

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, November 12, 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 could not accomplish this work.

I would also like to thank all members of Departamento de Informatica for all the knowledge I learned 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's 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 isn't easy to obtain the variables values, the application state and it's hard to detect what causes the errors even with posterior correction. That's why it's necessary to use these types of environments.

OpenGL in particular has many open source debuggers, however their stability and version can be questionable since they are created by an open non-profit community. A study about the usability of Bugle, APItrace, GLIntercept, GLSL(glslDevil) and VOGL is documented, hopefully helping the reader to select the best tool for his needs. Furthermore, it allows the reader to use this document as a user manual.

It was decided to experiment Bugle and GLSL on an Ubuntu environment because it was easier to experiment in a Linux operative system, however APItrace and GLIntercept are experimented on Windows in order to debug a bugged Windows application.

A brief study of the code of the mentioned debuggers is necessary in order to understand the very basics necessary to debug an OpenGL solution. It also helps understand how to update the code with the latest OpenGL version.

Also a study on commercial debuggers from the known companies such as AMD and NVIDIA is made in order to know the current commercial debuggers, the documentation will follow some similiar procedures as the open source debuggers.

Nau 3D engine, developed at Universidade do Minho, is an OpenGL based engine which renders projects written with xml. Adding internal debugger features could help immensely all who wish to work with the engine, allowing the engine developers to understand any possible occurring bug within the engine and also the engine user to find bugs in their own projects.

With the knowledge gathered by studying OpenGL debuggers, several debugging functions for *Nau* 3D engine were implemented. Experiencing usability itself and the many possible faults should ensure an improved product by using the many already existing ideas and methods.

Of course implementing the debugger is not enough, it's pointless to create a new feature without a proper manual regarding it's functionality, that's why it's written in this thesis all that's necessary for a new user to know in order to use *Nau*'s new debugging feature.

RESUMO

É muito raro existir um programa sem bugs, incluindo aplicações 3D com shaders. Os shaders em particular são mais complicados que as aplicações principais, pois esses 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 é de difícil deteção e mesmo posteriormente de correção. Por isso é importante utilizar debuggers especializados para este tipo de ambientes.

O OpenGL em particular tem vários debuggers, no entanto a estabilidade da maioria pode ser questionável. Um estudo sobre experiências da usabilidade de Bugle, APItrace, GLIntercept, GLSL(glsIDevil) e VOGL é documentado o que permite o leitor usar este documento como um manual de utilizador.

É decidido experimentar o Bugle e o GLSL num ambiente Ubuntu devido à facilidade da sua instalação em sistema operativo Unix, no entanto APItrace e GLIntercept é experimentado no Windows para fazer debug de uma aplicação com bugs.

Um estudo breve do código dos debuggers mencionados é necessário para entender as bases necessárias para debug de uma solução OpenGL. Isso também ajuda perceber como actualizar o código com as últimas versões OpenGL.

Também um estudo nos debuggers comerciais actuais de empresas conhecidas como AMD e Nvidia é efectuada de forma a conhecer os debuggers comerciais actuais, o procedimento da documentação terá algumas semelhanças com os debuggers open source.

O motor 3D *Nau* desenvolvido na Universidade do Minho é um motor 3D para OpenGL que faz renderização de projectos escritos em xml, ter um debugger interno nas suas capacidades pode ajudar imensamente a todos que desejam trabalhar com este motor, permitindo aos desenvolvedores do motor perceber possíveis bugs a acontecerem dentro do motor e também aos utilizadores do motor encontrarem bugs dos seus projectos.

Com os resultados dos estudos sobre os debuggers OpenGL, é implementado funcionalidades de debugging para o motor 3D *Nau*, é por isso que a primeira parte da tese, o estado da arte, é um assunto muito importante. Tendo a experiência da própria usabilidade e as possíveis falhas devem garantir um produto melhorado através das várias ideias e métodos já existentes.

É claro implementar somente o debugger não é suficiente, seria inútil criar uma nova capacidade sem um manual sobre as suas funcionalidades, é por isso que está escrito neste artigo tudo que é necessário saber para um novo utilizador para usar a nova capacidade de debugging da *Nau*.

CONTENTS

Contents iii

i	INTRODUCTORY MATERIAL	3
1	INTRODUCTION	4
1.1	Contextualization	4
1.2	Motivation	5
1.3	Document Structure	6
2	STATE OF THE ART	7
2.1	Bugle	7
2.1.1	Filter and Statistics Configuration	8
2.1.2	Graphic User Interface	9
2.1.3	Inner Workings	11
2.1.4	Maintenance	11
2.2	APITrace	11
2.2.1	Basic Functionalities	12
2.2.2	Log reading GUI	12
2.2.3	Real problem solving example	14
2.2.4	How it works from inside	15
2.2.4.1	Creating a trace	15
2.2.5	Maintenance	16
2.3	GLIntercept	17
2.3.1	Logging	17
2.3.2	Plugin Usage	19
2.3.3	OpenGL32.dll wrapper	19
2.3.4	Maintenance	19
2.3.4.1	Plugins	21
2.3.4.1.1	GLFreeCam	21
2.3.4.1.2	Plugin Creation	22
2.4	GLSLDevil/GLSL-Debugger	23
2.4.1	Graphic User Interface	23
2.4.2	How does GLSL logs and debugs	25
2.4.2.1	Common Debugging	25
2.4.2.2	Shader Debugging	25
2.4.3	Maintenance	26

2.5	VOGL	26
2.5.1	Functions and GUI	27
2.5.2	Maintenance	28
2.6	CodeXL	28
2.6.1	Debug Mode	29
2.6.2	Profiling Mode	29
2.6.3	CPU Time Based Profile	29
2.6.4	GPU Application Trace	29
2.7	Nsight	29
2.7.1	Graphics Debugging	30
2.7.2	Performance Analysis	32
3	DEBUGGER COMPARISONS	34
3.1	Open Source Applications	34
3.1.1	Comparison table	34
3.1.2	Feature table	35
3.2	Commercial/Freeware Applications	37
3.3	Conclusions regarding State of Art	37
ii	INCORPORATING THE DEBUGGER IN NAU	39
4	USING AN EXISTING DEBUGGER	40
4.1	Changes on GLIntercept	40
4.2	Changes on Nau	42
4.3	Changes on composer	43
5	HOW TO USE NAU'S DEBUGGER	44
5.1	functionlog	45
5.2	logperframe	46
5.3	errorchecking	46
5.4	imagelog	46
5.5	shaderlog	47
5.6	displaylistlog	47
5.7	framelog	48
5.8	timerlog	49
5.9	plugins	49
5.10	How to get OpenGL state	50
5.11	How to use the composer	51
6	CONCLUSIONS AND FUTURE WORK	55
6.1	Conclusions	55
6.2	Prospect for future work	56

iii	APPENDICES	61
A	INSTALLATION	62
A.1	Bugle	62
A.2	APITrace	64
A.3	GLIntercept	64
A.4	GLSLDevil/GLSL-Debugger	65
A.5	VOGL	65
B	USE / CONFIGURATION	67
B.1	Bugle	67
B.1.1	Statistics configuration	67
B.1.2	Filter Configuration	68
B.1.2.1	Statistics filterset	69
B.1.2.2	Trace and Log filterset	71
B.1.2.3	Error checking filtersets	72
B.1.2.4	Context attributes and extension override filtersets	73
B.1.2.5	Showextensions filterset	75
B.1.2.6	KHR_Debug filterset	75
B.1.2.7	Compilable C source filterset	76
B.1.2.8	Screenshot filterset	77
B.1.2.9	eps filterset	78
B.1.2.10	frontbuffer filterset	78
B.1.2.11	Wireframe filterset	79
B.1.3	Graphic User Interface	79
B.2	APITrace	81
B.2.1	Tracing	81
B.2.2	Retracing	81
B.2.3	Output replay to video	82
B.2.4	Trimming trace file	82
B.2.5	Profiling trace	82
B.2.6	Apitrace's GUI	84
B.3	GLIntercept	86
B.3.1	Tracing	86
B.3.2	Frame Logging	88
B.3.3	Shader Editor	88
B.3.4	ARB_debug_output Logging	89
B.3.5	Extension override	90
B.3.6	Function statistics	91
B.4	GLSLDevil/GLSL-Debugger	93

B.4.1	GL Trace	95
B.4.2	Shader	96
B.4.3	GL Trace Statistics	97
B.4.4	GL Buffer View	97
B.4.5	Shader Variables and Watch	98
B.5	VOGL	100
B.5.1	Copying the DLL	100
B.5.2	VOGL gui	100
B.5.3	Creating the trace file	101
B.5.4	Trimming a trace file	102
B.5.5	Replaying a trace file	102
B.5.6	Interactive replaying a trace file	103
B.5.7	Realtime editing and replaying a trace file	103
B.5.8	Converting a APITrace trace file	103
B.5.9	Dump images from a trace file	104
B.5.10	Get statistics from a trace file	104
B.5.11	Finding in a trace file	107

LIST OF FIGURES

Figure 1	Bugle showstats example	9
Figure 2	Bugle gdb state tab.	10
Figure 3	Bugle gdb shader error encountered.	10
Figure 4	Apitrace's qapitrace GUI	13
Figure 5	Apitrace looking up state	14
Figure 6	Debug correction	15
Figure 7	APITrace inner workings diagram	16
Figure 8	GLIntercept xml output in internet explorer.	18
Figure 9	GLSL Trace Statistics and Vertex Shader	24
Figure 10	GLSL fragment color viewer.	24
Figure 11	VOGL editor after snapshot.	27
Figure 12	Nsight HUD while application is running.	30
Figure 13	Nsight HUD while application is paused.	31
Figure 14	Nsight HUD while application is paused with wireframe on.	31
Figure 15	Composer's Pass controller.	51
Figure 16	Nau's GLIntercept log viewer.	52
Figure 17	Nau program information.	52
Figure 18	Nau buffer information.	53
Figure 19	Nau VAO information.	53
Figure 20	Nau State information.	54
Figure 21	Bugle showstats example	70
Figure 22	Bugle eps screenshot	78
Figure 23	Bugle gdb state tab.	79
Figure 24	Bugle buffers.	80
Figure 25	Bugle gdb shader error encountered.	80
Figure 26	Bugle gdb breakpoint.	81
Figure 27	Apitrace's qapitrace GUI	85
Figure 28	Apitrace looking up state	86
Figure 29	GLIntercept xml log	88
Figure 30	Extension override results	91
Figure 31	GLSL open application dialog	93
Figure 32	GLSL Buffer View and Fragment shader	94
Figure 33	GLSL Trace Statistics and Vertex Shader	94

Figure 34	Fragment shader per-fragment options.	97
Figure 35	Fragment coordinates viewer.	98
Figure 36	Fragment color viewer.	99
Figure 37	Fragment position viewer.	99
Figure 38	VOGL Reminder	100
Figure 39	VOGL editor after snapshot.	101
Figure 40	VOGL editor generate trace.	102

LIST OF TABLES

Table 1	Open Source Applications Pros and Cons table	35
Table 2	Open Source Applications Feature table	36
Table 3	Commercial/Freeware Applications Pros and Cons table	37

Part I

INTRODUCTORY MATERIAL

INTRODUCTION

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

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, when using uniform blocks NVIDIA accepts the instance block name, while AMD does not. 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. 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 [?]. 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 [NV1a], and AMD has CodeXL [AMD13]. Both debuggers can work integrated with Visual Studio. While the list of features is impressive, including shader code tracing, and GPU memory inspection, 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 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. 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.

The following open source debuggers, mentioned in the OpenGL wiki [Ope12], shall be covered in this work: Bugle[MB07b], Apitrace[FJea13], GLIntercept[DS13], 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 an 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 [NV1b]. *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 experimentation of the mentioned debuggers with subsections dedicated to testing and analysis. The open source debuggers will also feature a sub section dealing with upgradeability regarding OpenGL versions.

Afterwards 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 good 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 analysis and experimentation is done the creation of the promised debugger for *Nau* begins, the created debugger itself will use one of the five candidate open source debuggers as it's base because starting the from the scratch would be far too time consuming and inefficient.

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 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.

Nau's new debugging features will also require documentation of it's usability, this thesis will serve as a manual for all new debugging features implemented for *Nau*'s new features.

For all interested in the installation method of all five open source debuggers these are documented in the appendices section of this thesis.

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 it's relation with Valve. The list of the chosen debuggers is:

- Bugle - Open source
- APITrace - Open source
- GLIntercept - Open source
- glslDevil - Open source
- VOGL - Open source
- CodeXL - Commercial
- Nsight - Commercial

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 be avoided unless it is also mentioned alongside OpenGL.

In the following sections each of the debuggers will be tested and analyzed. To conclude this section a comparison is performed.

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 uses a successfully compiled Bugle in a Ubuntu 13.10 gnome operative system, this section shall report the experiments with Bugle.

Bugle's configuration is based on filters (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 a resources such as viewing textures, buffers or shaders.

The main list of features is as follows:

- Create filter chains;
- Log all GL calls (sec. [B.1.2.2](#));
- Configure and show meta-data statistics (sec. [??](#));
- View OpenGL version and extensions used on the application;
- Identify API errors;
- Screenshot or video capture;
- Output a compilable C file ([B.1.2.7](#));
- Alternate to `GL_LINE` to visualize the object's triangulation (sec. [B.1.2.11](#)).

Bugle uses two additional configuration files (also mentioned in the installation in the appendice [??](#)), `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`.

2.1.1 *Filter and Statistics Configuration*

Bugle is a debugger which works by relying with *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 form of information, so this is a *filter* used to extract information. While `stats_basic` may be useless by itself it is absolutely necessary for `filterset showstats` to show FPS and any other information relying on `stats_basic`.

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

The output a statistic related filters can show depends on the configuration of statistics in the `statistics` file, this gives even more freedom for the user to decide what to profile and what kind of debugging operation to perform. This file is where all stat related calculations are defined, for example frames per second is defined as: $framespersecond = d("frames")/d("seconds")$ with precision 1 and label "fps".

This means that whenever `showstats` uses `show frames per second` it'll output `fps` as shown in figure 1.

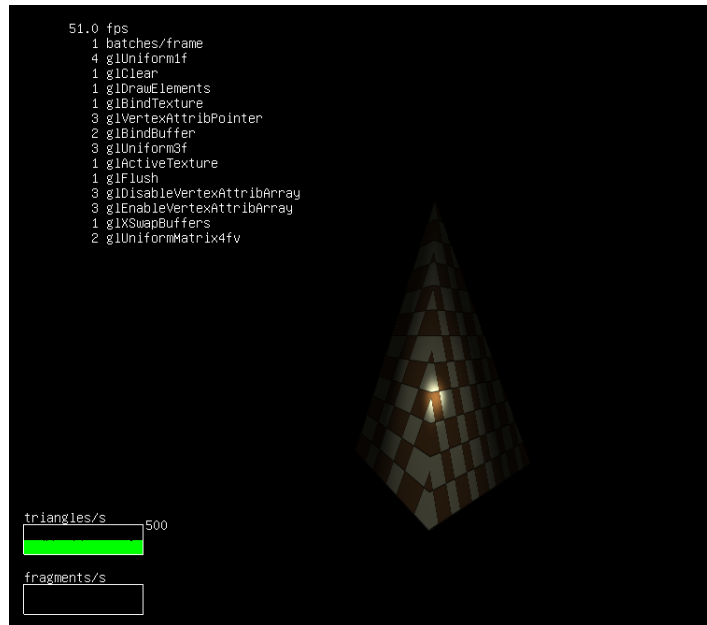


Figure 1.: Bugle showstats example.

2.1.2 Graphic User Interface

Bugle has its own GUI, it makes manual use of bugle through command line redundant. The GUI allows the user to use additional debugging beside *filter chains*, such methods include:

- Viewing the GL state, it's possible to check which states changed by checking show changed states, this is shown in figure 2;
- Step by step debugging, breakpoints on certain GL function are also possible;
- Viewing resources such as shaders and texture;
- Viewing buffers and framebuffer, these resources are update 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.

