` - Filter-set;
  - `%e` - Event;
  - `%m` - Log message;
  - `%p` - Process ID;
  - `%t` - Thread ID;
  - `%%` - Literal % .

An example of a Bugle log is the following log result:

```
[INFO] trace.call: glXQueryExtension(0x985530, NULL, NULL) = True
...
[INFO] trace.call: glClear(GL_COLOR_BUFFER_BIT)
[INFO] trace.call: glUniformMatrix4fv(8, 1, GL_TRUE, 0x7fff5086453c ->
{ {
{ 1.08247, 0, -0.011336, 0 },
{ 4.74842e-05, 1.73204, 0.00453425, 0.018138 },
{ 0.0106833, -0.00267088, 1.02014, 2.06059 },
{ 0.0104717, -0.00261799, 0.999942, 3.99999 } } })
[INFO] trace.call: glUniformMatrix4fv(9, 1, GL_TRUE, 0x7fff5086457c ->
{ {
{ 0.999945, 0, -0.0104718, 0 },
{ 0, 1, 0, 0 },
{ 0.0104718, 0, 0.999945, 1 },
{ 0, 0, 0, 1 } } })
[INFO] trace.call: glUniform3f(1, 1, 1, 1)
[INFO] trace.call: glUniform1f(0, 0)
[INFO] trace.call: glUniform3f(3, 0, 0, 1)
[INFO] trace.call: glUniform1f(2, 0.2)
[INFO] trace.call: glUniform3f(4, 0, 0, -3)
[INFO] trace.call: glUniform1f(5, 1)
[INFO] trace.call: glUniform1f(7, 32)
...
```

A.2.1.4 *Error checking filtersets*

This section will refer to error logging and control `filtersets`.

The `filterset checks` will search for undefined behaviour such as an OpenGL call attempt to use unreadable memory, `filterset error` exists as a safeguard calling glGetError after each function allowing the application to read its own errors without Bugle accidentally collecting them. Finally, `filterset showerror` functions like trace but will write errors to standard error instead.

```
chain errordebugging
{
    filterset checks
    filterset error
    filterset showerror
}
```

For instance `filterset checks` when `glEnableVertexAttribArray(7)` tries to read an undeclared buffer the following output to the log may occur:

```
[NOTICE] checks.error: illegal generic attribute array 7 caught in
glDrawElements (unreadable.memory); call will be ignored.
```

A.2.1.5 *KHR_Debug filterset*

This `filterset` will install a `glDebugMessageCallbackARB`, however it'll require a debug context. The option `sync` can be turned on by adding `sync yes`, it'll enable `GL_DEBUG_OUTPUT_SYNCHRONOUS`.

A filter example for this `filterset` while forcing debug context is the following:

```
chain debugcontext{
    filterset contextattribs
    {
        flag "debug"
    }
    filterset logdebug
    {
        sync no
    }
}
```

For example when an error occurs it can successfully output:

```
X Error of failed request:  GLXBadProfileARB
  Major opcode of failed request:  153 (GLX)
  Minor opcode of failed request:  34 ()
  Serial number of failed request:  43
  Current serial number in output stream:  44
```

A.2.1.6 *Context attributes and extension override filtersets*

The two mentioned `filtersets` in this section will force the application to take some options, for example `filterset extoverride` can force the OpenGL version while `filterset contextattribs` can force the application into debug mode.

The possible extension overrides are:

- `version <value>` - This option will force OpenGL version string into the indicated value, masking its real value;

- `disable "all"` - This will disable all extensions that are not explicitly enabled;

- `disable <value>` - This will disable the indicated extension;

- enable <value> - This will enable the indicated extension;

The possible context attributes overrides are:

- major <value> - Specifies the value for GLX_CONTEXT_MAJOR_VERSION_ARB;

- minor <value> - Specifies the value for GLX_CONTEXT_MINOR_VERSION_ARB;

- flag <flag> - Specifies the flags for GLX_CONTEXT_FLAGS_ARB, the available flags are:
    - debug
    - forward
    - robust

  If the value is any of the designated flags then it shall be turned on, if preceded by a ! then the opposite shall occur;

- profile <flag> - Specifies the flags for GLX_CONTEXT_PROFILE_MASK_ARB, the available flags are:
    - core
    - compatibility
    - es2

  Works exactly like flag.

An example of overriding would be:

```
chain override
{
    filterset extoverride
    {
        disable "GL_EXT_framebuffer_object"
        disable "GL_EXT_framebuffer_blit"
    }
    filterset contextattribs
    {
        major 3
        minor 3
        flag "debug"
        flag "robust"
        profile "core"
        profile "!compatibility" #disables compatibility
    }
}
```

It should also be noted that `extoverride` tries to suppress the extensions but can't fully prevent them, in case it fails to suppress it throws a warning announcing the failure in the following format:

```
[NOTICE] extoverride.warn: glXGetProcAddressARB was called, although
the corresponding extension was suppressed
```

A.2.1.7 *showextensions filterset*

This `filterset` will output all extensions into standard error when the application terminates, the already existing `chain` is composed as the following:

```
chain showextensions
{
    filterset showextensions
    filterset log
    {
        stderr_level 4
    }
}
```

An example of `filterset showextensions`'s output is the following:

```
[INFO] showextensions.gl: Min GL version: 2.0
[INFO] showextensions.glx: Min GLX version: 1.3
[INFO] showextensions.ext: Required extensions: GLX_ARB_get_proc_address
```

A.2.1.8 *Compilable C source filterset*

Bugle's capable of creating a compilable trace (create a `*.c` file instead of a common log), this trace file is compilable and work's as a replay of the original application.

The following chain is sufficient:

```
chain exe
{
    filterset exe
    {
        filename "exetrace.c"
    }
}
```

Unfortunately, at the time of this writing, this filter was in an experimental phase and its output is still not robust enough to be of real use. Nevertheless, its an interesting additional feature.

A.2.1.9 *Screenshot filterset*

`filterset screenshot` is used to create both screenshots and videos. The following `filterset` shows an example of such a chain:

```
chain screenshot
{
    filterset screenshot
    {
        filename "bugle%d.ppm" # A %d is the frame number
        key_screenshot "C-A-S-S" #Ctrl+Alt+Shift+S
    }
}
```

To create video files the following options are available:

- `video yes` - By default it is no (screenshot example) but in order to create a video it must be set to yes;

- `codec <value>` - Set's the codec of the output video, for example value can be "mpeg4";

- `bitrate <value>` - Set's the approximate rate of bit's per second for the video encoding;

- `allframes <value>` - If set to "yes" it'll capture all frames of the application, by default the video captures at 30 FPS, however, if set to "yes" the speed may vary;

- `lag <value>` - Sets a latency between video capture and encoding.

In order to create a video there is the following `chain` example:

```
chain video
{
    # Press C-V to start and to stop recording
    filterset screenshot C-V inactive
    {
        video "yes"
        filename "bugle.avi"
        codec "mpeg4"
        bitrate "7500000"
        allframes "no"
    }
}
```

A.2.1.10 *eps filterset*

The `filterset eps` works like `filterset screenshot`, however instead of screenshots it'll create a vector graphics file in the `*.eps` format. This file can be used by certain applications like MS Word or Adobe Illustrator. In the figure **??** you can see what it looks like when converted to png, the example is the same application as figure 20.

```
chain eps
{
    filterset eps
    {
        filename "bugle.eps"
        key_eps "C-A-S-W"   #Control+Alt+Shift+W
    }
}
```

A.2.1.11 *frontbuffer filterset*

The `filterset frontbuffer` forces the application to always output the frontbuffer's state.

```
chain frontbuffer
{
    filterset frontbuffer
}
```

A.2.1.12 *Wireframe filterset*

The `filterset wireframe` forces the application to render in wireframe mode.

```
chain wireframe
{
    filterset wireframe
}
```

<div style="text-align: right;">

B

</div>

APITRACE

B.1 INSTALLATION

Before building the application you need the following application to build the program:

- Qt version $>=$4.7 and $<$5 (version 5 won't work)

- Visual studio 2010 with SP1

- Cmake (tested with 2.8)

In order to build the application it is best to set the project in a folder close to root (for example `C:\temp\`) due to cmake limitations.

