Open Source Debuggers and Integration with a 3D Engine

Andre Alexandre Wang Liu   António Ramires Fernandes
Departamento de Informática
Universidade do Minho, Portugal
andre_liu507@hotmail.com, arf@di.uminho.pt

Abstract
Debugging is always an important phase in an application life. Applications using 3D hardware accelerated graphics in general, and OpenGL in particular, are usually hard to debug since the computation is split between two processors, each with its own memory space. To be able to debug these applications one has to rely on the API's debugging mechanisms to inspect the output of the computations performed. Several tools are available for this purpose, some of them are open source. Open source tools are great to integrate in already existing projects such as 3D engines since they allow the required customizations. The goal of this paper is twofold. Firstly we provide an overview of the existing open source debugging tools for OpenGL and secondly we discuss the integration of one of such tools in an existing 3D engine.

Keywords
OpenGL; Debugging; GLSL; 3D engine

1. INTRODUCTION

OpenGL 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 straight forward 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 [Khronos 14]. 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 vendors provides a debugging tool, at least for Windows operating system. NVIDIA released NSight [NVIDIA 14], and AMD has CodeXL [AMD 13]. Both debuggers can work integrated with Visual Studio. While the list of features is impressive, including shader code tracing, and GPU memory inspection, these tools are not up to date with the latest version of OpenGL. CodeXL claims to support OpenGL 4.3 while NSight only supports OpenGL 4.2. Furthermore, NVIDIA Optimus equipped laptops do not to take advantage of the full list of features available in NSight, namely shader code tracing.

Open source tools on the other hand are not as powerful as they don't allow shader tracing. 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.

This paper explores some of the available open source OpenGL debuggers (section 2). It also discusses the integration of one of such debuggers, GLIntercept, in an OpenGL based application (section 3). The selected application is the Nau 3D engine[1]. This engine allows for multi-pass rendering using OpenGL with shaders, and integrates NVIDIA's Optix ray tracing engine and Bullet physics en-

[1]https://github.com/Nau3D/nau

gine. It is a complex application covering a wide range of OpenGL functionality and therefore should provide a very rich case study. Conclusions are presented in section 4.

2. OPEN SOURCE DEBUGGERS

On the OpenGL wiki [OpenGL.org 12] four open source debuggers are listed:

- GLIntercept
- APITrace
- Bugle
- glslDevil

Recently Valve has released its own debugger, VOGL. Together these are the debuggers explored in this paper. Full sub sections are devoted to debuggers which support the latest OpenGL versions, namely: GLIntercept, APITrace and Bugle.

All these debuggers are based on a library that wraps OpenGL functions. This library replaces the OpenGL library as far as our application is concerned. When an OpenGL function is called, the application is actually calling the homonym function on the debugger library, enabling actions to be triggered before and after dispatching the call to the OpenGL library. The most common of these actions is logging function calls, and all the above mentioned debuggers perform this functionality.

Due to the open source nature of these tools there is no guarantee that future updates will be available when new OpenGL versions come out. Even if the team behind the tool is active there will be a period of time when the debugger is not fully updated. Some of these debuggers have a fallback mechanism to at least report the OpenGL function names called within an application. Nevertheless, for users wanting to try the latest features, it is important to have full support right away, or at least support for the features being tested.

In the next sub-sections some of the particular features of the debuggers will be discussed, namely their updatability.

2.1. GLIntercept

GLIntercept [Trebilco 13] is an OpenGL debugger originally designed for Windows, but a Linux port is available. The main features of interest for modern OpenGL users are:

- Full function logger: logs every OpenGL function call during the application life to a file;
- Frame logging: as an alternative to full frame logging it can selectively log only the function calls between two consecutive swap buffers calls;
- Dump textures and frame buffers to file;

The configuration of the debugger is based on text files and is performed manually. Nevertheless the syntax is simple and examples are provided.