This means it's possible to pause the application and check it's state, then pause again on a later stage to see which states changed. Using the checkboxes greatly helps to search for such changes.

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 may help evaluate whether there's an anomaly and where it may have occurred by slowly tracing by step.

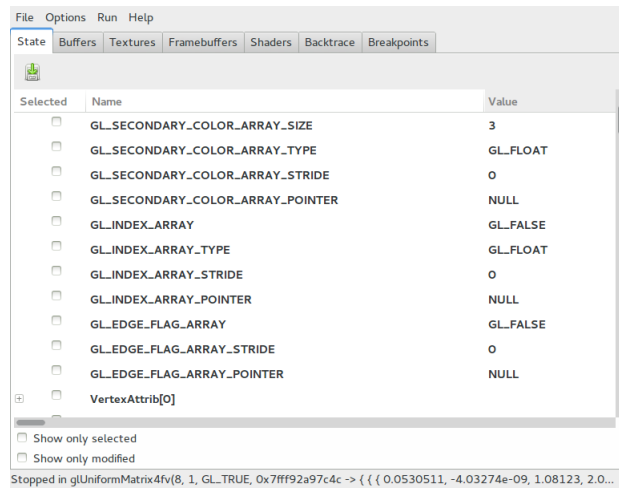


Figure 2.: Bugle gldb state tab.

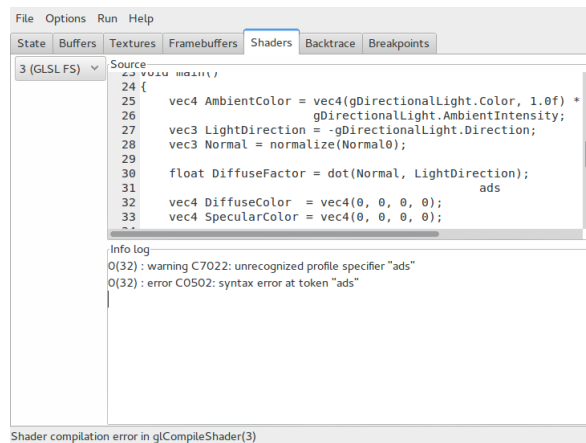


Figure 3.: Bugle gldb shader error encountered.

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's possible to link *.so files before loading applications. This is exactly what Bugle does. The gdb 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 the `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` 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 current library files including the ones that come within bugle's khronos-api folder, Bugle generate's it's files using the OpenGL libraries using the khronos-api xml files, these can be located in with <https://cvs.khronos.org/svn/repos/ogl/> So when it comes to GL version Bugle is very simple to update. Should the repository die or become out of date it's recommended to create a script to generate the xml files based on the official khronos header files.

It should be noted that the files requiring replacement are third party and not shown within the official repository, however it should be visible after being downloaded.

Once the new libraries are updated simply build with Scons as mentioned on the *Installation* section at the appendices.

2.2 APITRACE

APITrace [FJea13] is a command line tool for debugging, it has cross-platform compatibility allowing debugging for Linux, Windows and even Android.

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

It's trace file has is a binary file readable only by apitrace which may be a downside considering it forces the user to use only it's tools. However it can also be edited by apitrace allowing the user to resize the file or even change some input.

In short this debugger offers:

- Capability to create an OpenGL, OpenGL ES, Direct 3D and DirecDraw 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.

2.2.1 *Basic Functionalities*

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, every other APITrace debugging must be performed on the generated trace file.

Sometimes traces can result in big files, thus trimming can help reduce the file size considerably. A example why trimming is important is the fact that some files over 100 frames can take a considerable amount of time to load, trimming can reduce that load.

Replaying is APITrace's strongest feature, many APITrace operations resort to replaying. Since replaying is independent from the target application it's possible to store many different trace files for replaying.

2.2.2 *Log reading GUI*

APItrace relies on `qapitrace` GUI to read trace files, simply use the `.trace` file on the executable and it'll show the whole log arranged by frame as shown in 4.

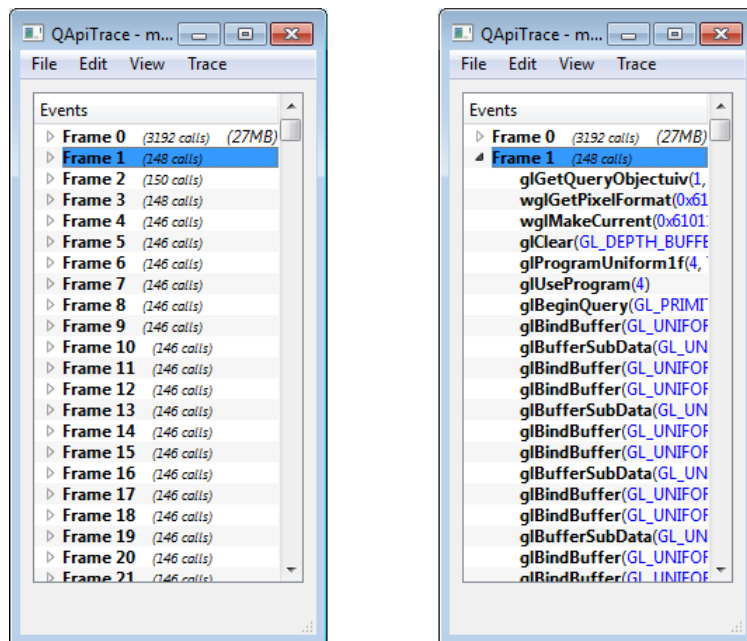


Figure 4.: On the left shows how Aptrace splits the log per each frame, on the right it has a frame node expanded showing it's function log.

Not only it's possible to manually edit the GL calls for the next replay it's also possible to lookup the state on the current function shown in 5. 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 at the place of an anomaly.

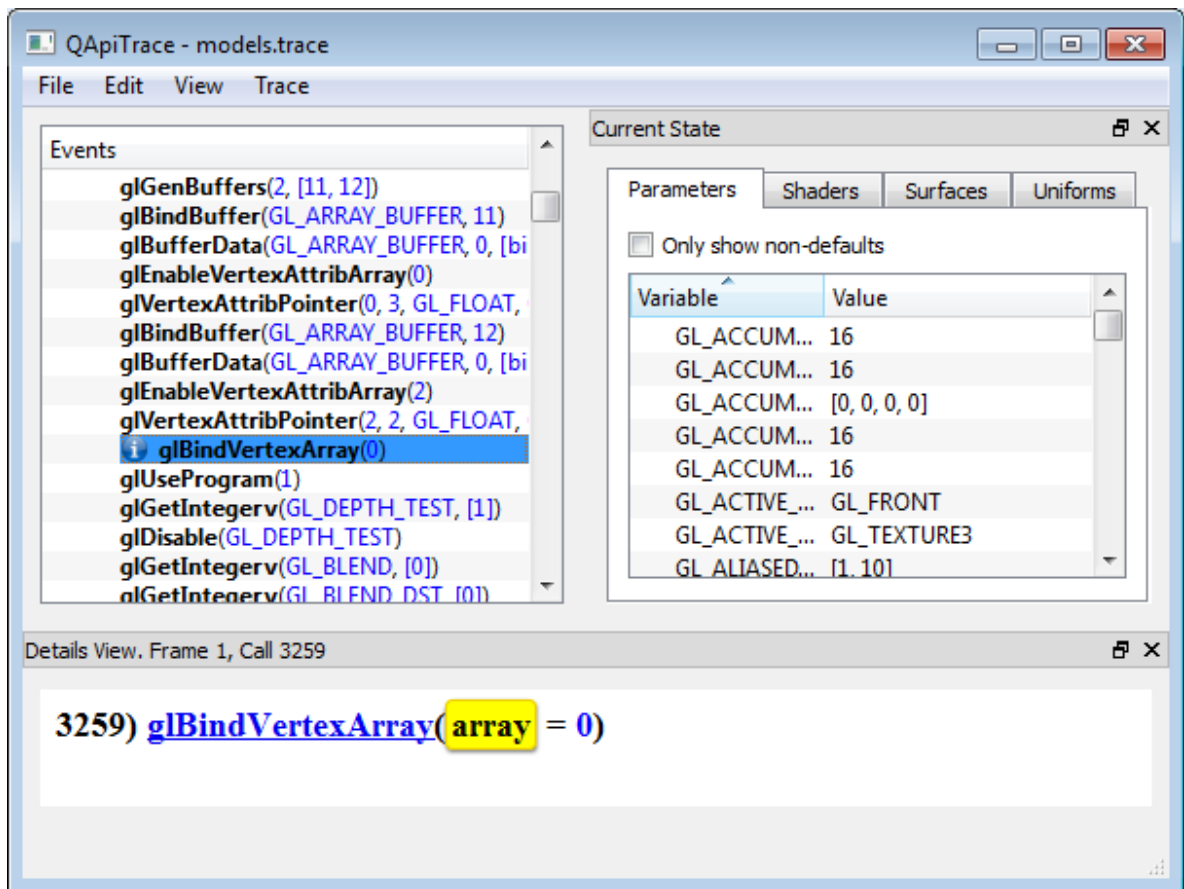


Figure 5.: Looking up the state before changing the program, it allows to see the current program's parameters, shaders, buffers(surfaces tab) and uniforms.

2.2.3 Real problem solving example

The example used in most debugging is a bugged application, for some reason the light intensity is always returning a value very close to 0 thus making the object always on night.

Using qapitrace GUI and using "Lookup state" before switching to program 1 (text rendering shaders) it can check the uniforms for the bugged shader.

The analysis of the uniforms concludes that it wasn't sending the proper matrices to the shader thus messing the calculations, problems include modelMatrix, viewMatrix and viewModelMatrix were zero matrices and the normalViewMatrix's determinant was 0. Those are clearly bugs that should not occur in a shader and the error can be determined as outside the shader but within the application's uniform management. The results can be compared at the figure 6.



Figure 6.: On the left is the model before correction, on the right is the model after correction.

As seen in image what happens if the bugged matrices are avoided by considering them as identity matrices (which is more cheating than a solution). It also corrects the `viewMatrix` by changing `normalViewModelMatrix` into `normalMatrix`.

2.2.4 *How it works from inside*

O APITrace for windows works using the usual [DLL injection](#) 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.

2.2.4.1 *Creating a trace*

A study of Apitrace's code concludes that the tracing works according to the following diagram in [figure 7](#):

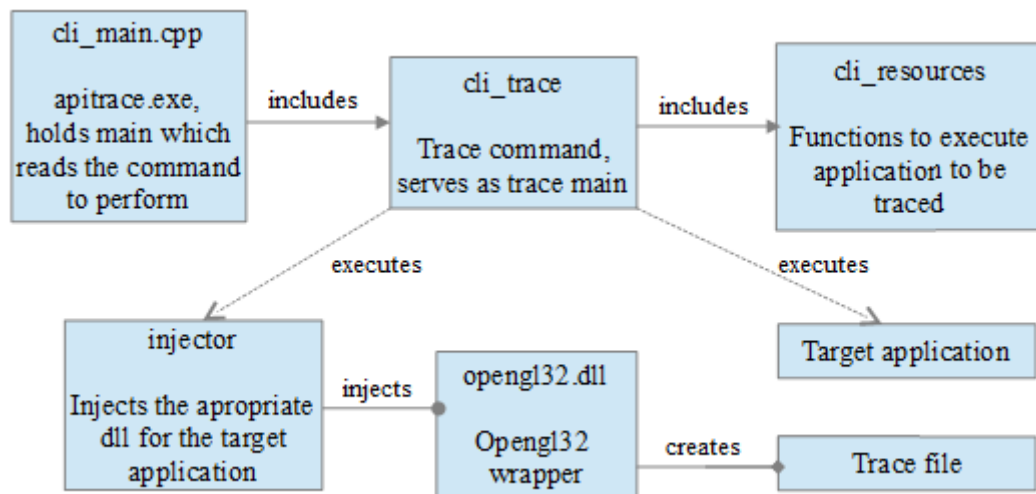


Figure 7.: APITrace inner workings diagram - trace action

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.

2.2.5 Maintenance

Should there be an update for opengl32.dll it will be important to know where to change the source code. In Apitrace case most code is packed into python scripts, by searching for the function OpenGL 4.3 with it's header it can be found in two files:

- `specs\glapi.py`
- `thirdparty\khronos\GL\glext.h`

The file in khronos folder isn't necessary to be manually updated, all it needs is to be updated with the most recent files published by khronos. The `glapi.py` is a script generated by `glspec.py` in the same folder, this will generate based on the khronos files, manual editing is obligatory.

This is done by using `make -b` command on the `specs\scripts` folder, this command will download all the current OpenGL specs (download the current OpenGL functions), it'll also generate `glapi.py`, `glxapi.py`, `wglapi.py`, `glparams.py`, `wglenum.py` and `eglenum.py`. Most of these files contains functions to replace, for instance `glapi.py` contains all and only `glapi.addFunctions` parameters, the pattern by comparing both files beforehand is noticeable. Basically all the mentioned `*.py` functions may need to be updated, there was however an inconsis-

tency between `glxapi.py` because the one being used uses `Function(*)` but the newly generated ones uses `GLFunction(*)`, perhaps an update to the application is necessary.

Even with a diff tool(a tool to merge or change differences between to files such as `vimdiff`) it may take hours of precisely replacing each function, they do not have an automatic method to replace the necessary functions, attempting to hastily replace will cause errors during compilation.

As for the rest all is needed is to replace the khronos header files with the most recent header files. It should be noted that the current OpenGL 4 specs do not require much updating since they are mostly `arb` extensions making it a "4.*"¹ spec.

2.3 GLINTERCEPT

GLIntercept [DS13] is an OpenGL debugger for windows, it has an official windows installer which contains a debugger up to OpenGL 4.3. It is the debugger most focused on windows among the five debuggers listed in this document.

GLIntercept is very manual when it comes to configuration and utilization requiring the user to manually copy and paste files and edit manually with text editors.

This debugger is unique due to the fact that it allows the inclusion of plugin libraries, the source code has it's own plugin solution to help creation of plugins. By using this capability it's possible for other programmers to extend the debugger.

The debugger itself is capable of tracing and xml logging, it can also capture the used frames and resources.

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.

Take note that SciTE (Scintilla based Text Editor) is a third party text editor created outside of GLIntercept, thus it's source belongs to a different project.

2.3.1 Logging

GLIntercept most basic feature is logging, in fact mostly without pluginsthat's what GLIntercept does, the basic log GLIntercept offers is the following:

¹ The spec should work for almost any OpenGL 4 version

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

It's possible to set GLIntercept to log per frame or in xml format. When it's set to log per frame it separates the log into multiple file according to the captured frame, the captured log looks like a normal log but it's limited to one frame. In case xml format is used, it comes with a viewer which shows the xml according to the figure 8.