Use command line and jump to the project folder (in this case "cd C:/temp/apitrace-master") then:

```
set PATH=%PATH%;C:\Qt\4.8.5\bin
qmake -query
cmake-gui -H%cd% -B%cd%\build
```

Take note that the `set` command must be used for the Qt bin where qmake.exe exists, in this case it was `C:\Qt\4.8.5\bin`. Using these commands will open cmake GUI, then press the configure button and set to Visual Studio 10 (Visual Studio 10 = Visual studio 2010, however Visual Studio 11 = Visual studio 2012). In Linux systems use `export PATH=\$PATH;<QT Bin directory>`.

This test was created on a 32 bit computer, according to the official guide in a 64 bit computer you'll need to use `cmake-gui -H% cd% -B% cd% build -DENABLE_GUI=FALSE` and configure for Visual Studio 10 win64 instead.

After this a visual studio solution will be generated and you can be either with the solution or using the following command:

```
cmake --build build --config MinSizeRel
```

If you don't have QT properly configured it won't generate a solution with qapitrace.exe.

B.2.1 *Tracing*

Run with `apitrace trace <Target Executable>` to start tracing, apitrace will generate a `<Target Executable>.model` file which can be read by qapitrace.

Take note that he trace should be run on the target's directory (`<Target Executable Path> apitrace trace <Target Executable>`) so it may be advantageous to add apitrace's bin folder to PATH variable, otherwise it is necessary need to run it like this: `<Apitrace path>\apitrace trace<Target Executable>`. The command `apitrace trace <Target Executable Path>/<Target Executable>` will not work in most cases due to detecting the current directory as the working directory.

In Windows it is possible to add the folder to the path variable in "Control Panel/System and Security/System" → Advanced system properties → Ambient Variables.

B.2.2 *Retracing*

In order to replay the trace use qapitrace or use glretrace.exe, using glretrace on command line will the print the tracing warnings on the command line.

Dumping OpenGL frame call – it is possible to dump a call with `apitrace replay -D <frame number> <trace file>.trace > <output file>.json` which will dump a file with all calls related to the frame, it will also place textures in their byte code.

B.2.3 *Output replay to video*

It is possible to output the replay using ffmpeg to create a video file, the following command will create the video file in mp4:

```
apitrace dump-images -o - <trace file>.trace | \\
ffmpeg -r 30 -f image2pipe -vcodec ppm -i pipe: -vcodec mpeg4 -y \\
<output file>.mp4
```

It is also possible to use libav instead:

```
apitrace dump-images -o - <trace file>.trace | \\
avconv -r 30 -f image2pipe -vcodec ppm -i - -vcodec mpeg4 -y \\
<output file>.mp4
```

To reduce a big trace file it's possible to trim it by using the following command:

```
apitrace trim --exact --frame 0-<target frame> -o trimed.trace \\
application.trace
```

However trimming it from a beginning different from 0 will most likely create an unreplayable trace file (because it is missing its first initialize calls).

B.2.5 *Profiling trace*

In order to profile a trace, one of the already existing scripts in apitrace has to be used, in this case the scripts were copied to the example application folder so the used command was the following:

```
apitrace replay --pgpu --pcpu --ppd models.trace | profileshader.py
```

The output of the profiler is the following:

```
+-----------+-------------+-------------------+-------------+-------------+
|  program  |  Draws [#]  |  Duration [ns]  v | Per Call[ns] | Longest[id] |
+-----------+-------------+-------------------+-------------+-------------+
|        4 |       38706 |         839456832 |        21688 |       62325 |
|        1 |       42838 |          91206304 |         2129 |       50705 |
|        0 |         167 |           1027008 |         6149 |        3037 |
+-----------+-------------+-------------------+-------------+-------------+
```

Also as noticed in the command a `--pgpu --pcpu --ppd` was used, this commands are used for the following:

- –pgpu record gpu times for frames and draw calls.

- –pcpu record cpu times for frames and draw calls.

- –ppd record pixels drawn for each draw call.

For instance if only –pgpu was used it would result in the following output:

```
+-----------+-------------+-------------------+-------------+-------------+
|  program  |  Draws [#]  |  Duration [ns]  v | Per Call[ns] | Longest[id] |
+-----------+-------------+-------------------+-------------+-------------+
|        4 |       38706 |         841012896 |        21728 |       62033 |
|        1 |       42838 |         116815840 |         2726 |       38851 |
|        0 |         167 |           5082688 |        30435 |        3037 |
+-----------+-------------+-------------------+-------------+-------------+
```

In `--pcpu` case, it would result in the following output:

```
+-----------+-------------+-------------------+-------------+------------+
|  program  |  Draws [#]  |  Duration [ns]  v | Per Call[ns]| Longest[id]|
+-----------+-------------+-------------------+-------------+------------+
|         1 |       42838 |                 0 |           0 |       3112 |
|         0 |         167 |                 0 |           0 |         15 |
|         4 |       38706 |                 0 |           0 |       3039 |
+-----------+-------------+-------------------+-------------+------------+
```

And lastly for `--ppd` it would result in the following output:

```
+-----------+-------------+-------------------+-------------+------------+
|  program  |  Draws [#]  |  Duration [ns]  v | Per Call[ns]| Longest[id]|
+-----------+-------------+-------------------+-------------+------------+
|         1 |       42838 |                 0 |           0 |       3112 |
|         0 |         167 |                 0 |           0 |         15 |
|         4 |       38706 |                 0 |           0 |       3039 |
+-----------+-------------+-------------------+-------------+------------+
```

This output occurred probably because there was no relevant cpu times and pixel draws, thus only `--pgpu` had data to fill the table, as a result `--pcpu` and `--ppd` were pointless in the example application. Perhaps OpenGL only applications will not output cpu times nor pixel draws.

This type table can be hard to understand for a beginner, most would not understand what program and draws are.

The program column indicates the shader program (the combination of vertex, fragment, geometry, etc. shaders), each loaded shader set has its own program id, the example has two different programs, the program 0 is the default no shader that existed in this test application (reason why it has so little amount of calls is because they are initializers), program 1 used for outputting text labels and program 4 for the model.

The draws columns indicate the number of draws, this is the number of gl calls for the specified program, it does not include all calls such as glGetIntegerv and it also excludes anything between glFlush and glClear.

The other three columns are: Duration (total duration spent on the program), Per Call (the average duration for each gl call) and finally Longest (Maximum duration on a single gl call).

GLINTERCEPT

C.1 INSTALLATION

GLIntercept source code is downloaded with Visual Studio 2008 solutions in the `Src\WorkSpaces` folder, one solution is for GLIntercept main dll wrapper and functions called `GLIntercept.sln`, another is for the plugins called `GLI Plugins.sln` and finally one for `SciTE.sln` a GUI for the text editor in the shader editor plugin, SciTE is a third party solution which will not be analysed.

After building you'll need to tell in the configuration files where is you build folder, the build will be outputted in the GLIntercept's `Bin` folder, it'll need to be changed in different sections:

- `FunctionLog` - change in `BaseDir = "<bin folder>\XSL";`

- `PluginData` - change in `BaseDir = "<bin folder>\Plugins";`

- `InputFiles` - change in `GLFunctionDefines = "<bin folder>\GLFunctions \gliIncludes.h";`

GLIntercept's OpenGL wrapper is built in `Bin\MainLib` folder. It is much easier to test using the official installer thus building is only necessary if you want to develop GLIntercept's wrapper or plugins, if you want additional functions you should use the plugin's solution.

In order to debug with GLIntercept you'll need to copy or create the config.ini into the debugging application's workspace along with the wrapper, to inject a DLL .

C.2 USAGE AND CONFIGURATION

C.2.1 *Tracing*

GLIntercept makes normal tracing calls, whenever an OpenGL function is called its function is logged according to the configurations.