2.1.1. Plugins

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 users to create them, and source code for several plugins is available. Some note worthy plugins for modern OpenGL are:

- Debug Context: Forces an OpenGL debug context and logs; `ARB_debug_output` and `GL_KHR_debug` messages;
- Shader Editor: Provides a text editor to edit and inspect shader code in real-time;
- Function Stats: Statistics regarding function calls;

Plugins allow to override the default functionality of GLIntercept by adding code to be executed before and after an OpenGL function call. It is also possible to define code to wrap render functions and the end of the frame.

Plugins are added to the default GLIntercept config file as follows:

```
  Plugins
  {
    FunctionStats = ("GLFuncStats\GLFuncStats.dll")
    {
        // plugin options
    }
}
```

The available options are provided in the template config file for each plugin.

2.1.2. Logging

GLIntercept works by overriding calls to `wglGetProcAddress`, wrapping the real function pointer with some code and returning it to the application being debugged. Since the usage of wglGetProcAddress is required by any application using post 1.1 OpenGL, this approach ensures that GLIntercept will log all functions. However, functions which are not defined in the header config files of GLIntercept will have their parameters represented by "???", for instance, assuming that `glGetProgramiv` is not defined the log would report `glGetProgramiv( ??? )`.

An example of a log snippet is:

```
...
glClear(GL_DEPTH_BUFFER_BIT |
        GL_STENCIL_BUFFER_BIT |
        GL_COLOR_BUFFER_BIT)
glDisable(GL_SCISSOR_TEST)
glUseProgram(7)
glUniformMatrix4fv(0,1,false,[1.732051,0.000000,
    0.000000,0.000000,0.000000,1.732051,0.000000,
    0.000000,0.000000,0.000000,-1.006018,-1.000000,
    0.000000,0.000000,9.458376,10.000000])
glUniform4fv(1,1,[0.000000,0.000000,1.000000,1.000000])
...
```

By default, return values are not logged in GLIntercept. This functionality could be partially covered with a plugin that traces `glGet*` functions. Yet, to fully implement such a feature it requires the plugin programmer to identify all GL functions that return meaningful values to the application.

2.1.3. Updating for newer OpenGL versions

As mentioned before, 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.

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

For a full update, GLIntercept has a script to use egl, gl, glx and wgl xml files from Khronos spec repository [2] 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. For instance in `glext.h` we can find:

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

And in the header configuration files we have

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

  GL_COMPUTE_SHADER_BIT              = 0x0020,

};
```

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

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

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

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

2.2. APITrace

APITrace [Fonseca 13] is a unique tool among this set of debuggers. When the application being debugged is running the debugger limits itself to log all calls producing a binary file with all the information. APITrace's main goal is to replay the log file once the application terminates.

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.

[2]https://cvs.khronos.org/svn/repos/ogl/trunk/doc/registry/public/api/

Although the log file is binary it can be edited using APITrace's tools allowing the user to resize the file or even change some input. Apitrace can read the frame uniforms and shaders by replaying the trace file, it replays each function until it reaches a user defined breakpoint (a user selected function) or the end of the file. Then it grants the user the ability to inspect the application state at that point. When replaying an application, APITrace provides a full GUI application that makes it very easy to use.

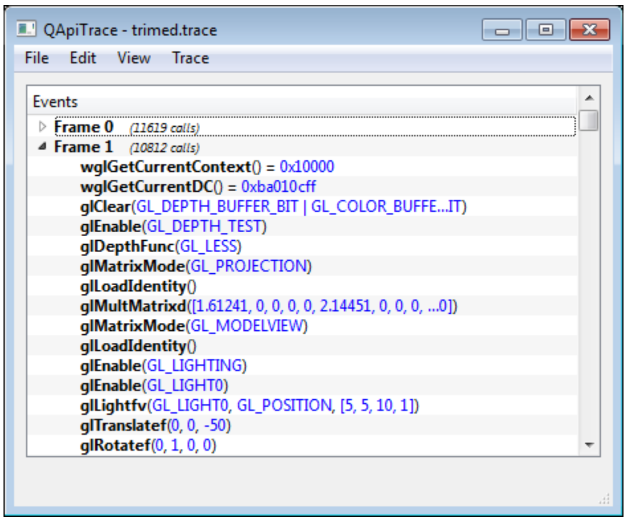
As far as logging goes, APITrace supports all OpenGL debugging extensions even if the extensions are not supported by the driver/hardware. In this case the extension functions themselves are not executed but they can be traced.

This debugger is also capable of profiling both CPU and GPU execution times for frames and draw calls.

APITrace has cross-platform compatibility allowing debugging for Linux, Windows, OSX, and Android. Besides OpenGL, APITrace can also work with Direct3D and OpenGLES. However the profiling features are only available to OpenGL.

2.2.1. APITrace GUI

When replaying a log file, APITrace provides a full GUI application that makes it very easy to use. A log viewing example is shown in figure 1.