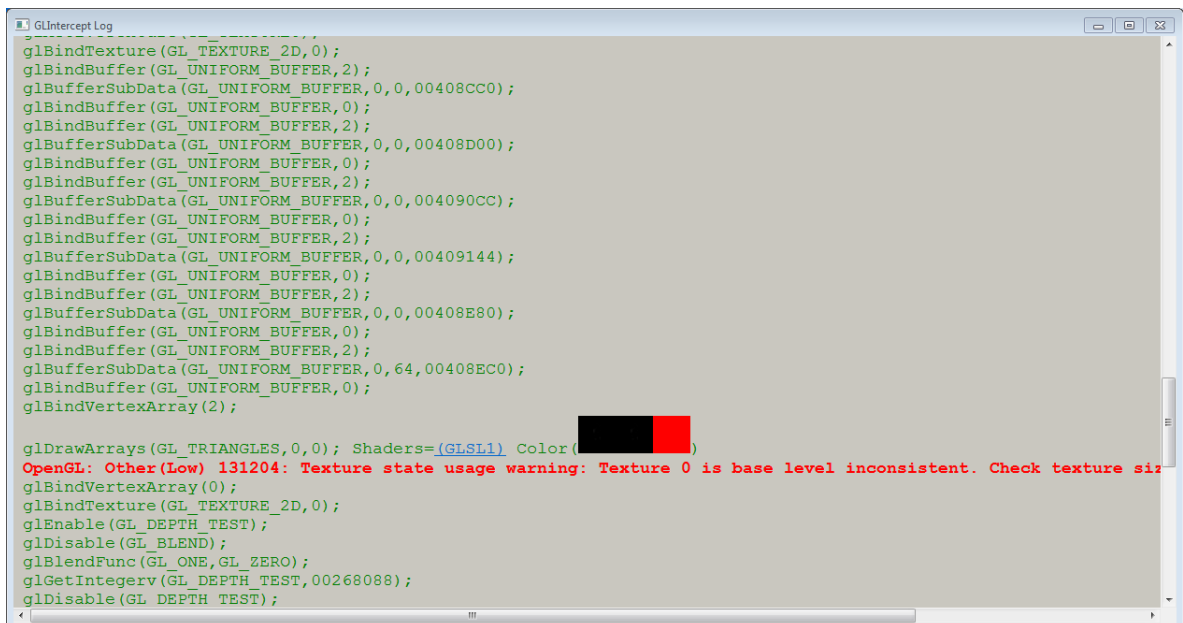


Figure 8.: GLIntercept xml output in internet explorer.

It's also possible to enable shader and texture capturing during logging, in such case it'll create separate folders with the mentioned resources, in GLIntercept capturing textures is referred as image logging. In case of single frame logging if shaders or textures are captured they will be put in the respective frame's folder, if xml logging is active it'll also capture the framebuffer.

2.3.2 Plugin Usage

Any other addition to GLIntercept is made by extending with plugins, plugins are essentially adding another wrapper within GLIntercept's wrapper. For instance the statistics plugin is made by counting every function that passes through the wrapper and outputting the results at the end of the application.

It's also possible to override parameters within the application with plugins, for example the *Free Camera* plugin changes the application's camera by swapping the related matrices with a new calculated matrices.

2.3.3 OpenGL32.dll wrapper

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. GLIntercept requires header configuration files with the function signatures and constants to provide this information. Currently, GLIntercept is up to date with OpenGL 4.4. In case the function information is not provided it'll replace the parameters information with "???".

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 for downloading 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. Simply put the headers in the `3rdParty\HeaderGen` folder and execute `XMLGenGLI.py` script, it should create the headers automatically, tuning is still required.

In an attempt to transform Glew's header into an appropriate GLIntercept's header it took several steps of careful tuning, so it's recommended to create a script which does the job, 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's 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 `glxext.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`:

```
//glxext.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.

2.3.4.1 Plugins

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 be 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.

This section will use the *GLFreeCam* to give an brief example because it overrides the application with new input and gives visible results.

2.3.4.1.1 GLFreeCam

GLFreeCam is a plugin that uses the `MiscUtils.<cpp/h>` and `InputUtils.<cpp/h>`, this plugin will allow the user to override the camera moving it beyond what it was programmed for.

All DLL's have rendering stages, these rendering stages are a new layer of functions which will trigger depending on the plugin. In *GLFreeCam* after the final rendering stages it'll calculate the elapsed time and the pressed keys to create a transformation matrix.

`GLFreeCam.cpp` will use `OpenGLFreeCamera.<cpp/h>` to alter the view frustum before rendering, it will do so by saving a global variable `transform_point` that will always be loaded as the current transformation matrix. This technique will not work for all applications, during tests it failed to move all models according to the new camera.

It can be concluded with this that in order to build a new plugin all it needs is to add additional functions to each stage, some of the overrides such as the inputs can be done with the already existing headers.

2.3.4.1.2 *Plugin Creation*

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

The already existing test plugin is a perfect base for a new plugin, it's 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's 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 destructor 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 a memory location, however if it points to a part of memory it will only become valid on `GLFunctionPost`, sometimes it's 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's 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's a vector of unsigned integers (`vector<uint>`), it's possible to use the configuration to read the desired set of keys.

In `CustomGetUniforms` it's 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's possible to add additional parameters in the `GLIntercept`'s config file to change the keys necessary to print the uniforms.

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's 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 it's statistics and resuming until it hits a certain function.

As practical it's 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. It's 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.

2.4.1 Graphic User Interface

`GLSL`'s main feature is it's GUI, it's capable of doing logging and showing it's log on 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 very 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.

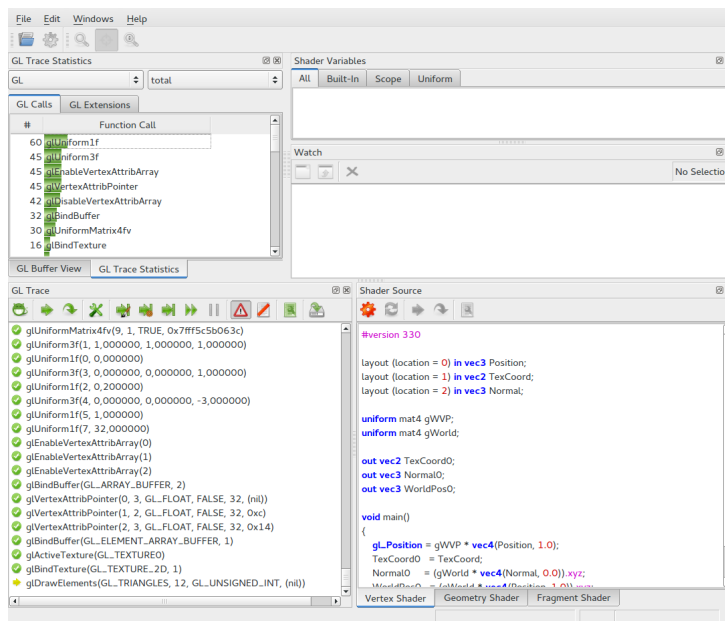


Figure 9.: GLSL it's featuring trace statistics and the vertex shader.

It's also possible to simulate debug of shaders whenever a draw call is the next function to be called, in order to do so the trace must be paused. Currently only GL 2 shaders can be reliably debugged while mesa limits only to GL 3. 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, since the frag color is the result of the frag shader, it's possible to see the fragment shader's output.

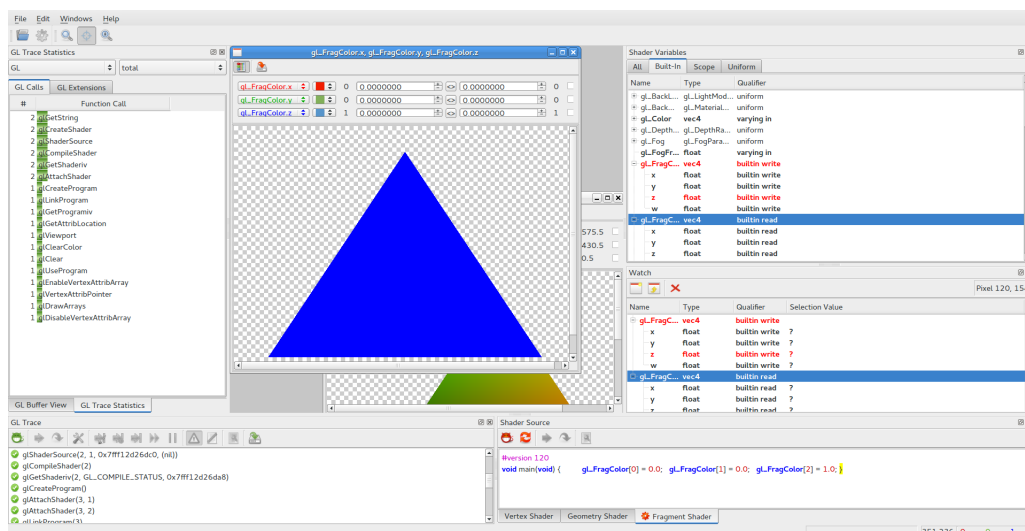


Figure 10.: GLSL fragment color viewer.

2.4.2 How does GLSL logs and debugs

2.4.2.1 Common Debugging

In the Windows version of GLSL a `glslddebug.dll` is created, according to the existing files and the functions within it is certainly the OpenGL wrapper, the functions inside are separated between hooked and orig, thus the difference between the hooked function (caught) and orig (original).

This means that when debugging all OpenGL's functions are hooked by the debugger then executed by the original, logging occurs thanks to the hooking.

The hooking functions is also part of the Unix build, considering how imperative is the existence of `libGL.so`, that must be where the hooking affects.

Whenever GLSL debugs an application it's possible to save a log file as well, this GLSL log file has the format according to the following example:

```
| glXSwapBuffers(0x1d11d50, 46137346)
| glXMakeContextCurrent(0x1d11d50, 46137346, 46137346, 0x1d2c7e8)
| glXMakeContextCurrent(0x1d11d50, 46137346, 46137346, 0x1d2c7e8)
| glXMakeContextCurrent(0x1d11d50, 46137346, 46137346, 0x1d2c7e8)
| glUseProgram(3)
W! OpenGL error GL_INVALID_OPERATION detected
| glUniform1f(0, 0, 102166)
| glActiveTexture(GL_TEXTURE0)
| glBindTexture(GL_TEXTURE_2D, 1)
| glUniform1i(1, 0)
| glActiveTexture(GL_TEXTURE1)
```

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

2.4.2.2 Shader Debugging

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.

Once it's loaded all it needs is to execute the functions step by step. As seen in `libglslddebug.c` there is a function that called `shaderStep`, there is no doubt that it'll proceed shader's debugging. Unfortunately the whole process still needs development since shader debugging is unreliably limited to mesa headers and unstable.

2.4.3 Maintenance

GLSL update's 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 `glexth.h` in <http://www.khronos.org/registry/> to substitute the current GLSL `glexth` file, other files from the same repository are `wglexth.h` and `glxext.h`. `WinGDI.h` actually belongs to Microsoft and it can be obtained from <http://www.csee.umbc.edu/~squire/download/WinGDI.h>. Finally `gl.h` and `glx.h` belongs to mesa repository, it can actually be found in <https://github.com/freedesktop/mesa-demos> or simply search for it in the computer if it has mesa's header files. Even so there is a good chance that updating will cause issues with enumerates.

In Windows build it'll generate a `trampolines.h` that connects with the original library, this library is necessary for the debug library. It seems such file is unnecessary in an Unix build.

In Unix build it's best not to forget to update the `libGL.so` mentioned in the *Installation* section at the appendices, should the drivers be updated, the debug will fail if built with a different lib than the one being currently used by the computer.

It should be noted that current shader debugging is dependent on the mentioned mesa header files, since current mesa does not support OpenGL 4 versions shader debugger won't support either.

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 allowing to be a good alternate to APITrace, just like APITrace cmake and Qt is required.

As a Valve debugger this debugger can capture steam games, pay in mind that currently this debugger is only in its 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
- 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

VOGL just like GLIntercept needs to copy it's own DLL wrapper to the target application, once copied it can be debugged via command line like APITrace or using a GUI like GLSL. Just like APITrace, VOGL can view it's log's with it's own GUI which is organized very similarly. It's also capable of looking up the state with replaying just like APITrace, this can be seen in figure 11.

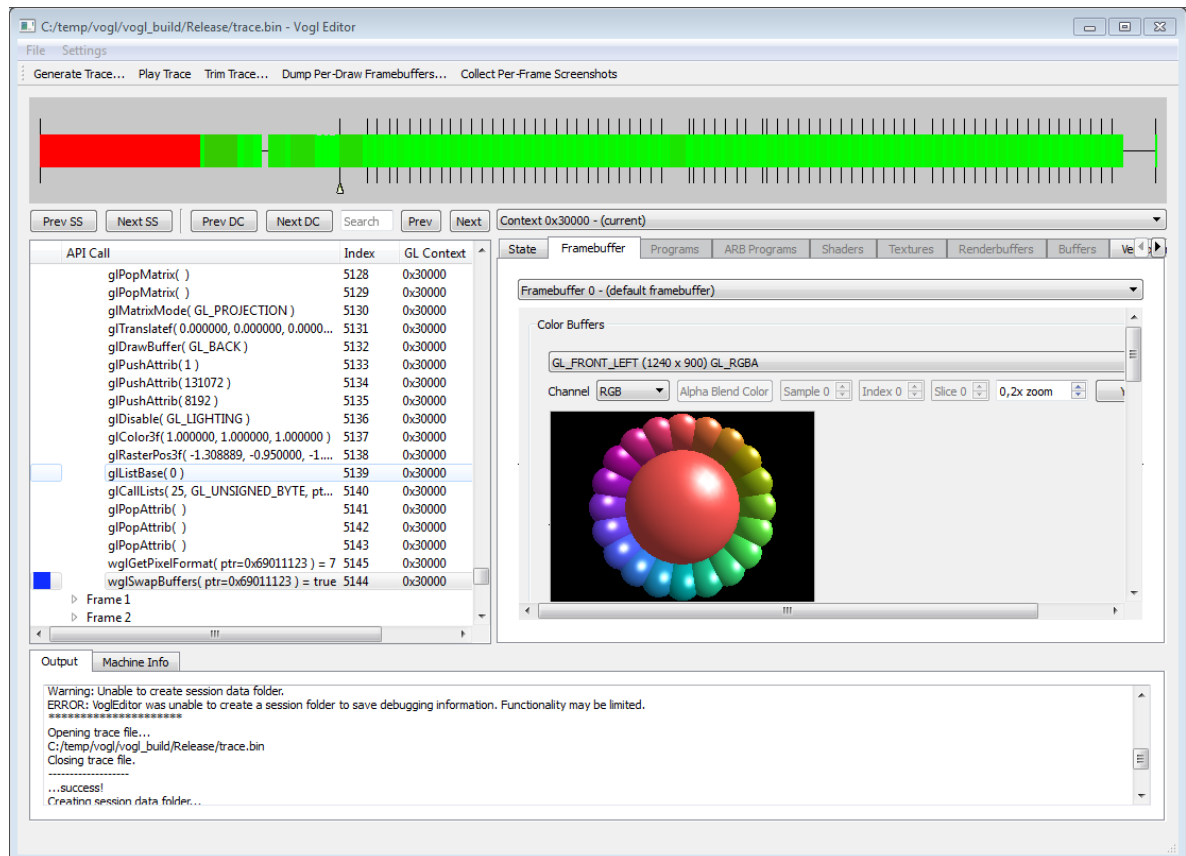


Figure 11.: VOGl editor after snapshot.

VOGL store trace files in .bin format, it's also possible to convert them to .json format, in .json format the frames are separated per .json file. When replaying a .json trace it's possible to edit the file during replay, the next time the frame is replayed the changes should be in effect unless it's cached (which requires restarting the replay).

The replay can actually start with interactive mode which allows the user to pause the replay mid-way, what it actually does is to stop at a certain frame. For example if in order to jump the frame backwards it'll replay from the beginning until it reaches the previous frame. Interactive mode is also slow.

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.

Unlike GLSL, VOGL cannot simulate shader debugging, this feature is not in the intended feature list either.

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, these `.inc` files are vital for VOGL's build and it will not succeed without them.

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 extend 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;
- Remote application debugging.

CodeXL has it's 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 breaks (pauses). When viewing the buffers it shows what is drawn on the buffer, same for loaded textures. VBO's are have more data associated from "Image view" and "Data view", data view shows the arrays in a table with it's own display options.

2.6.2 *Profiling Mode*

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

2.6.3 *CPU Time Based Profile*

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

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

2.6.4 *GPU Application Trace*

GPU application trace will give use 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. This will help the user to view when did the call occurred.

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 it's IDE. In this document it'll focus on 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 properly in the laptop, in order to use Nsight to it's fullest it should be run in a desktop.

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 runs overrides the application's output. Essentially it means that part of it's GUI runs on the debugged application itself and can also be used standalone apart from the IDE.

It's 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>`.

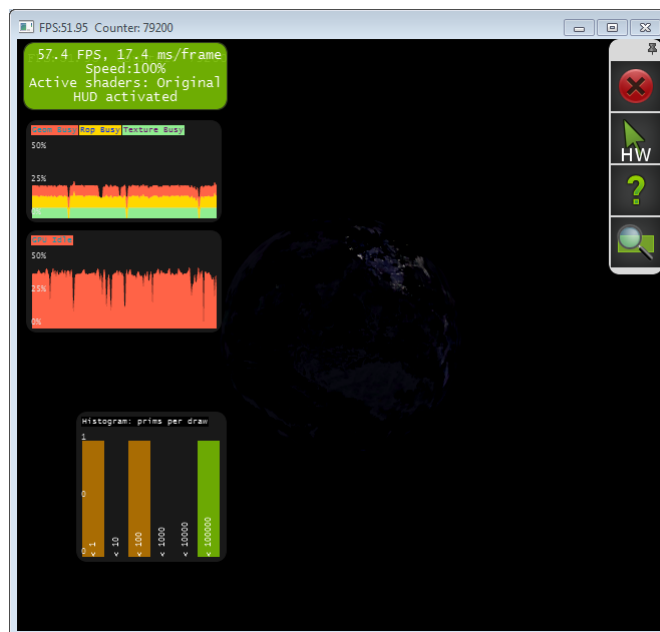


Figure 12.: Nsight HUD while application is running.

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.

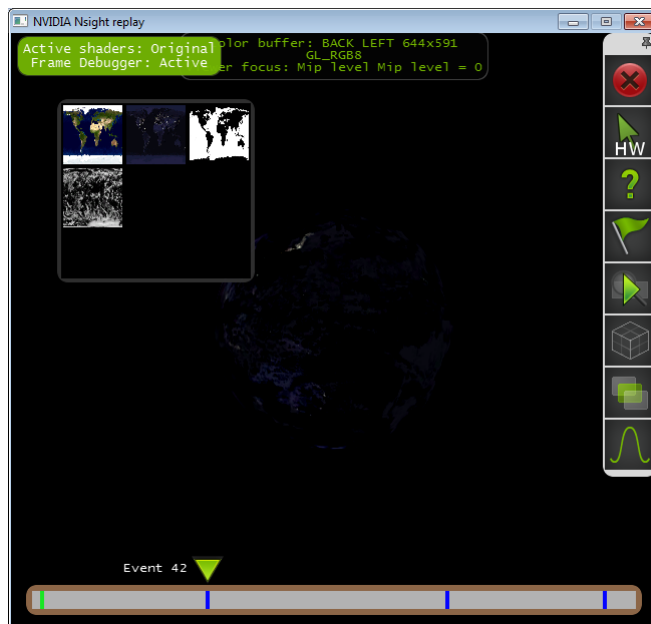


Figure 13.: Nsight HUD while application is paused.

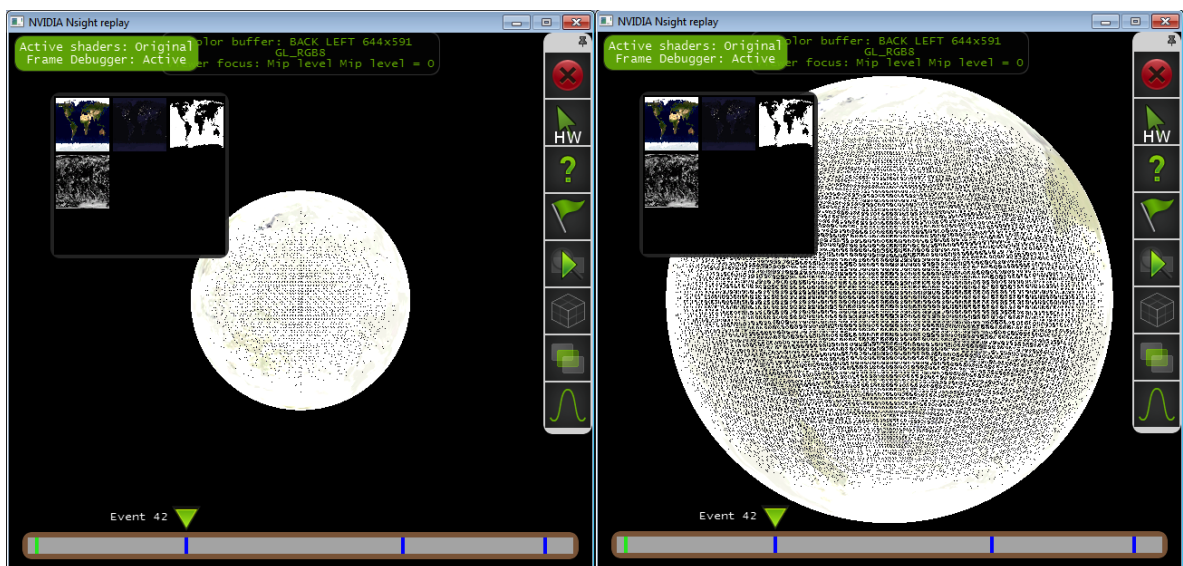


Figure 14.: Nsight HUD while application is paused with wireframe on.

The timeline shown on the bottom at the figures 13 and 14 when paused there 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 construction of 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 and draw vertexes on the selected rendering stage, one important use is the program section in the menu on the left, this allows 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 it can make break points like a common Visual Studio break point and debug the variables. This step by step shader debugging is likely to not work on a old laptop, there is however evidence that it works in different computers.

2.7.2 Performance Analysis

When using "Performance Analysis" it will display a wide variety of options to select, "Application Settings" and "Trigger and Actions" do not need to be changed, "Activity Type" will show a list of four items, unless it's required to trace other related processes "Trace Application" option will do.

"Trace Settings" start 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 needs to be pressed to start the analysis, it will generate an 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 give enough understanding to it's 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

 DEBUGGER COMPARISONS

3.1 OPEN SOURCE APPLICATIONS

3.1.1 *Comparison table*

	Bugle	APITrace	GLIntercept	GLSL-Debugger	VOGL
	Graphic User Interface;	Generation of organized trace files;	Easy to set up;	Graphic User Interface;	Graphic User Interface;
	Create information filters;	Trace organized per frame;	It's unnecessary to install additional tools to build;	Easier for an inexperienced user;	Generation of organized trace files;
	Chain information filters;	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;	Profiling a trace file;	Taking over the camera;	A different window per shader type;	Exporting frames to editable json;
	Configure metadata statistics;	Trimming a trace file;	Plugins may increase the potential for interested users;	View shader variables contents (incomplete);	Trimming a trace file;
	Taking over the camera;	Replaying the trace as long the trace file exists;	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;

	<p>Overriding context attributes;</p> <p>Screenshot / Video capturing;</p> <p>Create eps vector graphics screenshots;</p> <p>Step by Step debugging;</p> <p>Choose which function to stop at.</p>	<p>Creating videos with the trace file;</p> <p>View loaded uniforms when looking up the state with GUI ;</p> <p>Cross-Platform compatability.</p>	<p>Creation of xml files as a trace file;</p> <p>Configuration inside .ini files in a understandable C based code;</p> <p>Updating to a newer OpenGL requires only an additional include list;</p> <p>View total number of OpenGL calls at the end of trace.</p>	<p>View extension count on real-time per frame;</p> <p>Step by Step debugging</p> <p>Edit parameters on the next OpenGL function to call;</p> <p>Choose which function to stop at.</p>	<p>Dumping buffer state and screenshots;</p> <p>Editing trace files while replaying;</p> <p>Conversion from APITrace to VOGL;</p> <p>Cross-Platform compatability</p>
Cons	<p>Hard to install on windows;</p>	<p>Trace files are unreadable without the application</p> <p>Staying on the debugged application's working directory to trace;</p> <p>Command line dependent;</p>	<p>Default trace files aren't very organized;</p> <p>A beginner may not understand how to use the configuration;</p> <p>No Linux version.</p>	<p>Current code untested on windows;</p>	<p>Still in early alpha stages.</p>

Table 1.: Open Source Applications Pros and Cons table

3.1.2 Feature table

It should be noted in this section about the following tags:

- *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.

	APItrace	Bugle	GLIntercept	GLSL	VOGL
GUI	partial	yes	no	yes	yes
Trace step-by-step	no	yes	no	yes	no
Replay trace	yes	bugged	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	bugged	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	no	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

3.2 COMMERCIAL/FREWARE APPLICATIONS

	CodeXL (partially tested)	Nsight
Pros	Visual Studio extension; Has it's 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 (Not instant); Features may not work properly on laptop.

Table 3.: Commercial/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.

3.3 CONCLUSIONS REGARDING STATE OF ART

It's possible to conclude that among the five open source debuggers GLIntercept is the most simplest one, it's basic functions is limited to logging without using any GUI to read the output log, GLIntercept also has a great advantage of allowing pluginsto extend the wrapper.

APITrace and VOGL are very similar to each other, while APITrace is older than VOGL the latter has an improved interface, in fact VOGL has shown more future potential then APITrace. However unlike VOGL, APITrace is up to date with the lates OpenGL specifications. These two debuggers are the only ones showing the ability to replay, even compared to the two proprietary debuggers.

GLSL and VOGL have the most professional interfaces, GLSL is the only debugger among the five, capable of simulating shader debugging, however this feature isn't up to date with the latest OpenGL specifications.

GLSL and BUGLE are the only ones capable of step by step debugging. Actually VOGL is capable of pausing a replay, however it's slow and what it actually does is to replay until a certain frame.

Bugle is very rich compared to the other debuggers, it has more features, shows more information and it's possible to configure it to tailor to the user needs. Among the five debuggers Bugle is currently the most complete debugger. 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. 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 of their respective manufacturer's hardware and show detailed statistics and logs.

The proprietary debugger's flaw is the fact that they are hardware limited including the fact that very old GPUs are incompatible with some features. 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. Then again which should be used is hardware dependent.

Part II

INCORPORATING THE DEBUGGER IN NAU

USING AN EXISTING DEBUGGER

Nau is an open-source 3D graphics engine that works with OpenGL as its 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, apart from the shaders.

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

An embedded profiler detailing both CPU and GPU times provides helpfull 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.

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 debugger of choice was GLIntercept because Nau is a windows library and uses visual studio solutions, thus using GLIntercept was a faster and easier choice.

This decision resulted into an altered version of GLIntercept, the original version used was the 1.3.0 version thus most plugins and features up to that point should work with the changes. The main reason why GLIntercept was chosen was because it's the simplest of the debuggers. By doing so GLIntercept will include it's plugins while integrating it's basic logging features into Nau.

4.1 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 the plugins alters an internal function table, but now it's capable of removing plugins which implies methods to clean the function table in order to avoid crashing.

This was done because the class `ExtensionFunction` indexes the functions to execute during wrapping in an array, if it's 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 (inefficient) but
            //it's 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 overridden function
        //if not deleted
        const FunctionData *foundFunc =
            functionTable->GetFunctionData(overrideIndexList[i]);
        *foundFunc->internalFunctionDataPtr =
            (void**)overrideCurrIndexList[i];
    }
}
currExtensionIndex -= removedExtensionsCount;
overrideIndexList.clear();
```

```
    overrideCurrIndexList.clear();
    overrideMustDeleteList.clear();
}
```

It was also placed a new function to restore the original loaded settings, 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.

Also it was decided to disable any functionality to the wrapped functions until it's 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 it's active. This forces the necessity to use `gliSetIsGLIActive(true)` when releasing the paused state and `gliSetIsGLIActive(false)` when returning to the paused state. The driver's `AddLoggerString` was exported as `gliInsertLogMessage` in order to allow Nau to add personal messages to the log file.

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

4.2 CHANGES ON NAU

Nau will now interact with GLIntercept, this also means that the created `OpenGL32.dll` is a must have for Nau. Also a `gliConfig.ini` is necessary to load standard default configurations.

The `<debug>` tag now exists and is important 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 it's 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 all he has is to edit this file. How tags work shall be mentioned on the usage section.

It's 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.

Aside from additional tag it's also been incorporated within Nau methods to read which OpenGL state to fetch and the corresponding functions for the enum.

4.3 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 stopping the rendering, this causes the rendering result's to pause (it won't show the results at all which may look glitched), once it's 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 uniforms.

When paused the composer will attempt to gather buffer data, by using OpenGL methods. It'll 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'll map the remaining buffers, since the only VAO buffers have full information the remaining buffers will allow the user to change the table structure.

In order to improve data gathering when paused it's possible to render Nau pass by pass or frame by frame, while the option to render pass is on the debug menu there is the advanced pass controller which allows to: render one pass, render a frame (or whats left of it), render all passes until it renders the target pass.

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 it's easier installation, practical use and compatibility. This does mean it's include folders require an additional `dirent.h`, since `dirent` has all it's functions embedded within the header file no new dll is required aside GLIntercept's wrapper.

The information is extracted using methods from Lighthouse 3D's `vsGLInfoLib` [\[Cos\]](#) [\[Ramb\]](#), by changing the existing methods and removing the unused ones `glInfo` was created. These new functions do not require GLIntercept.

In addition to Nau's new OpenGL state methods a new window was created, this doesn't necessarily need to be in the debug menu, it was placed there because since it was a new functionality but it can be moved away in future updates.

HOW TO USE 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 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's "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>
  <shaderlog>
    ... see shaderlog section
  </shaderlog>
  <displaylistlog>
    ... see displaylistlog section
```

```

</displaylistlog>
<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, however there are a few exceptions.

5.1 FUNCTIONLOG

This will affect the created log file, when enabled a txt log will be created, while most tags are self explanatory it may be important to know that:

- Using `<enabled value="true"/>` is mandatory to create the log file;
- 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>

```

5.2 LOGPERFRAME

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

```
<logperframe>
  <logperframe value="bool"/>
  <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.

5.3 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>
```

5.4 IMAGELOG

Image log determines if the logger will log images, the images should be logged on a separate subfolder 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"/>
    <imagesavepbuffervertex value="bool"/>
    <imageicon>
        <imagesaveicon value="bool"/>
        <imageiconsize value="uint"/>
        <imageiconformat value="png"/>
    </imageicon>
</imagelog>

```

5.5 SHADERLOG

This will create a copy of the OpenGL shaders on the main log's folder, these shaders will be saved on a Shader folder.

```

<shaderlog>
    <enabled value="bool"/>
    <shaderrendercallstatelog value="bool"/>
    <shaderattachlogstate value="bool"/>
    <shadervalidateprerender value="bool"/>
    <shaderloguniformsprerender value="bool"/>
</shaderlog>

```

5.6 DISPLAYLISTLOG

When enabled OpenGL display lists are saved on a DisplayList folder under the main log path.

```

<displaylistlog>
    <enabled value="bool"/>
</displaylistlog>

```


5.7 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 they are 3 separate booleans (for example `ColorBufferLog` is now: `<frameprecolorsave>`, `<framepostcolorsave>`, `<framediffcolorsave>`).

```
<framelog>
  <enabled value="bool"/>
  <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>
```

5.8 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>
```

5.9 PLUGINS

Adding a plugin is slightly different on Nau because of the configuration format, but not too different, 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 are 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's 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"/>
```

If a different name is given to the plugin it may not be guaranteed to work.

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

5.10 HOW TO GET 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 enums (and their name and values) and the function name.

Using the example it's possible to say 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.

The function parameters aren't used at the moment, but may prove useful in case of future updates, the enums can be copied from the official `gl.xml` khronos spec file, they use the same enum format except for length.

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

5.11 HOW TO USE THE COMPOSER

The composer now has an additional `Debug` menu, in order to enable the other options it's 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.