Basic logging must have log per frame disabled:

```
LogPerFrame
{
  Enabled = False;
  FrameStartKeys = (ctrl,shift,f);
  OneFrameOnly = True;
}
```

Also it can be configured according to the `FunctionLog` bracket, in such case obviously `LogEnabled` must be true, `LogFlush` only purpose is to decide whether to output as soon the function is traced, the rest is redundant:

```
FunctionLog
{
  LogEnabled = True;
  LogFlush   = False;
  //LogPath      = "c:\temp\";
  LogFileName = "gliInterceptLog"

  //AdditionalRenderCalls = ("glClear");

  //LogMaxNumFrames = 200;

  //LogFormat   = XML;

  XMLFormat
  {
    XSLFile = gliIntercept_DHTML2.xsl;
    BaseDir = "C:\Program Files\GLIntercept_1_2_0\XSL";
  }
}
```

The following is an example of a GLIntercept txt log:

```
===============================================================================
 GLIntercept version 1.20 Log generated on: Sun Jul 06 16:36:14 2014

===============================================================================

wglChoosePixelFormat(EB010F83,0012FCE0)=7
wglChoosePixelFormat(EB010F83,015112C0)=7
wglSetPixelFormat(EB010F83,7,015112C0)=true
```

```
wglCreateContext(EB010F83)=00010000
wglGetCurrentContext()=00000000
wglGetCurrentDC()=00000000
wglMakeCurrent(EB010F83,00010000)=true
glGetString(GL_VERSION)="3.3.0"
glGetString(GL_VENDOR)="NVIDIA Corporation"
glGetString(GL_RENDERER)="GeForce 9300M GS/PCIe/SSE..."
...
glLoadIdentity()
glRotatef(270.000000,1.000000,0.000000,0.000000)
glScalef(5.000000,5.000000,5.000000)
glTranslatef(0.000000,0.000000,-1.500000)
glGetDoublev(GL_MODELVIEW_MATRIX,0025D598)
...
```

In case `LogFormat` is active and set to XML it will output a XML file with the following format (. . . means that there is more content in the middle):

```
<?xml version='1.0'?>
<?xml-stylesheet type="text/xsl" href="gliIntercept_DHTML2.xsl"?><GLINTERCEPT>
<HEADER>
<VERSION>1.20</VERSION>
<TIMESTAMP>Sun Jul 06 16:37:19 2014
</TIMESTAMP>
</HEADER>
<FUNCTIONS>
<FUNCTION>
<NAME>wglChoosePixelFormat</NAME>
<PARAM name="a" type="GLvoid*"><VALUE data="A4010FD6"/></PARAM><PARAM name="b"
type="GLvoid*"><VALUE data="0012FCE0"/></PARAM>
<RETURN type="GLint"><VALUE data="7"/></RETURN>
</FUNCTION>
<FUNCTION>
<NAME>wglChoosePixelFormat</NAME>
<PARAM name="a" type="GLvoid*"><VALUE data="A4010FD6"/></PARAM><PARAM name="b"
type="GLvoid*"><VALUE data="005212C0"/></PARAM>
<RETURN type="GLint"><VALUE data="7"/></RETURN>
</FUNCTION>
...
<FUNCTION>
<NAME>glVertex3f</NAME>
</FUNCTION>
</FUNCTIONS>
</GLINTERCEPT>
```

C.2.2 *Frame Logging*

Generates a log similar to tracing, however each log is created on the current frame when using `<ctrl-shift-f>`. This command will cause GLIntercept to capture the log that from the frame and save it in the logging folder.

This is a variation of basic tracing requiring activation of log per frame, its format will be according to:

```
LogPerFrame
{
  Enabled = True;
  FrameStartKeys = (ctrl,shift,f);
  OneFrameOnly = True;
}
```

C.2.3 *Shader Editor*

The shader editor is either bugged or incomplete, whenever it tries to compile the application it always get uniform mismatch error, it also cause the program to freeze for a while when the editor is open. It does seem to work for `.cg` shaders.

In order to use the shader editor press `<ctrl-shift-s>` when the program is running, the following plugin is also needed:

```
PluginData
{

  BaseDir = "C:\Program Files\GLIntercept_1_3_0\Plugins";

  Plugins
  {
    (...)

    OpenGLShaderEdit = ("GLShaderEdit/GLShaderEdit.dll")

    (...)

  }
  (...)
}
```

C.2.4 *ARB_debug_output Logging*

ARB_ debug_ output is produced along the XML logs, it is created by adding the following lines:

```
PluginData
{

  BaseDir = "C:\Program Files\GLIntercept_1_3_0\Plugins";

  Plugins
  {
    (...)


    DebugContext = ("GLDebugContext/GLDebugContext.dll")
    {
      ForceDebugMode = True;
      LogToFunctionLog = True;
      LogToErrorLog = True;
      BreakOnMessage = False;

      MessageControl
      {
        AllMessages = ("Dont Care", "Dont Care", "Dont Care", True)
      }
    }


    (...)
  }
  (...)
}
```

C.2.5 *Extension override*

It is possible to change the program extensions like version of OpenGL to use, however it will only override the output information, not the exact version (internally the application will use the same versions as it would normally use, unless the output information is used for some function).

```
PluginData
{

  BaseDir = "C:\Program Files\GLIntercept_1_3_0\Plugins";


  Plugins
  {



    ExtensionOverride = ("GLExtOverride/GLExtOverride.dll")
    {
      VersionString       = "1.1.0 - Custom version string";
      ShaderVersionString = "1.0.0 - Custom shader version string";
    }



  }
}
```

For example the configuration above will alter the OpenGL version shown in the following screen-shot in figure 21:

Figure 21.: Extension override results

The existing possible extension overrides are:

- `VendorString        = "Custom vendor string";`

- `RendererString      = "Custom renderer string";`

- `VersionString       = "Custom version string";`

- `ShaderVersionString = "Custom shader version string";`

- `ExtensionsString    = (GL_EXT_A, GL_EXT_B, GL_EXT_C, GL_EXT_D);`

- AddExtensions = (GL_ARB_shading_language_100,
  GL_ARB_shader_objects, GL_ARB_fragment_shader, GL_ARB_vertex_shader);

- RemoveExtensions = (GL_S3_s3tc,WGL_EXT_swap_control);

- WGLExtensionsString = (GL_EXT_A, GL_EXT_B, GL_EXT_C, GL_EXT_D);

- WGLAddExtensions = (GL_EXT_A,GL_EXT_B);

- WGLRemoveExtensions = (WGL_ARB_buffer_region,
  WGL_ARB_extensions_string, WGL_NV_render_texture_rectangle);

C.2.6 *Function statistics*

This extension does not is not included in the config examples provided by GLIntercept, however it does exist in the plugins folder.

This plugin creates a call count list after exiting the application allowing the user to check which functions were called the most.

```
PluginData
{

  BaseDir = "C:\Program Files\GLIntercept_1_3_0\Plugins";

  Plugins
  {


    FunctionStats = ("GLFuncStats\GLFuncStats.dll");



  }
}
```

The following is an example of this plugin's output, . . . was used on the list to shorten it.

```
======= OpenGL function call statistics ==========
 Total GL calls: 82938
 Number of frames: 7  Average: 11888 calls/frame (excluding first
frame count of 11606)
```

```
======= OpenGL function calls by call count ==========
glNormal3f ................................... 38126
glVertex3f ................................... 38126
glPixelStorei ................................ 2436
glGetIntegerv ................................ 1221
wglGetProcAddress ............................ 717
glBegin ...................................... 382
glEnd ........................................ 381
glMap2f ...................................... 336
glEvalMesh2 .................................. 256
glBitmap ..................................... 203
glEnable ..................................... 80
glMapGrid2f .................................. 80
glGetError ................................... 62
glMatrixMode ................................. 44
glNamedProgramLocalParameters4fvEXT .......... 40
glGetDoublev ................................. 40
glLoadIdentity ............................... 38
glNamedProgramLocalParameter4fvEXT ........... 37
glTranslatef ................................. 32
glRotatef .................................... 32
glDisable .................................... 31
...


======= OpenGL function calls by name ==========
glBegin ...................................... 382
glBindProgramARB ............................. 19
glBitmap ..................................... 203
glClear ...................................... 8
glColor3f .................................... 8
glColor4f .................................... 7
glDepthFunc .................................. 8
glDisable .................................... 31
glEnable ..................................... 80
glEnd ........................................ 381
glEvalMesh2 .................................. 256
...
```

D

GLSLDEVIL/GLSL-DEBUGGER

D.1 INSTALLATION

In order to build GLSL you'll need to download it from https://github.com/GLSL-Debugger/GLSL-Debugger repository, have cmake, glut libraries, glew libraries, bison, flex and qt4 installed.

Once you download the files from the repositories build them with cmake, you can use `cmake-gui -H%cd% -B%cd%\build` on Windows and initiate the build procedures.

In Windows you can create a Visual Studio solution with cmake, on Unix systems you can create a common make solution meaning you can build relying on `make` command. Windows systems may force you to indicate the appropriate library and include directories.

In Unix 64-bit system a problem regarding `libGL.so` may occur within `/usr/lib/x86_64-linux-gnu/`, it needs re-linking it to the current driver, otherwise the debugger may not work properly. For example in Ubuntu 13.10 64-bit using nvidia 319 update drivers will require to link to `/usr/lib/nvidia-319-updates/libGL.so`.