```
Frame 0 (11619 calls)
Frame 1 (10812 calls)
    wglGetCurrentContext() = 0x10000
    wglGetCurrentDC() = 0xba010cff
    glClear(GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFE...IT)
    glEnable(GL_DEPTH_TEST)
    glDepthFunc(GL_LESS)
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    glMultMatrixd([1.61241, 0, 0, 0, 0, 2.14451, 0, 0, 0, ...0])
    glMatrixMode(GL_MODELVIEW)
    glLoadIdentity()
    glEnable(GL_LIGHTING)
    glEnable(GL_LIGHT0)
    glLightfv(GL_LIGHT0, GL_POSITION, [5, 5, 10, 1])
    glTranslatef(0, 0, -50)
    glRotatef(0, 1, 0, 0)
```

Figure 1. APITrace log viewer.

The GUI provides every bit of information regarding the application status for each draw call, uniform values, shader code, and even a graphical profiling tool.

2.2.2. Updating for newer OpenGL versions

APITrace suffers from similar problems as GLIntercept regarding the update to new OpenGL versions or extensions. Most of the process is automated, and it is based on the official OpenGL header files. Nevertheless, manual tuning is required for parameter types and some function signatures.

Again, while the update could be performed manually for a reduced number of functions, it is unfeasible for someone outside the development group to achieve a full significant OpenGL version update.

Plus, when such an update is performed, a rebuild is required.

2.3. Bugle

Bugle [Merry 07] is mostly used in Linux operative systems. It can also be installed in Windows but, as mentioned on the web site, ”it is significantly trickier than on UNIX-like systems, and currently only recommended for experts”.

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 that can be configured inside a chain. However, unlike GLIntercept plugins, no method other than changing the C code directly can create new filters. There are filters for tracing, collecting statistics, error logging, taking screen shoots, video capturing, amongst others. An example of a chain is as follows:

```
chain myshowstats
{
    filterset stats_basic
    filterset stats_primitives
    # Note: stats_fragments requires GL_ARB_occlusion_query
    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"
    }
}
```

This chain will compute several statistics that will be used by the filter `filterset showstats` to display on the debugged application’s interface according to the `show` and `graph` listed by the filter. The `filterset showstats` displays data according to the statistics configuration file, for example:

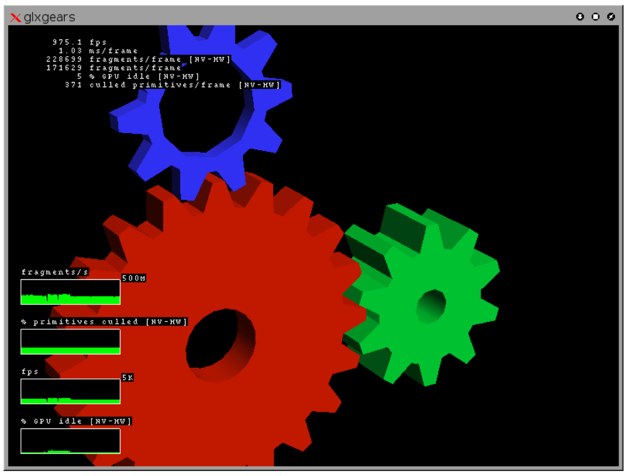
```
"frames per second" = d("frames") / d("seconds")
{
    precision 1
    label "fps"
}
```

2.3.1. Bugle GUI

Bugle displays statistics on top of the application being debugged (figure 2), as well as in its own GUI (figure 3). Bugle’s own GUI allows the user to set breakpoints on any OpenGL function and inspect the current OpenGL state, including buffers, textures, framebuffer contents and shaders.

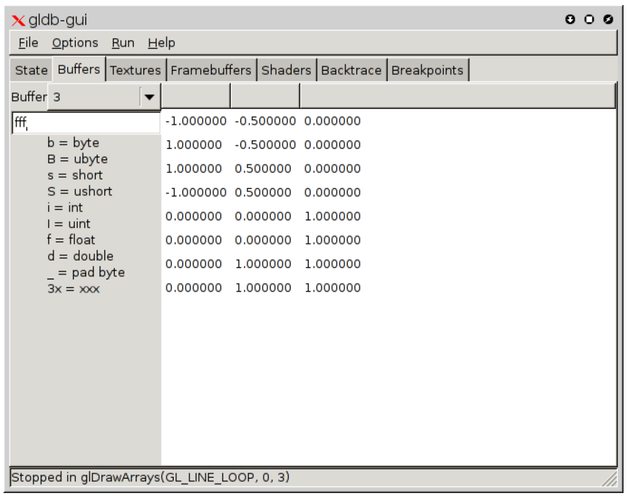
2.3.2. Updating for newer OpenGL versions

Bugle generates it’s API configuration files using the khronos-api xml files, which can be located in Khronos



```
   975.1 fps
    1.03 ms/frame
  228699 fragments/frame [NV-HW]
  171629 fragments/frame
       5 % GPU idle [NV-HW]
     371 culled primitives/frame [NV-HW]
```

Figure 2. Bugle stats on top of the application.


gldb-gui
File Options Run Help
State Buffers Textures Framebuffers Shaders Backtrace Breakpoints
Buffer 3

```
fff,          -1.000000 -0.500000 0.000000
  b = byte     1.000000 -0.500000 0.000000
  B = ubyte    1.000000  0.500000 0.000000
  s = short   -1.000000  0.500000 0.000000
  S = ushort   0.000000  0.000000 1.000000
  i = int      0.000000  0.000000 1.000000
  I = uint     0.000000  1.000000 1.000000
  f = float    0.000000  1.000000 1.000000
  d = double
  _ = pad byte
  3x = xxx