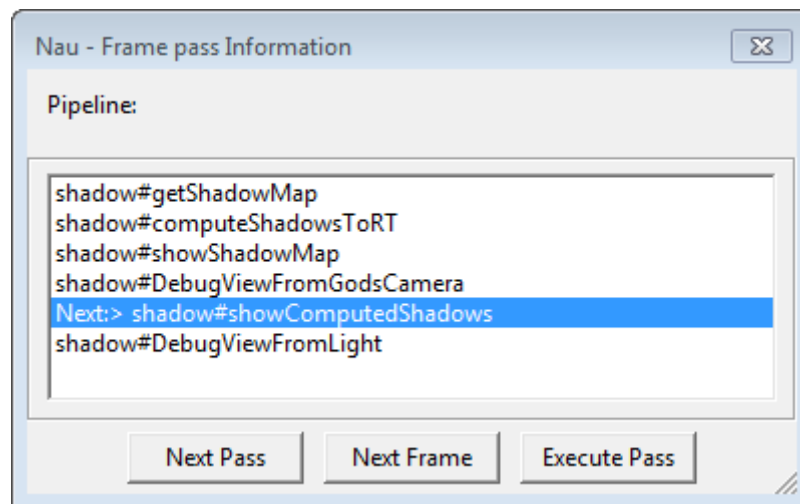


Figure 15.: Composer's Pass controller.

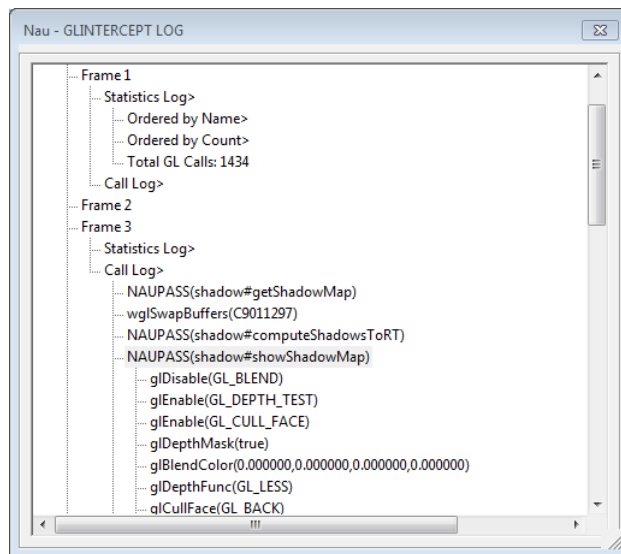


Figure 16.: Nau's GLIntercept log viewer.

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 being logged. Take note that when a line ends with > as seen in the figure below it means it has a subnode.

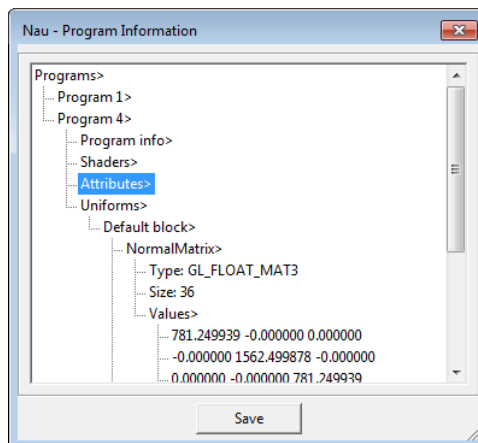


Figure 17.: Nau program information.

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 a thousand lines) they had to be paged for both visual reasons and usability reasons since reading a buffer that big would take long to load. The option

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;

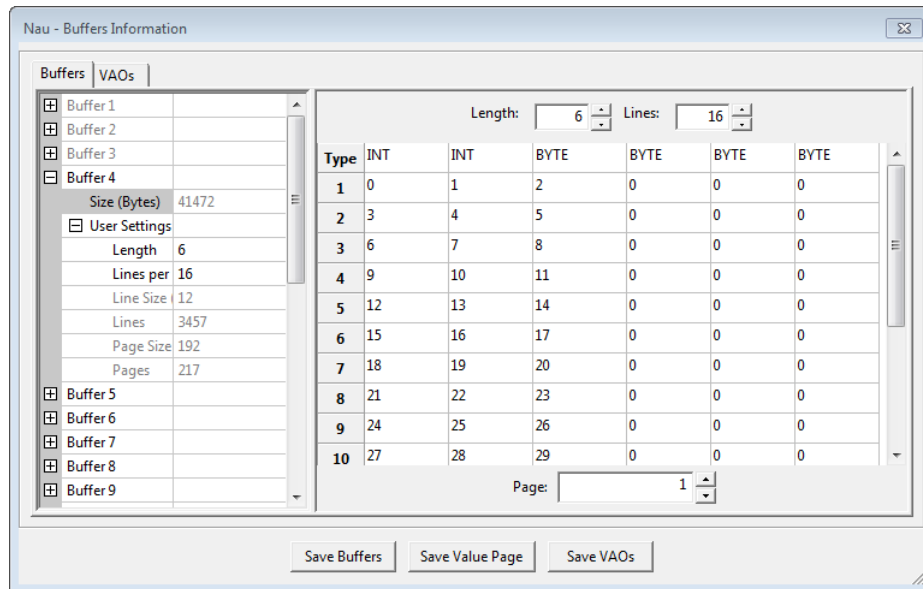


Figure 18.: Nau buffer information.

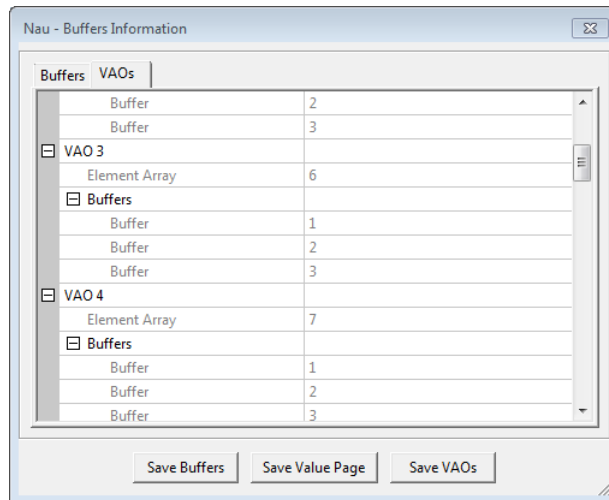


Figure 19.: Nau VAO information.

In order to incorporate Nau's new state reading functions a new Window was created, all it needs is to load the xml file with the states and it'll load the states according to figure 20.

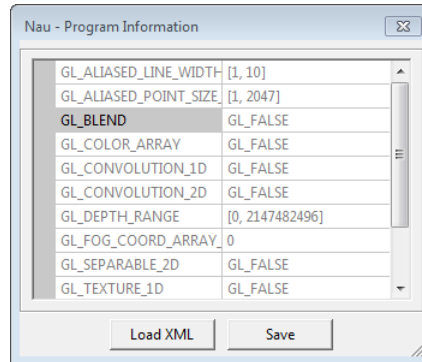


Figure 20.: Nau State information.

CONCLUSIONS AND FUTURE WORK

6.1 CONCLUSIONS

This work concludes that the current Open Source debuggers while not covering all features can complement each other with features the other do not have, while most have the basic debugging features they often lack something.

Organizing information itself isn't easy, as mentioned in the introduction this is almost a "finding a needle in a haystack situation" however this "haystack" is also the most complete source of information.

It is in fact hard to find a bug even with a debugger, mostly because it's necessary to remember where the bug needs to be found or with which method. Also in order to find a bug it's important to have a good understanding regarding how OpenGL works, which may not be so easy and even confusing in some cases.

All free debuggers have shader debugging problems, mostly because it's run on the graphic board's kernel unlike common application code, it's debugging is must be parsed with a proper IDE. This limits proper shader debugging to the respective manufacturer's debugging tools such as CodeXL and Nsight. Perhaps GLIntercept's SciTe is the correct way to debug a shader since it acts almost like it's own IDE to compile the shader, unfortunately it's not bug free.

Creating the comparison tables helped put a conclusion on the current state of art of the debuggers, it helps see which of the debuggers is the most complete and interesting. It also helps noting which features are actually very unique where others fail to have.

Bugle is in fact the most complete debugger but it's a Unix based debugger that does not carry any cross compilation features, which is unfortunate in a sense since most 3D applications are Windows video games. The only way to run it on Windows is through the use of Unix environment emulators like MingW.

Apitrace may actually be very interesting if combined with scripts since it's command line based, how it integrates with ffmpeg and allows replays are unique features found in this debugger. In fact not even commercial debuggers perform replays like Apitrace.

GLIntercept while ideally a Windows debugger it may not cover all features, however it's plugin features and easy and documented code allows new programmers to extend the debugger. Since it's coded using Visual Studio it's source code is ideal for other Visual Studio projects.

GLSL is in fact the debugger with the most number of incomplete features, the original debugger was made for OpenGL 2.0 making it's shader debugging very limited. The debugger itself is very beginner friendly recommended for those less experienced with debugging. Unfortunately the developers are not focused on developing the Windows version of this debugger thus the Windows version has it's own bugs.

VOGL is a newcomer, and it already shows promising results, it's very similar to APITrace, while it's still alpha and has it's own expected share of bugs it's capable a several important debugging operations.

Ideally the best way to debug is to use the commercial debuggers, they are both very similar when it comes to overall features, however Nsight allows to demonstrate runtime statistics while CodeXL requires pausing.

Every experience counts, whether it's a frustrating installation or a smooth GUI it helps understand better what do we want and what we don't want in a debugger, experiencing all of this while cataloguing every requirement in order to debug was a very good experience.

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. For instance, plugininformation hasn't been fully integrated yet, while there may also be a few faults that miss the eye.

The new additional features are actually very useful, it allows a user to integrate additional GLIntercept pluginsincreasing the range of possibilities for *Nau* itself, the logs should also help the project owner check for his own mistakes or the even developer itself can find mistakes created by the engine.

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

6.2 PROSPECT FOR FUTURE WORK

GLIntercept's pluginfeature is in fact interesting, for future work there is the possibility to create a pluginthat affects Nau directly. Regardgin the integration not all information generated by plugins is used by Nau, being available on text format only. Integrating this information in Nau's GUI, Composer, could be a helping hand in some debugging scenarios. Also there should be room for performance improvements, large log often causes a large workload on the Composer.

Shader debugging is still lacking, for there was no successful shader debugging, this means that this area requires more work and a proper shader compiler may greatly help future programmers.

BIBLIOGRAPHY

- [AAWL14] António Ramires Fernandes Andre Alexandre Wang Liu. Open source debuggers and integration with a 3d engine. page 8, 2014. Sibling article.
- [AMD13] AMD. Codexl. CodeXL. AMD, November 2013. [Online; accessed 14 Noevember 2013].
- [Com] Wikipedia Community. Vogl. Wikipedia. [Online; accessed 24 October 2014].
- [Cos] Pedro Costa. Cg-pi/mv/vs/vsglinfolib.cpp at master · pfac/cg-pi · github. GitHub. [Online; accessed 7 July 2014].
- [DS13] Dtrebi...@gmail.com and SkewMat...@gmail.com. Glintercept - opengl call interceptor/logger. Glintercept. Code.google.com, June 2013. [Online; accessed 1 Noevember 2013].
- [FJea13] Fonseca, Alexander Monakov José, and et al. Apitrace. GitHub, November 2013. [Online; accessed 1 Noevember 2013].
- [HX13] Hanson and Chris 'Xenon'. GlsI-debugger. Github, October 2013. [Online; accessed 14 Noevember 2013].
- [KS10] Thomas Klein and Magnus Strengert. GlsIdevil - opengl glsl debugger. GlsIDevil. [Http://www.vis.uni-stuttgart.de](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].
- [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].
- [Rö] Toni Rönkkö. Diredt api for microsoft visual studio. Softgalleria. [Online; accessed 24 August 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.
- [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

DLL

DLL Injection

[Wik]: 'a technique used for running code within the address space of another process by forcing it to load a dynamic-link library', 24, 30, 31, 98

Dynamic Linked Library, 5, 30, 31, 45, 46, 98

Frames per second, $FPS = \frac{frames}{seconds}$, 11, 12, 20

GUI

Graphic User Interface, 5, 8, 22–24, 28, 31, 46, 61, 68, 70, 90, 95, 97

IDE

Integrated Development Environment, 4, 89

Plugin

plug-in, plugin, extension, add-on, add-on:
Piece of software that adds a specific feature, 33, 36, 37, 40–42, 45, 46, 68, 73, 74, 90, 97, 98

VAO

Vertex Array Object, 76, 86, 87

Part III

APPENDICES



INSTALLATION

A.1 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 `glx.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.
- `gtkglx`, `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 it's important to 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 it'll be shown a different approach, the latest version of GTK+ is hard to install, but 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 glib isn't 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, 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 the current version of FFmpeg isn't satisfactory, 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, <root bugle directory> shall be referred as the folder where the extracted contents are, once extracted 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.

Afterwards create a .bugle folder in the home directory where,place filters and place statistics file. It's possible to get both .bugle files in <root bugle directory>/doc/examples.

A.2 APITRACE

Before building the application following applications are needed 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's 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
```

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 it's necessary 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 build either with the solution or using the following command:

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

If QT isn't properly configured it won't generate a solution with `qapitrace.exe`.

A.3 GLINTERCEPT

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 shader editor, SciTE is a third party solution which will not be analysed.

The original solution comes with Visual Studio 2008 but it's possible to upgrade the solution to more recent versions. Visual Studio 2008 express edition is more than enough to build the GLIntercept solutions however SciTE needs `afxwin.h` which is unavailable in express edition.

After building the gli configuration file `gliConfig.ini` needs to know where is the required files such as the headers, plugins and xml formats, these are outputted in the GLIntercept's Bin folder, these files are in different sections requiring the configuration to change:

- `FunctionLog - uses BaseDir = "<bin folder>\XSL";`
- `PluginData - uses BaseDir = "<bin folder>\Plugins";`
- `InputFiles - uses GLFunctionDefines = "<bin folder>\GLFunctions\gliIncludes.`

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

In order to debug with GLIntercept it's necessary copy or create the `gliConfig.ini` into the debugging application's workspace along with the wrapper, to manually [inject a DLL](#) .

A.4 GLSLDEVIL/GLSL-DEBUGGER

In order to build GLSL download it from <https://github.com/GLSL-Debugger/GLSL-Debugger> repository, have `cmake`, `glut` libraries, `glew` libraires, `bison`, `flex` and `qt4` installed.

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

In Windows create a Visual Studio solution with `cmake`, on Unix systems create a common make solution relying on `cmake -Bbuild` in the command line and then use `make` command on the new build directory. Windows systems may force the user 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 start it on it's 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 `glslldb`.

A.5 VOGL

Here is demonstrated the Windows installation of VOGL, before building VOGL a few libraries need to be installed, specifically `pthread 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 called `external`, sibling to `vogl`, is required, so on the same directory where the VOGL project is placed use the following commands:

```
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, add either `external/SDL/VisualC/SDL/Win32/Release` or `external/SDL/VisualC/SDL/Win32/Debug` (depending on which is compiled) to the `PATH` windows variable because of DLL dependencies.

Same goes for `pthread`s, extract the `Pre-built.2` folder and rename it to `pthread.2` then move it to `external/windows`, add `external/windows/pthread.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 QT 5.3 is installed

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

B

USE / CONFIGURATION

B.1 BUGLE

B.1.1 *Statistics configuration*

This file is used to determine statistics format, its existence is imperative whenever using stats outputs such as `logstats` and `showstats`, see section [B.1.2.2](#). It can be configured to fetch the statistics in four forms:

- `s(<stat>)` - the value of the stat at the start
- `e(<stat>)` - the value of the stat at the end
- `d(<stat>)` - the difference in the stat ($e(<stat>) - s(<stat>)$)
- `a(<stat>)` - the average value of the stat

`<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's 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's 0 so it won't be outputted and flood the statistics with zeros, for example some gl functions used during initialization has zero calls for the rest of the application and are simply flooding the screen. In the label unlike the previous example this one lists as "*", this allows to label with the function name instead of a static label.

B.1.2 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.