In order to use GLSL you'll need to start it on its own working directory else it won't find the appropriate libraries, in the build folder created during cmake there should be a `bin` folder containing the executable `glsldb`.

D.2 USAGE AND CONFIGURATION

D.2.1 *Graphic User Interface*

GLSL is a very graphic debugger, one of the first steps to use GLSL is to choose which application to debug, it can do so with the `Open Program` button or `<ctrl-o>`, afterwards it needs the program text box (the application executable) to be filled. Optionally the arguments and working directory can also be inputted, the dialog is as shown in figure 22.
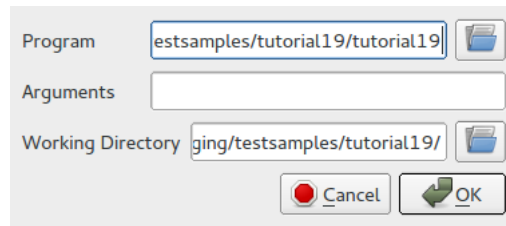
Figure 22.: GLSL opening a new application to debug, in the example its opening an application called tutorial19

Once that's done GLSL is ready to debug, it'll start the debugging by pressing run button (green gear icon) or F5. Watching the figure 23 and 24 should be much more intuitive than a text description.
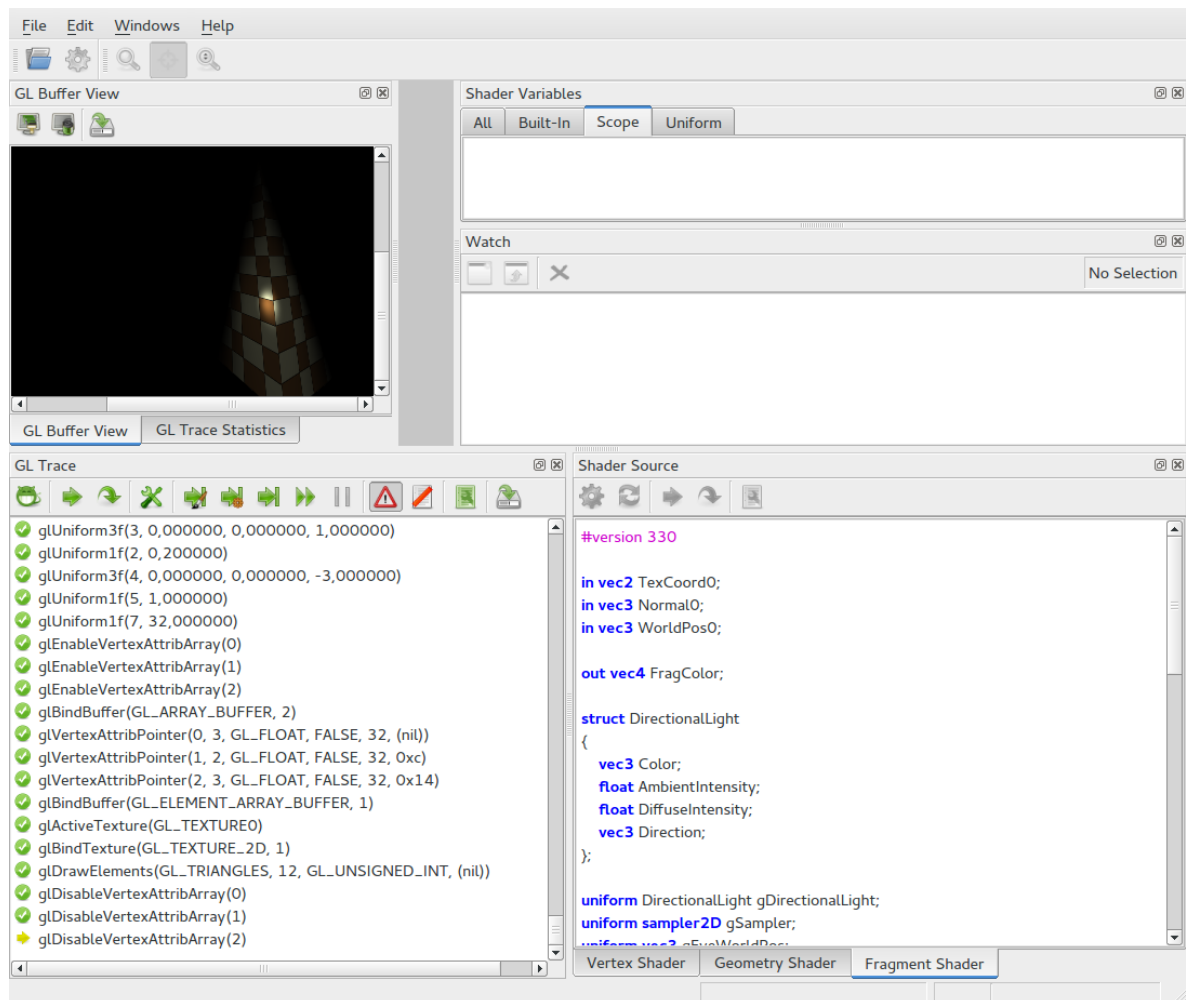

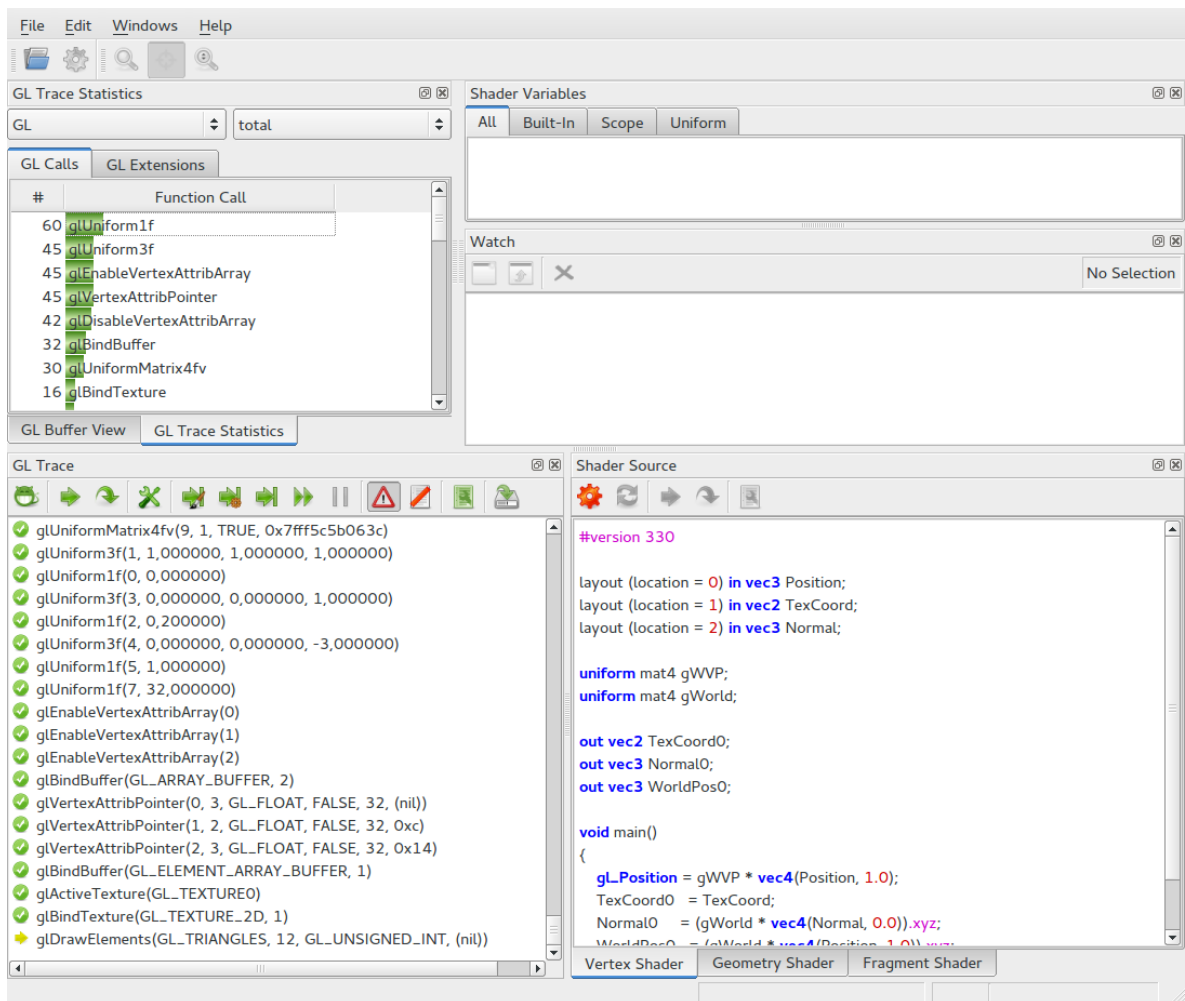Figure 23.: GLSL showing buffer view and fragment shader.