Stopped in glDrawArrays(GL_LINE_LOOP, 0, 3)
```

Figure 3. Bugle’s GUI.

spec repository, making it the most simple debugger to update. Updating is a fully automated process with no manual tuning required.

On the downside, once the API configuration files are updated, a rebuild is required.

2.4. Other Debuggers

GLSL-Debugger [Hanson 13], a descendant of glslDevil [Klein 10], is the only open-source debugger that provides GLSL tracing capabilities, based on Mesa[Mesa.org 99], an open-source implementation of the OpenGL specification. However, at the time of this writing Mesa only supports GLSL 3.3, hence, it may be of little use for cutting edge developers. This debugger has a complete GUI and a large number of features, but its dependency on Mesa makes it hard to keep up to date with the latest OpenGL versions.

A recent addition to the list of available debuggers is

VOGL [Software 14], Valve's OpenGL debugger. Still only supporting OpenGL 3.3, but with an interesting list of features that makes it worth while to monitor its development. Perhaps the most interesting feature is the ability to deal with very large trace files efficiently. OpenGL state management allows to efficiently start tracing from the middle of the trace file without having to replay the entire file, as required with APITrace. Trimming and editing of trace files is also considered an important feature and VOGL's developers are working hard on these tools.

2.5. Comparision table

The following table provides a short feature comparison between the debuggers that have full support for current OpenGL, namely: GLIntercept, APITrace and Bugle.

	GLIntercept	APItrace	Bugle
GUI	no	yes	yes
Log trace	yes	yes	yes
Trace step-by-step	no	no	yes
Replay trace	no	yes	no
OpenGL debug ext.	yes	yes	yes
Trace statistics	yes	no	yes
Runtime statistics	no	no	yes
Show extensions	no	no	yes
Video capturing	yes	yes	yes
Screenshooting	yes	yes	yes
Capture frame log	yes	yes	yes
Capture frame buffer	yes	yes	yes
Application profiling	no	yes	yes
Shader information	yes	yes	yes
Uniform data reading	no	yes	no
Plugin addition	yes	no	no
Man. tuning GL updates	yes	yes	no
Rebuild on GL updates	no	yes	yes
OS	Win	Win/Unix	Win+-/Unix

Table 1. Open Source Applications Feature table

3. INCORPORATING THE DEBUGGER IN NAU

Nau [Ramires] 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 C/C++ code.

Nau works with Optix [NVIDIA 09], from NVIDIA, enabling hybrid rendering algorithms in pipelines that contain both rasterization and ray-tracing passes.

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

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

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

Integrating a debugger into Nau is certainly a path to explore, and this section reports on the progress achieved so far. Being able to have access to the OpenGL calls log file in runtime, will probably save many headaches when writing code for Nau.

Since Composer already provides a GUI, and as both Bugle and APITrace also have a GUI, we opted for GLIntercept. Furthermore, its plugin architecture might reveal very useful in this integration scenario as no source code editing in the debugger is required. Finally, GLIntercept is the only debugger that does not require a rebuild for each OpenGL update, or new extension.

GLIntercept was designed to provide a large amount of information, not only for debugging but profiling as well. Since Nau already contains debugging mechanisms not all of GLIntercept features will be used as they are better covered inside Nau itself. For instance, if enabled GLIntercept profiler records CPU times for every OpenGL call which is not very useful since we are more interested in the GPU times. Nau records both CPU and GPU times per user defined block. GLIntercept also allows the recording of errors, however this occurs with a significant performance cost. We plan to add to Nau the most recent OpenGL debugging extension [Khronos 14]. A GLIntercept plugin is available for this purpose, but implementing it directly in Nau provides more flexibility.

3.1. Changes on GLIntercept

By default GLIntercept's behaviour can't be changed when the target application starts.Since the debugger is now integrated in the target application it makes sense to be the application controlling the logging facilities. Therefore, the log is no longer automatically created, it has to be started manually with a function on the target application. Also, the plugins must be activated within the target application.

For the target application to control the debugger it was required to allow to edit in runtime the debugger configuration settings. To achieve this we created a C module integrated in the debuggers library to expose the required functions.

Because the settings can now be edited in run-time several changes regarding removing and adding new settings also had to be made. For instance, plugins alter an internal function table, and now the ability to remove plugins implies methods to clean the function table in order to avoid crashing.

3.2. Changes in Nau

Everything in Nau can be defined in XML project and material definition files. Hence, it makes sense to follow the same principle regarding the debugging options. Therefore, the XML definition was extended to allow the configuration of GLIntercept's options and plugin usage. No source code editing is necessary to configure debugging options for a particular project.

The `<debug>` tag has been added in order to use the new features. On the absence of this tag, GLIntercept will not be used by Nau. This allows Nau not to use GLIntercept at all, so the new features are not forced on the user, and are disabled by default, causing no impact relative to the original performance.

3.3. Changes on composer

As mentioned before, Composer is a GUI for Nau. Debugging information such as the log of OpenGL calls can become a powerful debugging tool if provided frame by frame, or even at a finer grain such as pass by pass. To achieve this Composer was expanded to control rendering: pausing rendering, rendering pass by pass, or frame by frame.

Once paused, it will load the txt logfile and split the log in frames and passes. Due to the plugin architecture of GLIntercept it is feasible to generate more info in the future.

3.4. Creating a Project File for Debugging

To activate Nau's debugger all that is required is the addition of the `debug` tag on the project XML file. The children tags are optional and allow the override of the default settings for GLIntercept as defined in `gliconfig.ini`. Without any children or options the default settings will be used.

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 an unsigned int;
- ”string” is for text values.

The debug tag has a set of children as in the following code. The `glilog` attribute is optional, when ”false” it won't create the function log file.

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

3.4.1. functionlog

This will define logging options such as the location of the log file and if the number of frames to be logged should be limited to a certain value to prevent excessively long logs. All these options have default values in `gliconfig.ini`, so the definition of these tags is only required to override the default settings.

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

3.4.2. logperframe

This allows the user to select frames to log. If this tag is defined then the logging is not started as soon as the application starts. To start logging the user must press a key, or combination of keys, as defined in the children tags. To log only selected individual frames the tag `logoneframeonly` must have a value true. Otherwise, if `logmaxnumlogframes` is defined then only the defined number of frames will be logged. If none of the mention tags are defined then once logging is activated is keeps logging frames until the application is terminated.

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

It should be noted that if `logframekeys` has multiple children then a key combination is being defined, for example `<item value="ctrl"/><item value="f"/>` means that the user needs to press `<ctrl-f>` to create the snapshot.

3.4.3. imagelog

Image log determines if the logger will log textures, which textures and in which format. In this section we can configure which texture types to save, 1D, 2D, 3D and cube textures. In here we can also configure the format in which images are saved, `<imagesavepng>`, `<imagesavetga>`, and `<imagesavejpg>`, are the allowed save formats and `<imageicon>` determines the icon type. Note that if the configuration is set to log all frames this will imply that all the images in the application will be saved every frame. This feature is designed to function with logging of single frames for performance reasons.