B.1.2.1 *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`

Each of these `filtersets` gathers a certain type of statistics, this is described in the statistics section. However, these filters alone will not produce any output, they exist only to enable the mentioned statistics. 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's 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 21:

```
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"
  }
}
```

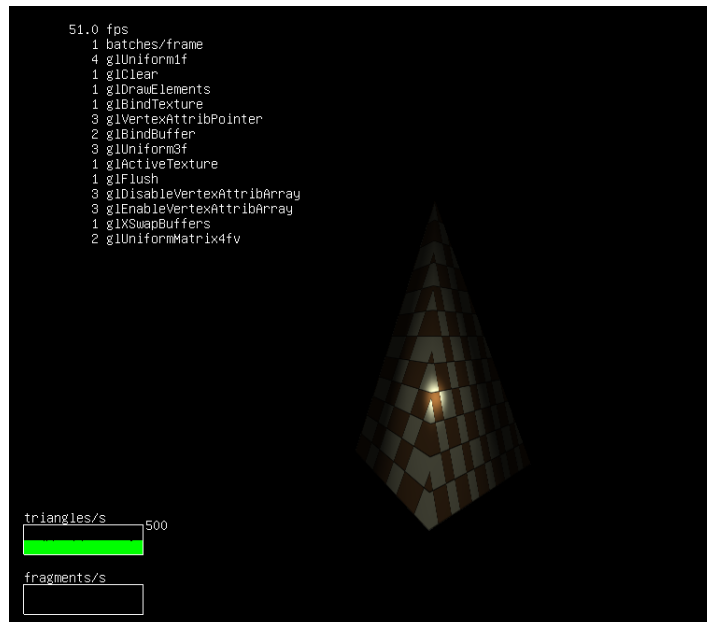


Figure 21.: Bugle showstats example.

B.1.2.2 *Trace and Log filterset*

The most common features of the open source debuggers 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, it's 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 behavior, 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 using the mentioned `chain trace` 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)
...
```

B.1.2.3 *Error checking filtersets*

This section will refer to error logging and control filtersets.

The `filterset` error exists as a safeguard calling `glGetError` after each function allowing the application to read it's own errors without Bugle accidentally collecting them (in case the application needs to read and know there was an error Bugle won't accidentally hide it from the application).

The `filterset` checks will search for undefined behaviour such as an OpenGL call attempt to use unreadable memory and ignore it instead of allowing it to crash the application, resulting in a warning into the log.

For instance when `filterset checks` is active and `glEnableVertexAttribArray(7)` within the application 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.
```

This results in the application ignoring what usually would cause a crash.

The `filterset showerror` will trace errors into to standard error instead just like `trace logs` the application. These `filtersets` can be used in the following manner:

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

B.1.2.4 *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 context-attrs` 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 it's 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 it's 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
    {
        version "1.5"    #Set maximum version

        disable "GL_EXT_framebuffer_object"
        disable "GL_EXT_framebuffer_blit"
    }
    filterset contextattribs
    {
        major 3
        minor 2
        flag "debug"
        flag "robust"
        profile "core"
        profile "!compatibility" #disables compatibility
    }
}
```

This results in the application receiving OpenGL as version 1.5 when asking for the version string, however the value of `GL_VERSION` will be 3.2.0, in this case it resulted in 3.2.0 NVIDIA 319.60 as the `GL_VERSION`.

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
```

B.1.2.5 *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
```

Note that `show extensions` isn't affected by the overrides from the previous sections.

B.1.2.6 *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 yes
    }
}
```

And the resulting debug messages can be the following:

```

[INFO] logdebug.message: Buffer detailed info: Buffer object 1 (bound
to GL_ELEMENT_ARRAY_BUFFER_ARB, usage hint is GL_STATIC_
DRAW) will use VIDEO memory as the source for buffer object
operations. [source: API type: other id: 131185]
[INFO] logdebug.message: Buffer detailed info: Buffer object 1 (bound
to GL_ELEMENT_ARRAY_BUFFER_ARB, usage hint is GL_STATIC_
DRAW) will use VIDEO memory as the source for buffer object
operations. [source: API type: other id: 131185]
[INFO] logdebug.message: Buffer detailed info: Buffer object 2 (bound
to GL_ARRAY_BUFFER_ARB, usage hint is GL_STATIC_DRAW) will
use VIDEO memory as the source for buffer object operations. [source:
API type: other id: 131185]
[INFO] logdebug.message: Buffer detailed info: Buffer object 2 (bound
to GL_ARRAY_BUFFER_ARB, usage hint is GL_STATIC_DRAW) will
use VIDEO memory as the source for buffer object operations. [source:
API type: other id: 131185]
[INFO] logdebug.message: Texture state usage warning: Waste of
memory: Texture 0 has mipmaps, while it's min filter is inconsistent
with mipmaps. [source: API type: other id: 131204]
[INFO] logdebug.message: Texture state usage warning: Waste of
memory: Texture 0 has mipmaps, while it's min filter is inconsistent
with mipmaps. [source: API type: other id: 131204]

```

B.1.2.7 *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"
    }
}

```

B.1.2.8 *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's 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, the available codecs can be seen in ffmpeg's documentation, 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"
    }
}
```

B.1.2.9 *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 22 shows what it looks like when converted to png, the example is the same application as figure 21.

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

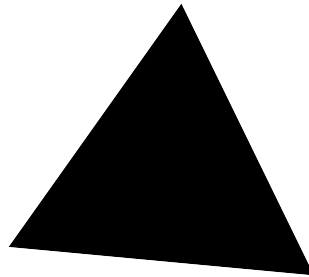


Figure 22.: Bugle eps file converted to png.

B.1.2.10 *frontbuffer filterset*

The `filterset frontbuffer` forces the application to always output the frontbuffer's state, in other words the application screen will only show the frontbuffer.

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

B.1.2.11 *Wireframe filterset*

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

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

B.1.3 *Graphic User Interface*

The `gldb-gui` needs to be run on the command line within the target application's working directory, if no chain is specified it will use an empty pass, even it can still show the application's state during the last pause as show in 23. In the bottom of the state bar it allows to only show the selected states or the ones that got modified between two different pauses.

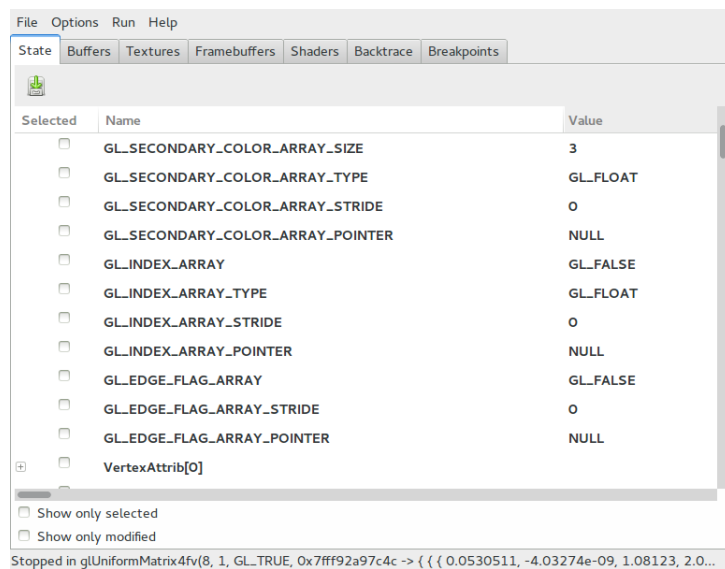


Figure 23.: Bugle gdb state tab.

While it's paused it's possible to view the application's buffers as shown in figure 24, it's possible to change the format according by changing the line composition according to the labels on left.

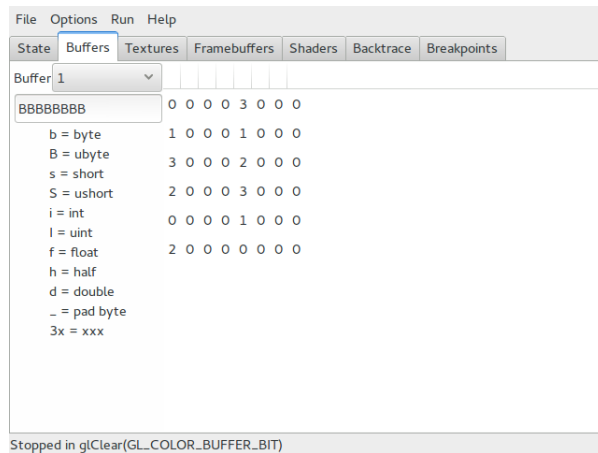


Figure 24.: Bugle buffers.

Another noteworthy feature of the GUI is that it reports any error encountered within the application's shaders as shown in figure 25. It's also possible to see both the application's textures and framebuffer in its own corresponding tabs.

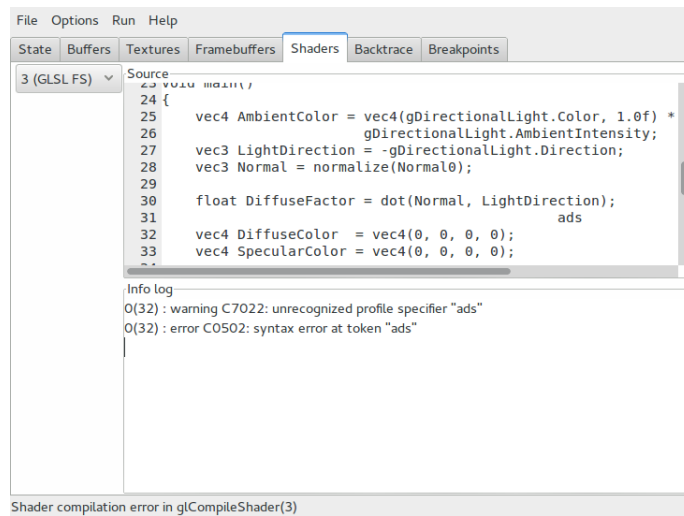


Figure 25.: Bugle glldb shader error encountered.

It also allows step by step debugging and placing breakpoints within the application, to debug step by step use the step option (or its respective hotkey) on the Run menu, to place breakpoints add the specific function which will break when traced on the Breakpoints tab, in the figure 26 the application will pause whenever `glDebugMessageCallbackARB` is called.

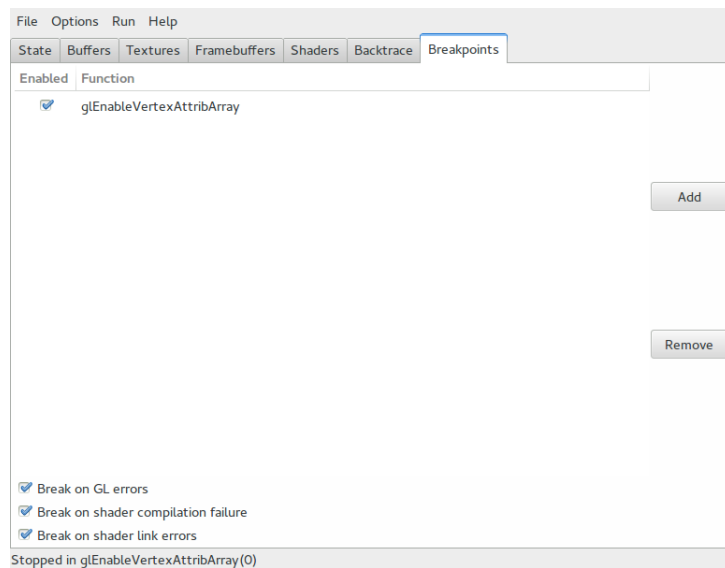


Figure 26.: Bugle gdb breakpoint.

B.2 APITRACE

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 the 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's 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's 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 print the tracing warnings on the command line.

Dumping OpenGL frame call – It's 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's 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
```

B.2.4 *Trimming trace file*

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 trimmed.trace \  
application.trace
```

However trimming it from a beginning different from 0 will most likely create an unreplayable trace file (because it's 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 it's 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).

B.2.6 *Apitrace's GUI*

With the GUI `qapitrace.exe` it's possible to view the generated trace files, in windows systems it can start by dragging the `.trace` file to the `qapitrace` executable. The trace shall be listed in a node tree with frames per node, each frame shall have his calls as a sub-node. In the figure 27 it's possible to see how the `qapitrace` GUI shows trace files.

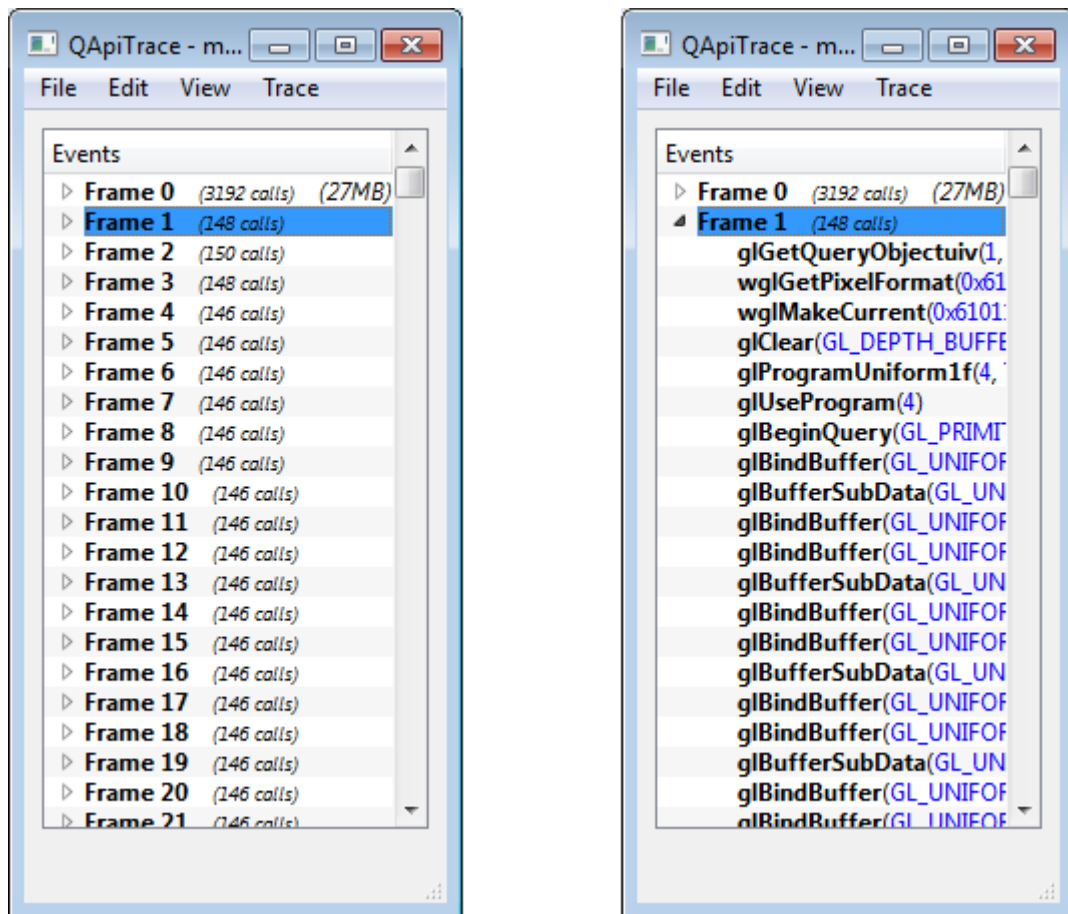


Figure 27.: On the left shows how Apitrace splits the log per each frame, on the right it has a frame node expanded showing it's function log.

It can edit the gl calls by right clicking on the traced call, considering that by editing can cause an error it is possible to conclude it may affect the replay action.

It can also check the replay of the trace on the trace menu, it also possible to get a partial replay by right clicking on a frame and use “Lookup State” option, once the replay is done it'll will show the state as shown in figure 28.