Figure 24.: GLSL-Debugger featuring trace statistics and the vertex shader.

The following subsections shall describe what each function does, these sections shall be grouped by sub-window, each button is listed from left to right.

D.2.1.1 *GL Trace*

This sub-window logs all the trace results, the debugging starts from here.

Run

This is the most basic function and essentially starts the debugging, it'll unlock most of the functions.

Step

This can only be used after pausing during a run, it'll trigger the next function and pause again.

Skip

This can only be used after pausing during a run, it'll skip the next function and pause again.

Edit

This can only be used after pausing during a run, it is used to edit the parameters of the next function to trigger.

Jump to next Draw Call

This will resume running but pause as soon it reaches a draw call, when it reaches the draw call it'll enable the Shaders sub-window. Its hotkey is <F7>.

Jump to next Shader Switch (F6)

This will resume running but pause as soon it reaches a the next change of shaders or essentially the next glprogram function. Its hotkey is <F6>.

Jump to next User Defined OpenGL

This will resume running but pause as soon it reaches a particular function, the function is chosen by the user when a popup appears. Its hotkey is <ctrl-F6>.

Run (F8)

This will resume running and won't pause unless Stop button is pressed or an error occurs when *Halt on Error* is enabled. Its hotkey is <F8>.

Stop (Alt+Break)

This will pause the application enabling diverse debugging options. Its hotkey is <alt+break>.

Halt on Error

When this is enabled the debugger will pause whenever an error occurs.

Disable GL Trace

When this is enabled the debugger will not record any trace, essentially it'll almost run the application without any form of debugging.

Show GL Trace settings

This will show a popup which allows the user to choose which functions to be recorded in the trace.

Save GL Trace

Use this function in order to save a log file.

D.2.1.2 *Shader*

This sub-window is only available when the next function to trigger is a draw call. Unfortunately only OpenGL 2 shaders work (version 110), this functionality is highly bugged.

Debug Shader

This initiates shader debugging, because of the limitations it is untested. Its hotkey is `<ctrl+F5>`.

Reset Debug Session (Ctrl+Shift+F5)

This restarts shader debugging, untested since no old shader program worked. Its hotkey is `<ctrl+F5>`.

Step (F11)

Same as *Step* from GL Trace, this one works for shader debugging. Its hotkey is `<F11>`.

Step Over (F10)

Same as *Skip* from GL Trace, this one works for shader debugging. Its hotkey is `<F10>`.

Edit Per-Fragment Option

This button will open output options for the fragment shader, thus it is only available in the fragment shader tab, the options are shown in figure 25. Its hotkey is `<F4>`.

D.2.1.3 *GL Trace Statistics*

This window will show a count of the called GL functions, it may either count per frame or a total called since the beginning of the debugging.

Combo Boxes

These combo boxes will determine how and which statistics to display, the left combo box offers which type of GL function to display such as GL, GLX and WGL, the right combo box decides how they are counted from total to calls per frame.
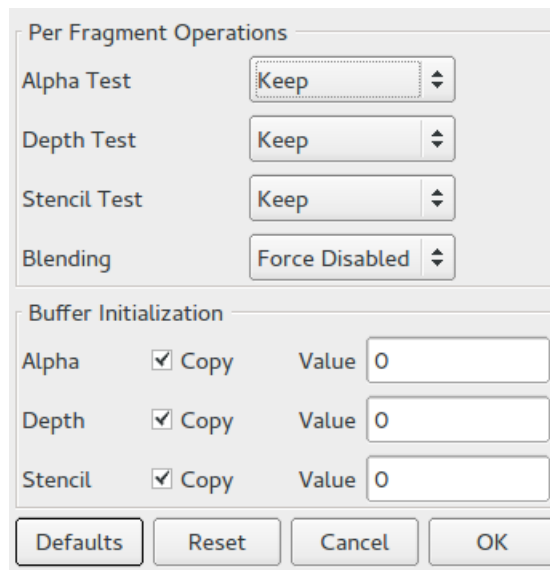
Figure 25.: Fragment shader per-fragment options.

Tabs

This sub-window is split into two tabs, *GL Calls* and *GL Extensions*, GL Calls will display results per function and GL extension will display the results according to the extensions, these results are also influenced by the combo boxes.

D.2.1.4 *GL Buffer View*

This sub-window will allow the user to capture the front buffer, it cannot choose which buffer to capture.

Capture

This is the basic manual capture, it'll capture the front buffer from the state of the current gl function (not frame), it may capture nothing if it is used before anything is drawn.

Automatic Capture

This is the same as capture, however when on it'll capture after each gl function.

Save as image

This allows the current captured buffer to be saved as an image.

Shader Variables and Watch

GLSL can only view shader variables during shader debugging, during shader debugging the uniforms and variables are split according to the tabs as shown in the screenshots figures 26, 27 and 28.

Watch is available when one of the uniforms is chosen to be put on watch (double clicking). Once on watch it is possible to view them as shown in the screenshots, the figure 26 shows fragment coordinate viewing, 27 shows fragment color viewing and 28 shows fragment position viewing. All shown views interact and update according to the shader debugger.

Since fragment color viewing is the same as outputting the fragment shader it is possible to see the fragment shader results by debugging, the triangle shown in the figures 26 and 27 is the 3D model used in the application.
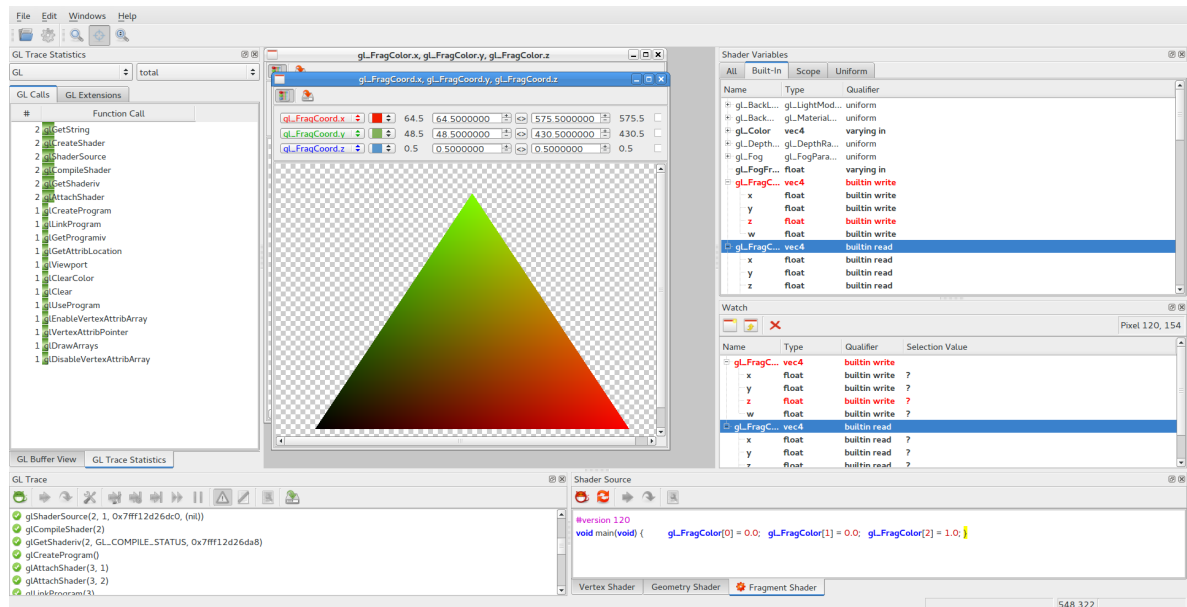

Figure 26.: Fragment coordinates viewer.
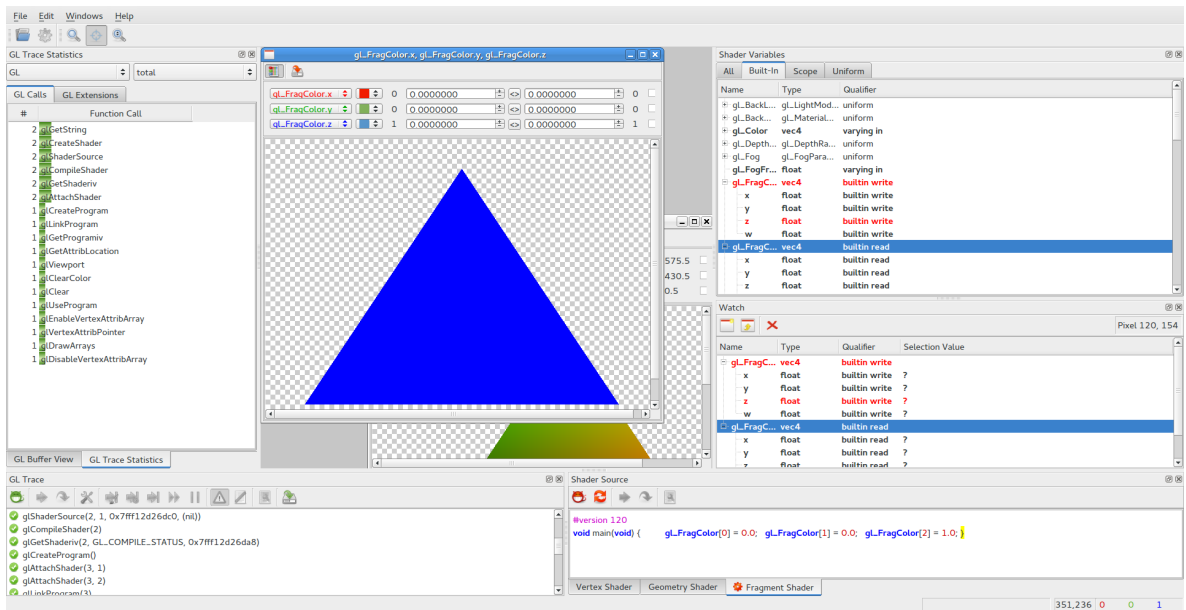
Figure 27.: Fragment color viewer.

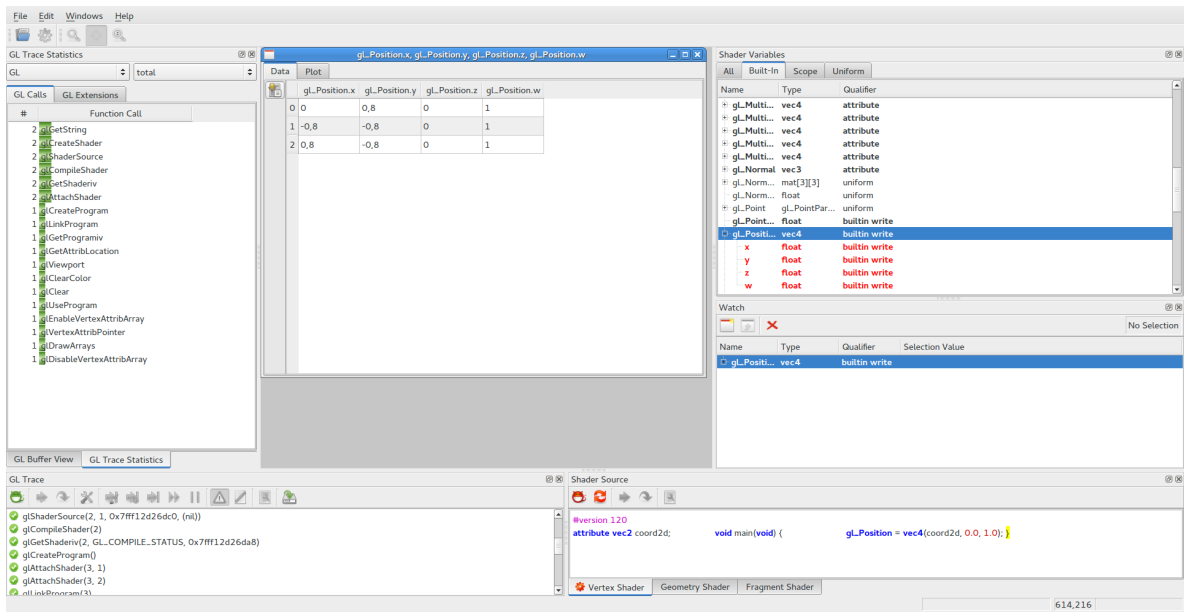Figure 28.: Fragment position viewer.

<div align="right">

E

</div>

VOGL

E.1 INSTALLATION

Here is demonstrated the Windows installation of VOGL, before building VOGL a few libraries need
to be installed, specifically pthreads 2, SDL 2, libjpeg-turbo and QT 5.3 (QT 5.2 will cause errors).
While libjpeg-turbo does not require to be placed in a specific directory it needs to be included later
in one of the VOGL's projects. Once downloaded a folder sibling to vogl is required named external,
so on the same directory where you wish to place the VOGL project, you'll need to:

```
git clone https://github.com/ValveSoftware/vogl.git
mkdir external
mkdir external/windows
mkdir vogl/vogl_build/win32
```

Extract the folder in SDL 2's archive, rename it to SDL and move it to the `external` folder, you'll
need to add either `external/SDL/VisualC/SDL/Win32/Release` or `external/SDL/ VisualC/SDL/`
(depending on which is compiled) to the `PATH` windows variable because of DLL dependencies.

Same goes for pthreads, extract the `Pre-built.2` folder and rename it to `pthreads.2` then
move it to `external/windows`, add `external/windows/pthreads.2/dll/x86` (use `x64`
instead of `x86` for a 64 bit build) DLL's to `PATH` windows variable as well.

With this VOGL is ready to be built, on VOGL's directory (the one created by the git clone use
the following cmake command `cmake-gui -H%cd% -B%cd%\vogl_build\win32` to open
cmake's GUI, then add the following entry:

- Name - Qt5_DIR

- Type - PATH

- Value - `Qt5.3.2\5.3\msvc2013_opengl\lib\cmake\Qt5` depends where you in-
  stalled QT 5.3

Then configure with Visual Studio 12 (to create a Visual Studio 2013 solution). Once generated the
solution will be complete, however it still requires to include `libjpeg-turbo` (pthreads and SDL

are immediately included if you put them on the `external` folder). The only project that requires the library is `vogltrace` so edit its properties `VC++ Directories`, add to `Include Directories` the `libjpeg-turbo/include` and to `Library Directories` the `libjpeg-turbo/lib`. Once it is done VOGL should be good to build.

E.2 USAGE AND CONFIGURATION

E.2.1 *Copying the DLL*

In the official VOGL wiki it mentions mostly about Linux methods which may need some changes in Windows, to make the normal trace method mentioned in the official wiki it is necessary to copy and rename `vogltrace32.dll` to `opengl32.dll` on the target application working directory, this is the Windows equivalent of `LD_PRELOAD`.

As an alternative it is possible to use `vogleditor32.exe` GUI instead in order to create and manage trace files, the editor also requires `vogltrace32.dll` as seen in figure 29. Note that this thesis relies on 32 bit instead of 64, in order to use 64 bit version of VOGL just replace 32 with 64.
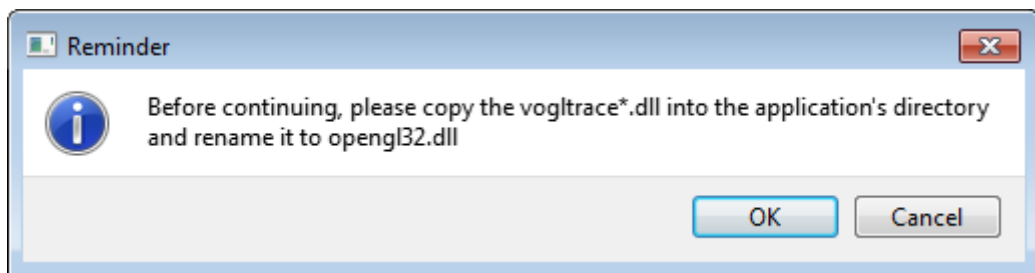

Figure 29.: VOGL editor reminder, occurs when attempting to trace an application.

E.2.2 *VOGL gui*

VOGL GUI carries most VOGL's functionalities from creating traces to replaying, the trace is logged similarly to Apitrace and when a function is selected it is possible to get the snapshot (similar to Apitrace's lookup state capability) and obtain the current function state, this is done by either double clicking or press the snapshot button to the right.