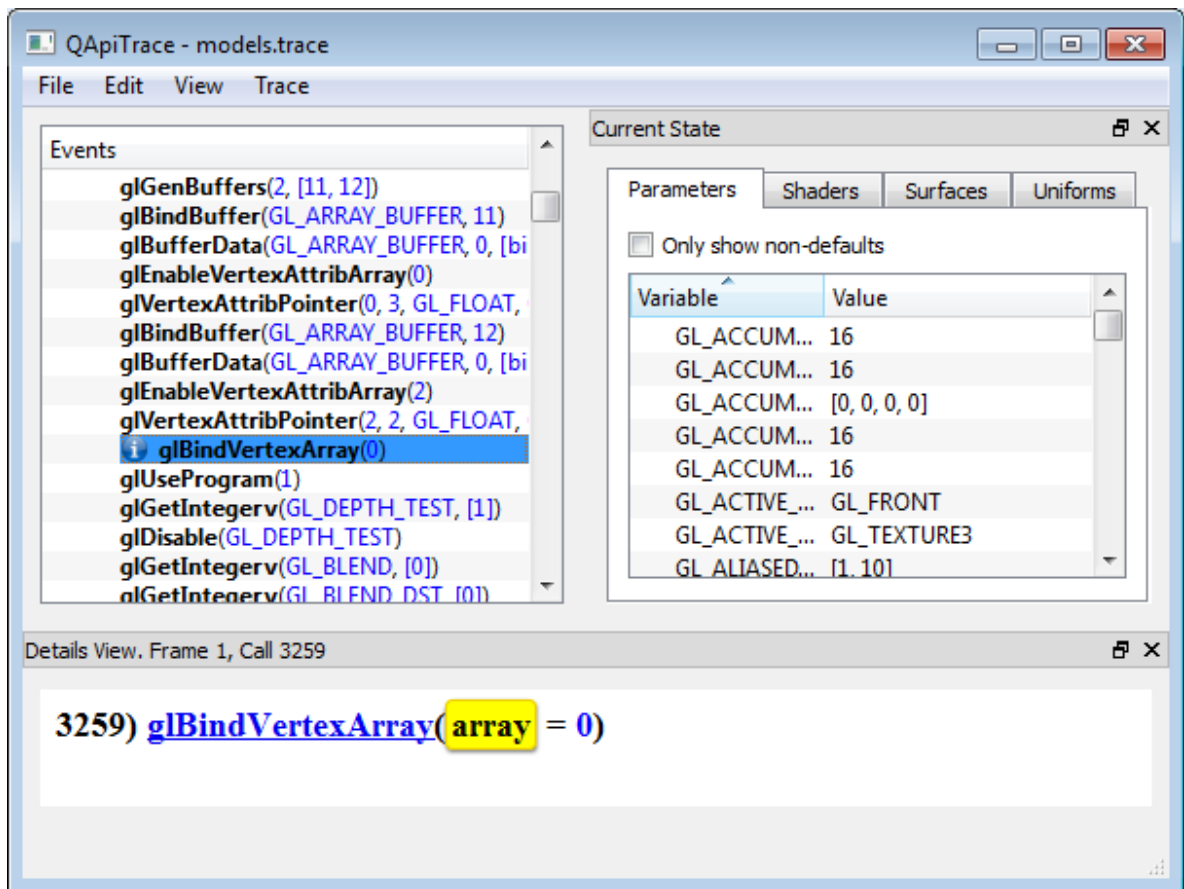


Figure 28.: Looking up the state before changing the program, it allows to see the current program's parameters, shaders, buffers(surfaces tab) and uniforms.

B.3 GLINTERCEPT

B.3.1 *Tracing*

GLIntercept makes normal tracing calls, whenever an OpenGL function is called it's 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;

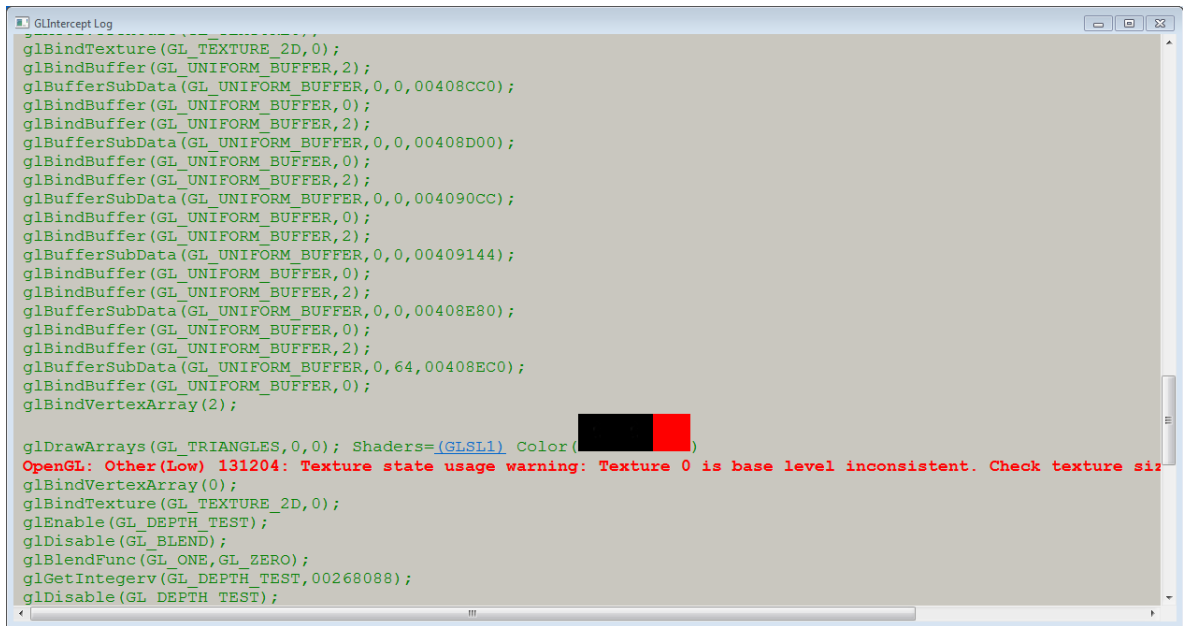
    //LogFormat   = XML;

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

The following is an example of a `GLIntercept` txt log:

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

In case `LogFormat` is active and set to `xml` it will output a `xml` file that can be rendered with internet explorer, this is shown in the following figure 29.



```
GLIntercept Log
-----
glBindTexture(GL_TEXTURE_2D,0);
glBindBuffer(GL_UNIFORM_BUFFER,2);
glBufferSubData(GL_UNIFORM_BUFFER,0,0,00408CC0);
glBindBuffer(GL_UNIFORM_BUFFER,0);
glBindBuffer(GL_UNIFORM_BUFFER,2);
glBufferSubData(GL_UNIFORM_BUFFER,0,0,00408D00);
glBindBuffer(GL_UNIFORM_BUFFER,0);
glBindBuffer(GL_UNIFORM_BUFFER,2);
glBufferSubData(GL_UNIFORM_BUFFER,0,0,004090CC);
glBindBuffer(GL_UNIFORM_BUFFER,0);
glBindBuffer(GL_UNIFORM_BUFFER,2);
glBufferSubData(GL_UNIFORM_BUFFER,0,0,00409144);
glBindBuffer(GL_UNIFORM_BUFFER,0);
glBindBuffer(GL_UNIFORM_BUFFER,2);
glBufferSubData(GL_UNIFORM_BUFFER,0,0,00408E80);
glBindBuffer(GL_UNIFORM_BUFFER,0);
glBindBuffer(GL_UNIFORM_BUFFER,2);
glBufferSubData(GL_UNIFORM_BUFFER,0,64,00408EC0);
glBindBuffer(GL_UNIFORM_BUFFER,0);
glBindVertexArray(2);

glDrawArrays(GL_TRIANGLES,0,0); Shaders=(GLSL1) Color( )
OpenGL: Other (Low) 131204: Texture state usage warning: Texture 0 is base level inconsistent. Check texture size
glBindVertexArray(0);
glBindTexture(GL_TEXTURE_2D,0);
glEnable(GL_DEPTH_TEST);
glDisable(GL_BLEND);
glBlendFunc(GL_ONE,GL_ZERO);
glGetIntegerv(GL_DEPTH_TEST,00268088);
glDisable(GL_DEPTH_TEST);
```

Figure 29.: GLIntercept xml log rendered in internet explorer

B.3.2 *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, it's format will be according to:

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

B.3.3 *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_2_0\Plugins";

    Plugins
    {
        (...)

        OpenGLShaderEdit = ("GLShaderEdit/GLShaderEdit.dll")

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

B.3.4 *ARB_debug_output Logging*

ARB_debug_output is produced along the logs, it is created by adding the following lines:

```
PluginData
{
    BaseDir = "C:\Program Files\GLIntercept_1_2_0\Plugins";

    Plugins
    {
        (...)

        DebugContext = ("GLDebugContext/GLDebugContext.dll")
        {
            ForceDebugMode = True;
            LogToFunctionLog = True;
        }
    }
}
```

```

LogToErrorLog = True;
BreakOnMessage = False;

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

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

```

B.3.5 *Extension override*

It is possible to change the program extension strings, however this will only affect the application if the application itself uses the extension strings, otherwise it won't make any noticeable effect.

```

PluginData
{

    BaseDir = "C:\Program Files\GLIntercept_1_2_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 screenshot in figure 30:

```

General Information
Vendor: NVIDIA Corporation
Renderer: GeForce 9300M GS/PCIe/SSE2
Version: 1.1.0 - Custom version string
GLSL: 1.0.0 - Custom shader version string
Num. Extensions: 238
DataSize:80
DataSize:112

```

Figure 30.: 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);

B.3.6 *Function statistics*

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

This plugincreates 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_2_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
...	

B.4 GLSLDEVIL/GLSL-DEBUGGER

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 31.

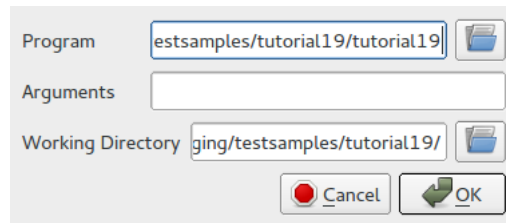


Figure 31.: GLSL opening a new application to debug, in the example it's opening an application called tutorial19

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 32 and 33 should be much more intuitive than a text description.

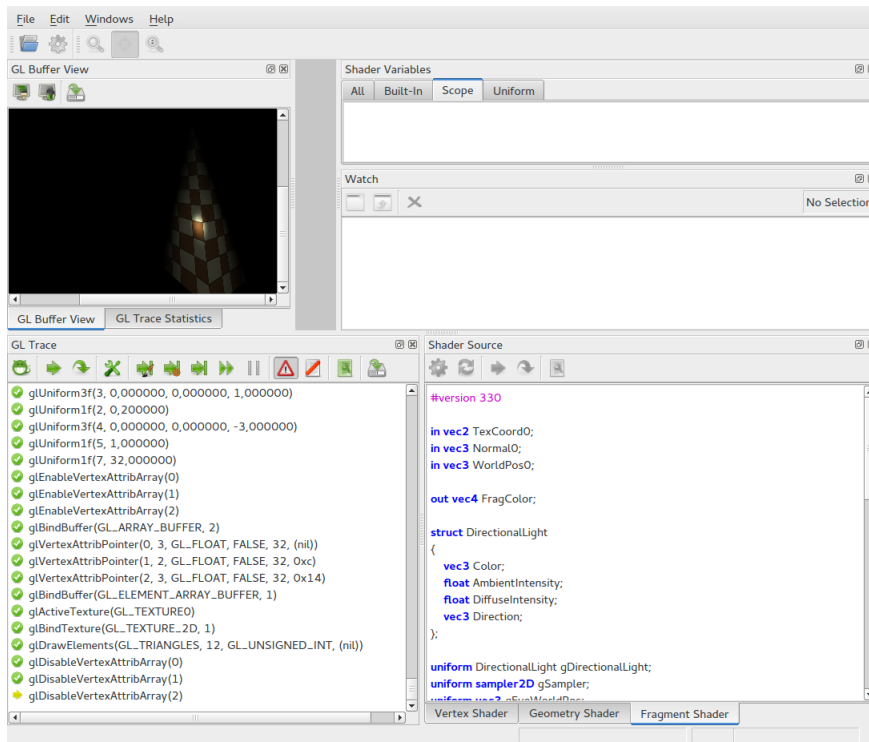


Figure 32.: GLSL showing buffer view and fragment shader.

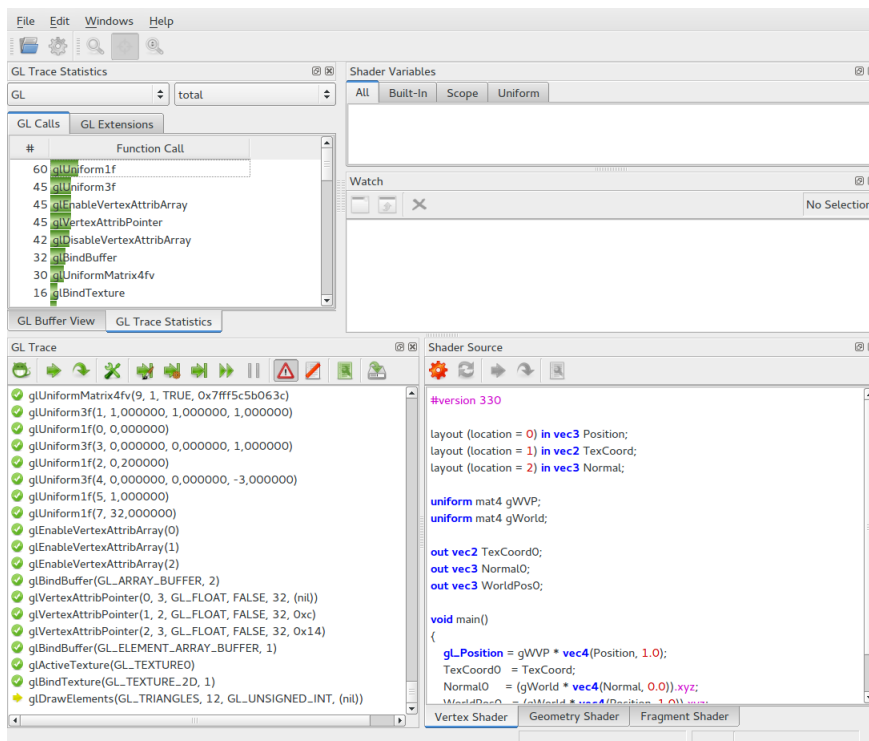


Figure 33.: GLSL it's featuring trace statistics and the vertex shader.

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.

B.4.1 *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's 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. It's hotkey is <F7>.
- Jump to next Shader Switch
This will resume running but pause as soon it reaches a the next change of shaders or essentially the next glprogram function. It's 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. It's 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. It's hotkey is <F8>.
- Stop (Alt+Break)
This will pause the application enabling diverse debugging options. It's 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.

B.4.2 *Shader*

This sub-window is only available when the next function to trigger is a draw call. Unfortunately only OpenGL 2 shaders work reliably, using mesa shaders for OpenGL 3 should also work however not as reliable as OpenGL 2.

- Debug Shader

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

- Reset Debug Session

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

- Step

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

- Step Over

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

- Edit Per-Fragment Option

This button will open output options for the fragment shader, thus it's only available in the fragment shader tab, the options are shown in figure 34. It's hotkey is `<F4>`.

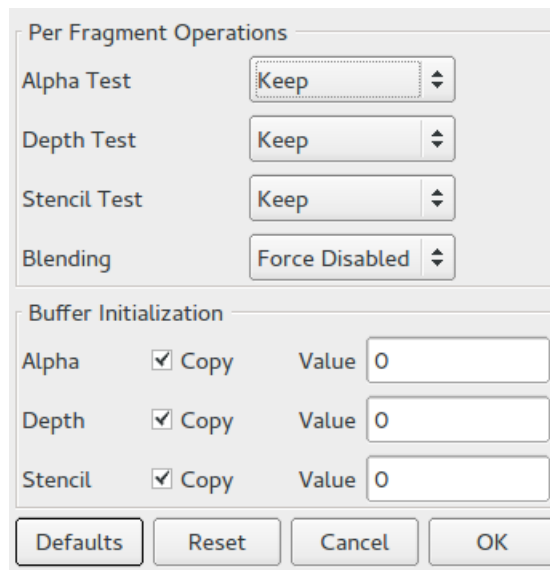


Figure 34.: Fragment shader per-fragment options.

B.4.3 *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.

- Tabs

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

B.4.4 *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's used before anything is drawn.

- Automatic Capture

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

- Save as image

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

B.4.5 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 35, 36 and 37.

Watch is available when one of the uniforms is chosen to be put on watch (double clicking). Once on watch it's possible to view them as shown in the screenshots, the figure 35 shows fragment coordinate viewing, 36 shows fragment color viewing and 37 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's possible to see the fragment shader results by debugging, the triangle shown in the figures 35 and 36 is the 3D model used in the application.

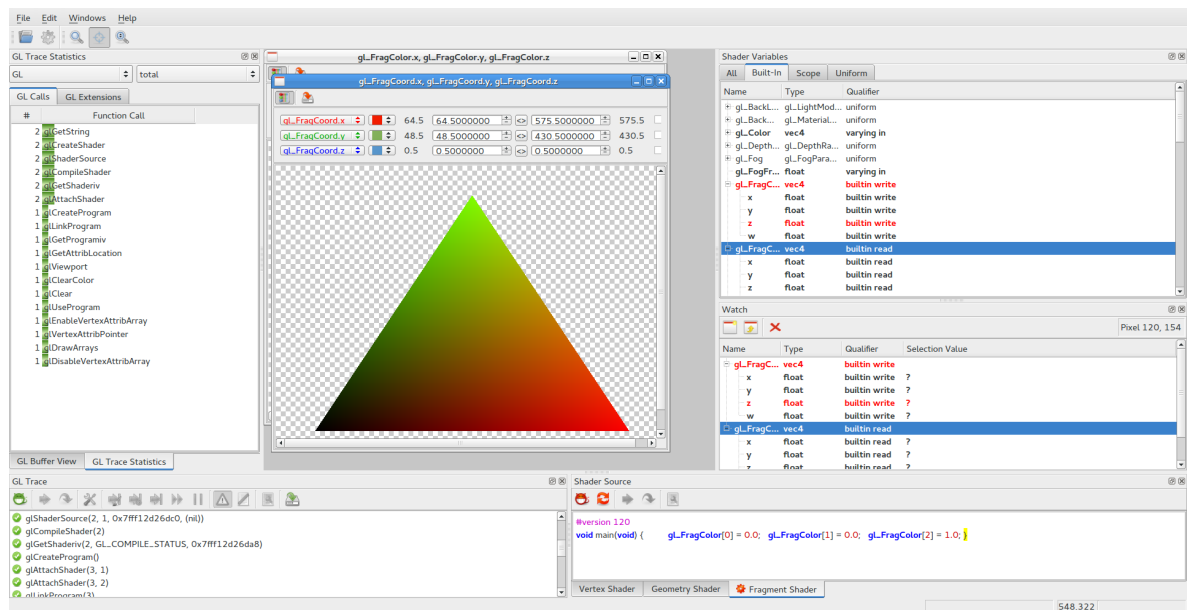


Figure 35.: Fragment coordinates viewer.

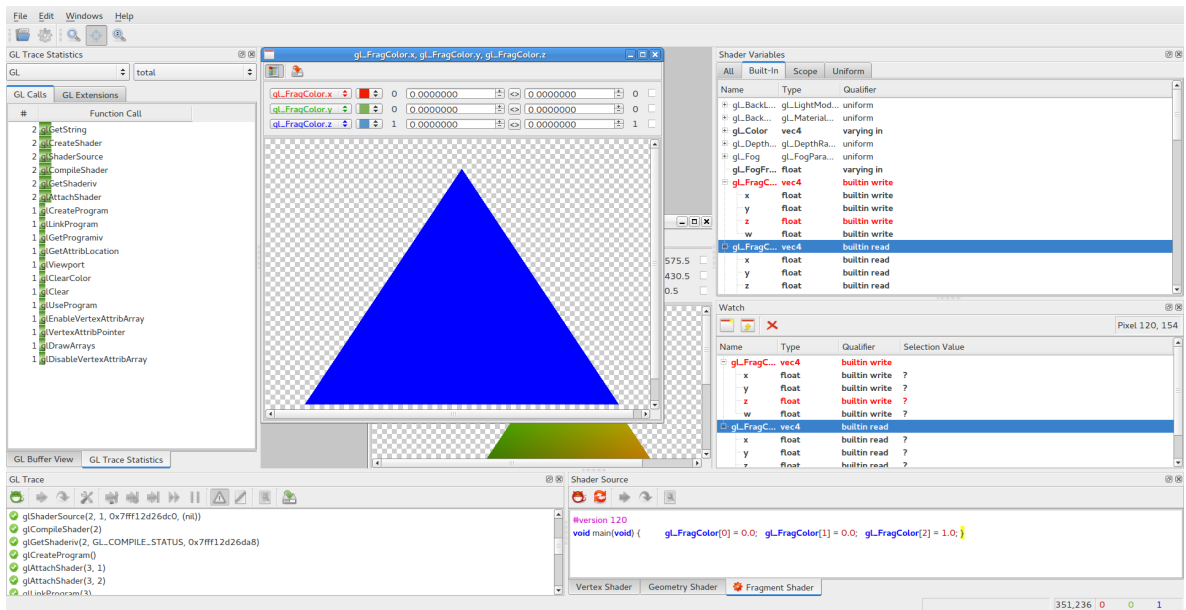


Figure 36.: Fragment color viewer.

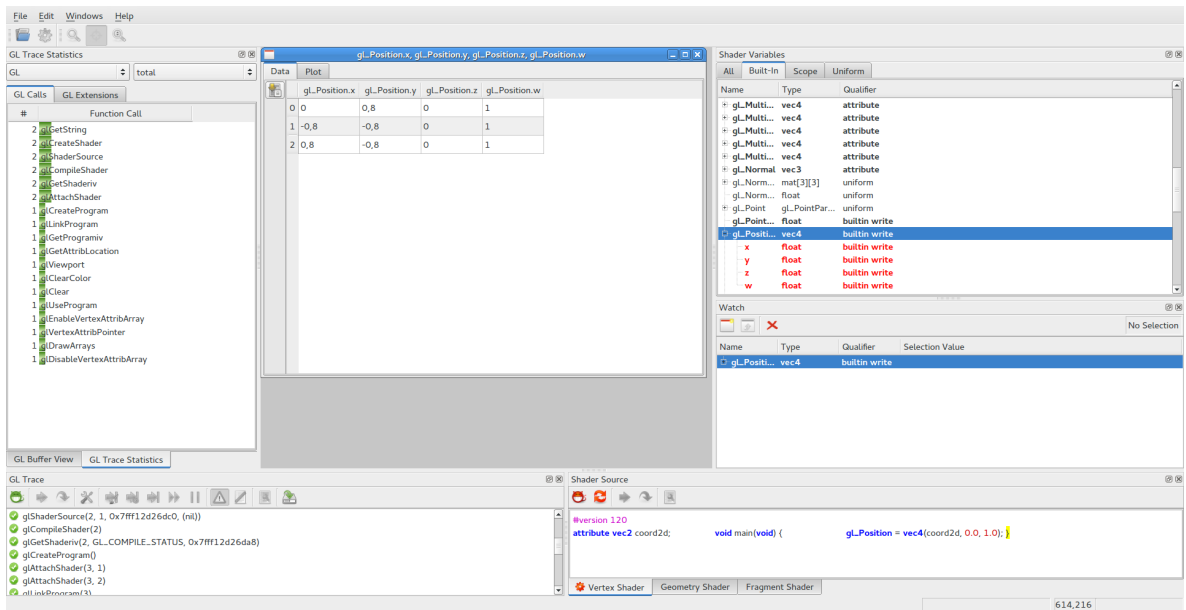


Figure 37.: Fragment position viewer.

B.5 VOGL

B.5.1 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's 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's 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 38. 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.

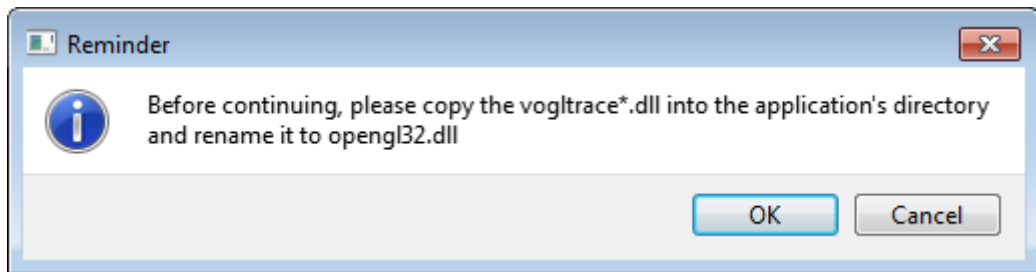


Figure 38.: VOGL editor reminder, occurs when attempting to trace an application.

B.5.2 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's 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's possible to see the framebuffers as shown to the left in figure 39, 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.

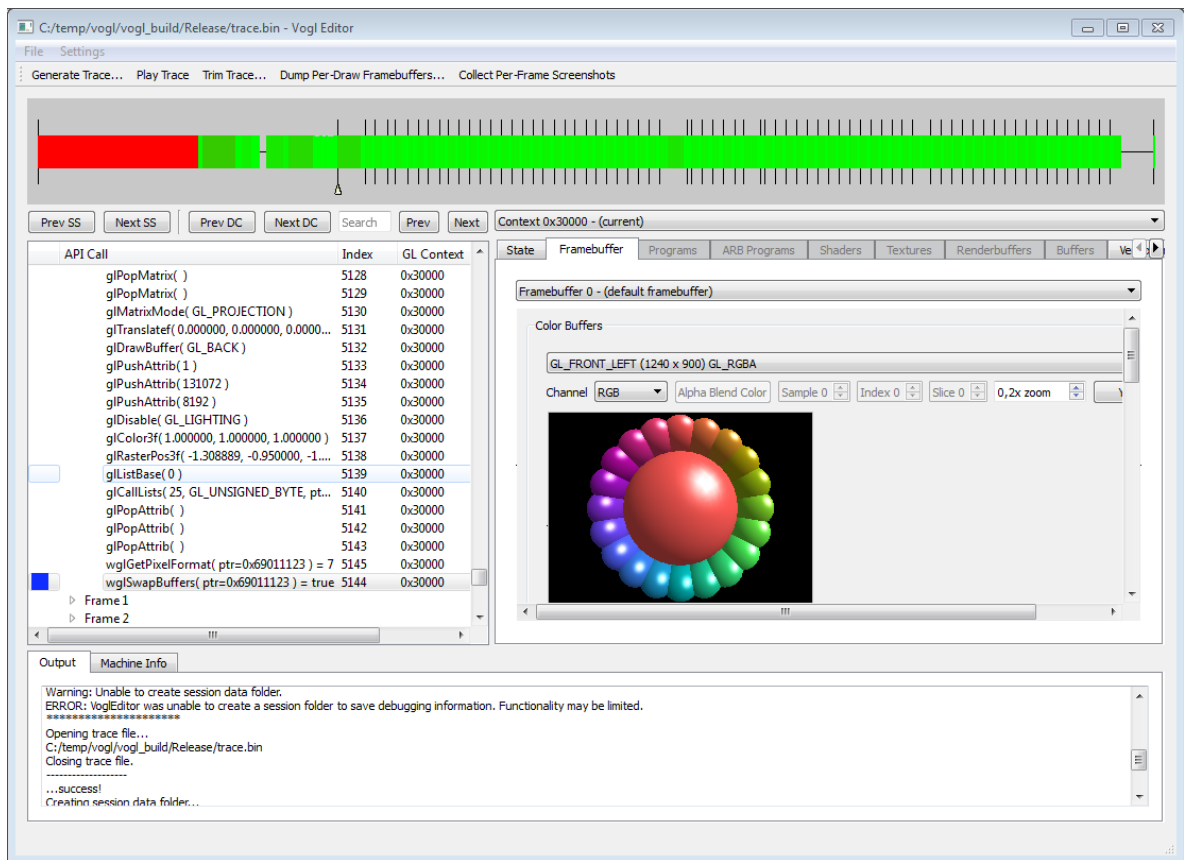


Figure 39.: VOGl editor after snapshot.

B.5.3 Creating the trace file

In order to use VOGl it's 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's 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 on the toolbar the `Generate Trace...` which will open a file opening dialog shown in figure 40, it's 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 38, this means that it's necessary to have the DLL wrapper in place when generating the trace file.

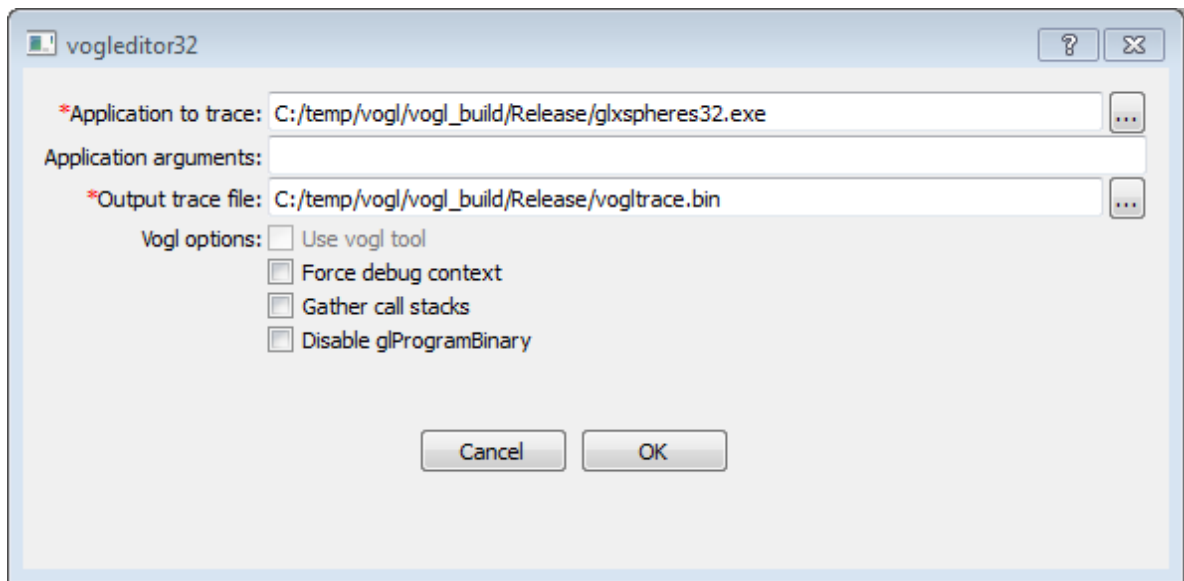


Figure 40.: VOGL editor generate trace.

The trace file is generated as soon the application closes.

B.5.4 *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's 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's 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.

B.5.5 *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's 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.

B.5.6 *Interactive replaying a trace file*

It's 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.

B.5.7 *Realtime editing and replaying a trace file*

It's possible to use VOGL to edit and replay a trace file at the same time, in order to do so it's 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's 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.

B.5.8 *Converting a APITrace trace file*

In order to convert and APITrace `.trace` file it's necessary to use APITrace's `glretrace <.trace> -benchmark` function, it means that to convert it's necessary 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`

B.5.9 *Dump images from a trace file*

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

It's also possible to dump frame buffer images per draw call from a trace file with `vogl32 replay <tracefile> -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.

B.5.10 *Get statistics from a trace file*

By using `vogl32 info <tracefile>` it's 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)

B.5.11 Finding in a trace file

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 `functionname`, it's 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's also possible to find parameters with `vogl32 find <tracefile> -find_param <paramname>`
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" : "0x0000000000017F800",
      "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)