Once the snapshot it is possible to see the framebuffers as shown to the left in figure 30, the OpenGL state in the state tab and more.

The red starting green bar on the top of the editor is actually the OpenGL call bar, it places each OpenGL call within the bar serving as a function timeline.
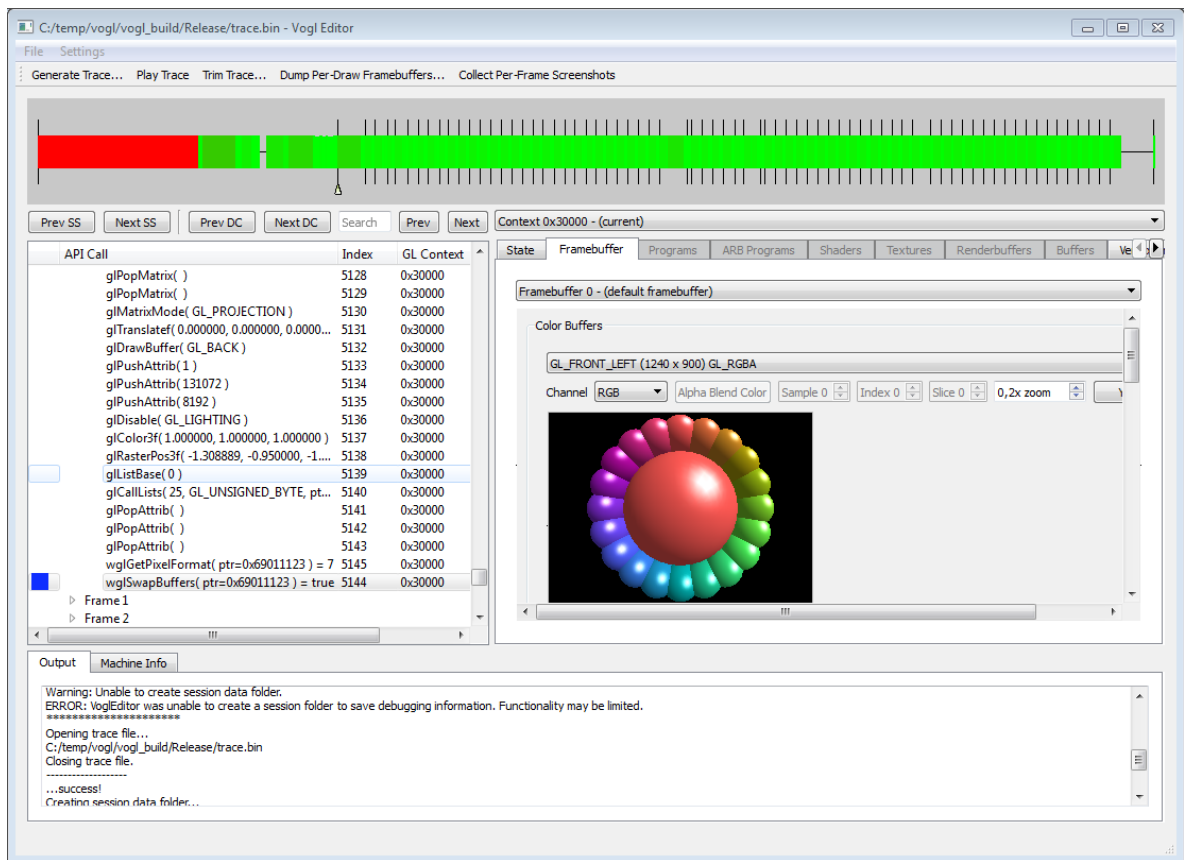
Figure 30.: VOGL editor after snapshot.

E.2.3 *Creating the trace file*

In order to use VOGL it is vital to create the trace file, this is where it begins. The Windows version will differ slightly compared to Linux, to recreate the same steps in Windows it is necessary to:

- Copy the DLL mentioned before (`vogltrace32.dll`);

- `set VOGL_CMD_LINE="--vogl_debug --vogl_tracefile <VOGL_tracefile> --vogl_pause";`

- Execute the application.

As an alternative VOGL editor has the toolbar option `Generate Trace...` which will open a file opening dialog shown in figure 31, it is important to place the `.bin` output trace file, VOGL's traces are in `.bin` format.

Before the application runs it'll show a reminder as shown in figure 29, this means that it is necessary to have the DLL wrapper in place when generating the trace file.
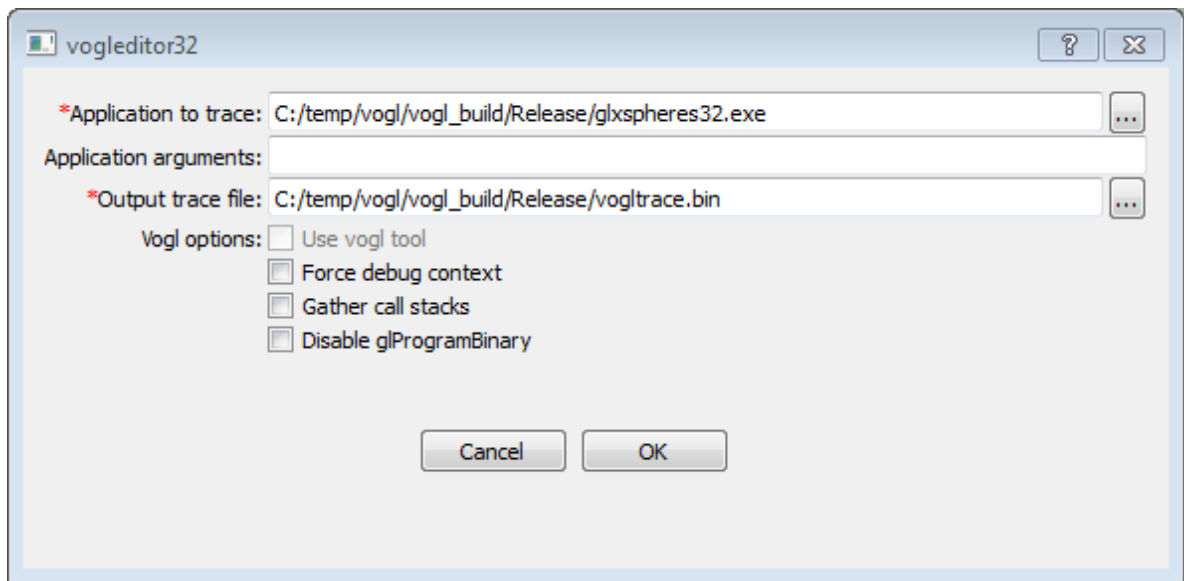
Figure 31.: VOGL editor generate trace.

The trace file is generated as soon the application closes.

E.2.4 *Trimming a trace file*

If the resulting generated trace is very big VOGL will immediately suggest to trim the trace file for faster and easier debugging, otherwise It is possible to trim the trace file by using `Trim Trace...` on the toolbar.

Trim has two attributes, `Trim Frame` which is the frame where the trimming begins and `Trim length` which is the size of the trimming. Trimming is done by replaying the requested frames and recreating the trace.

In the command line it is possible to trim the file with `vogl32 replay <tracefile> --trim_file <tracefile-trim>.bin --trim_len 2 --trim_frame 10`. `--trim_len` and `--trim_frame` are the equivalent of the GUI's trimming attributes.

E.2.5 *Replaying a trace file*

Most of the times VOGL works by replaying just like Apitrace, as long there is a `.bin` trace file it can replay it by choosing the `Play Trace` option in the toolbar.

It is also possible to replay the trace file in the command line with `vogl32 replay <tracefile>`. For a higher performance replay add the `--benchmark` after the `replay` parameter, by doing so it'll also throw in the application's performance.

E.2.6 *Interactive replaying a trace file*

It is possible to replay interactively a trace file with `vogl32 replay <tracefile> -interactive`, this will allow the following commands to the application:

- `<space>` to pause replay;

- `<s>` to initiate slow mode;

- `<left arrow>` or `<right arrow>` to seek, adding `<ctrl>`, `<shift>` or `<alt>` will make seek even further (seek is to jump through draw calls);

- `<J>` or `<T>` to trim the current frame.

E.2.7 *Realtime editing and replaying a trace file*

It is possible to use VOGL to edit and replay a trace file at the same time, in order to do so it is important to dump the trace file into `.json`, in order to do so the following must be done:

```
mkdir dump
vogl32 dump <tracefile> dump/<dumpname>
```

It is important to separate the dump from the rest of the files because it creates one `.json` file per frame, doing so avoids making a mess in the folder. Once dumped extract the `.zip` archive in the `dump` folder and rename it by adding a suffix `.orig`, this way the replay will rely on the extracted files instead.

Now that the files are ready use `vogl32 replay dump/<dumpname>_000000.json --endless` to start the replay, any change made to a `.json` file will affect the corresponding frame, if the change does not occur restart the replay to reset the cache.

E.2.8 *Converting a Apitrace trace file*

In order to convert and Apitrace `.trace` file it is necessary to use Apitrace's `glretrace <.trace> -benchmark` function, it means that to convert you'll have to replay the Apitrace trace file and trace it with VOGL.

- Copy the DLL to the trace file folder;

- `set VOGL_CMD_LINE="--vogl_debug --vogl_tracefile <VOGL_tracefile> --vogl_pause";`

- `glretrace <apitrace_tracefile> --benchmark`

It is possible to dump images per draw call in the editor with `Dump Per-Draw Framebuffers` in the editor's toolbar.

It is also possible to dump frame buffer images per draw call from a trace file with `vogl32 replay <tracef` `–dump_framebuffer_on_draw –dump_framebuffer_on_draw_prefix dump/cap`, this will result in several images whose names follow the following format:

- GLCTR: GL call counter

- FR: Frame #

- DCTR: Frame draw counter

- W, H: Width/height of FBO (or the default framebuffer)

- FBO: Framebuffer object handle #

- FBO attachment

- Texture's internal format

- Texture handle #

The editor can also dump per-frame screenshots with `Collect Per-Frame Screenshots` from the toolbar.

E.2.10 *Get statistics from a trace file*

By using `vogl32 info <tracefile>` it is possible to get the statistics from a trace file, the statistics are outputted in the following format, take note that some parts have been omitted with `(...)` due to the huge size of the statistics:

```
vogl 32-bit Release Built Oct 26 2014 17:26:21
Output statistics about a trace file.
Scanning trace file trace.bin
Total file size: 4,616,033
SOF packet size: 60 bytes
Version: 0x0106
UUID: 0xfd38a90b 0xdb594ba1 0x72689b83 0x75b88e93
First packet offset: 60
Trace pointer size: 4
Trace archive size: 908 offset: 4615125
```

```
Can quickly seek forward: 1
Max frame index: 72
----------------------
Total trace archive files: 3
"compiler_info.json"
"frame_file_offsets"
"machine_info.json"
----------------------
Found trace file EOF packet on swap 72

num non whitelisted funcs: 2
total gl state snapshots: 0
total swaps: 72
total make currents: 6
(...)
total display list calls: 4537
Avg display lists calls per frame: 63.013889
total gl get errors: 0
Avg glGetError calls per frame: 0.000000
total context creates: 0
total context destroys: 0
----------------------
Total calls to glLinkProgram/glLinkProgramARB: 0
Total calls to glProgramBinary: 0
Total unique program handles passed to glUseProgram/
glUseProgramObjectARB: 0
Total unique program pipeline handles passed to glUseProgramStages:
0

----------------------
API histogram: 48
glMaterialfv: Total calls: 8787 23.2%, Avg calls per frame:
122.041667
glPopMatrix: Total calls: 4608 12.1%, Avg calls per frame:
64.000000
glPushMatrix: Total calls: 4608 12.1%, Avg calls per frame:
64.000000
glTranslatef: Total calls: 4536 12.0%, Avg calls per frame:
```

63.000000

(...)

glShadeModel: Total calls: 1 0.0%, Avg calls per frame: 0.013889

wglChoosePixelFormatARB: Total calls: 1 0.0%, Avg calls per frame: 0.013889

wglGetCurrentDC: Total calls: 1 0.0%, Avg calls per frame: 0.013889

wglGetExtensionsStringARB: Total calls: 1 0.0%, Avg calls per frame: 0.013889

wglUseFontBitmapsA: Total calls: 1 0.0%, Avg calls per frame: 0.013889

API Category histogram: 5

"VERSION_1_0": Total calls: 37297 98.3%, Avg calls per frame: 518.013889

"wgl": Total calls: 650 1.7%, Avg calls per frame: 9.027778

"RAD_debugger": Total calls: 3 0.0%, Avg calls per frame: 0.041667

"ARB_extensions_string": Total calls: 1 0.0%, Avg calls per frame: 0.013889

"ARB_pixel_format": Total calls: 1 0.0%, Avg calls per frame: 0.013889

API Version histogram: 2

"1.0": Total calls: 37300 98.3%, Avg calls per frame: 518.055556

"": Total calls: 652 1.7%, Avg calls per frame: 9.055556

Warning:

Warning: Number of non-whitelisted functions actually called: 2

Warning: wglGetCurrentDC

Warning: wglUseFontBitmapsA

Warning:
----

5 warning(s), 0 error(s)

VOGL allows the user to find functions or parameters within the trace file, simply use `vogl32 find`
`<tracefile> -find_func <functionname>` and it'll list all functions found with the same
function name, it is also possible to include wildcards, for example using `glC.*` it'll list all functions
starting with `glC`. The search output is in the following format:

```
----- Function match, frame 70:
{
    "func" : "glClear",
    "thread_id" : "0x16FC",
    "context" : "0x30000",
    "call_counter" : 37033,
    "crc32" : 4040923817,
    "begin_rdtsc" : 14286596747157,
    "end_rdtsc" : 14286596838246,
    "gl_begin_rdtsc" : 14286596755356,
    "gl_end_rdtsc" : 14286596838102,
    "backtrace_hash_index" : 0,
    "rnd_check" : 64982,
    "inv_rnd_check" : 553,
    "params" : {
       "mask" : "0x4100"
    }
}

----- Function match, frame 71:
{
    "func" : "glClear",
    "thread_id" : "0x16FC",
    "context" : "0x30000",
    "call_counter" : 37495,
    "crc32" : 3484429998,
    "begin_rdtsc" : 14286636517653,
    "end_rdtsc" : 14286636606663,
    "gl_begin_rdtsc" : 14286636525780,
    "gl_end_rdtsc" : 14286636606528,
    "backtrace_hash_index" : 0,
    "rnd_check" : 40913,
```

```
    "inv_rnd_check" : 24622,
    "params" : {
        "mask" : "0x4100"
    }
}


Total matches found: 72
0 warning(s), 0 error(s)
```

It is also possible to find parameters with `vogl32 find <tracefile> -find_param   <paramnam`
the following is a search for `GL_FRONT`:

```
----- Parameter 0 match, frame 71:
{
    "func" : "glMaterialfv",
    "thread_id" : "0x16FC",
    "context" : "0x30000",
    "call_counter" : 37927,
    "crc32" : 63488095,
    "begin_rdtsc" : 14286651430725,
    "end_rdtsc" : 14286651441264,
    "gl_begin_rdtsc" : 14286651440157,
    "gl_end_rdtsc" : 14286651441210,
    "backtrace_hash_index" : 0,
    "rnd_check" : 25250,
    "inv_rnd_check" : 40285,
    "params" : {
        "face" : "GL_FRONT",
        "pname" : "GL_AMBIENT",
        "params" : {
            "ptr" : "0x000000000017F800",
            "mem_size" : 16,
            "crc64" : "0x89D4B0CC0713D174",
            "values" : [ 0.77647054195404053, 0.20130716264247894,
0.37385621666908, 264, 0.25 ]
        }
    }
}
```

```
----- Parameter 0 match, frame 71:
{
    "func" : "glMaterialfv",
    "thread_id" : "0x16FC",
    "context" : "0x30000",
    "call_counter" : 37928,
    "crc32" : 3337696999,
    "begin_rdtsc" : 14286651444810,
    "end_rdtsc" : 14286651454359,
    "gl_begin_rdtsc" : 14286651453837,
    "gl_end_rdtsc" : 14286651454296,
    "backtrace_hash_index" : 0,
    "rnd_check" : 59464,
    "inv_rnd_check" : 6071,
    "params" : {
       "face" : "GL_FRONT",
       "pname" : "GL_DIFFUSE",
       "params" : {
          "ptr" : "0x000000000017F800",
          "mem_size" : 16,
          "crc64" : "0x89D4B0CC0713D174",
          "values" : [ 0.77647054195404053, 0.20130716264247894,
0.37385621666908, 264, 0.25 ]
       }
    }
}

Total matches found: 8788
0 warning(s), 0 error(s)
```