```
<imagelog>
    <imagesave1d value="bool">
    <imagesave2d value="bool">
    <imagesave3d value="bool">
    <imagesavecube value="bool">
    <imagesavepng value = "bool">
    <imageicon>
        ...
    </imageicon>
</imagelog>
```

3.4.4. framelog

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

There is also the possibility of creating a movie with the frame buffer, including all the specified buffers. For the stencil buffer it is possible to specify a color table for stencil values, for instance if the stencil value is 1 the color could be #FF0000 (red).

```
<framelog>
    <framepostcolorsave value = "bool">
    <frameimageformat value="string"/>
    <framestencilcolors>
        <item value="uint"/>
        <item value="uint"/>
        ...
    </frameStencilColors>
    ...
    <frameicon>
        ...
    </frameicon>
    <framemovie>
        ...
    </framemovie>
</framelog>
```

3.4.5. plugins

Adding a plugin is slightly different on Nau because of the configuration format.

GLIntercept plugins require at least the plugin name and the location of its DLL. For example in `glicongif.ini` we could write:

```
FunctionStats = ("GLFuncStats.dll")
```

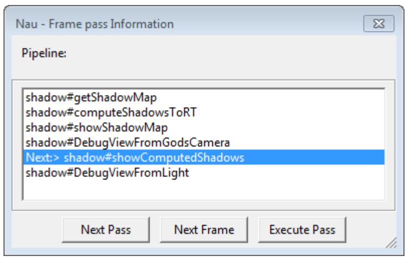
Some plugins can fit extra parameters and these extra parameters should be placed as described in the example, their format is the same as the GLIntercept's config file. These parameters can be found in the `config.ini` file of the plugin. Converting this information for a Nau project results in:

```
<plugins>
    <plugin name="FunctionStats"
        dll="GLFuncStats.dll">
        parameter1 = "parameter1 value";
        parameter2 = "parameter2 value";
        ...
    <plugin>
    ...
</plugins>
```

3.4.6. WORKING WITH COMPOSER

A debug menu was added to Composer. In order to access the debug options it is necessary to `Pause` Composer. Once paused the composer will start reading the GLIntercept main log file (if available) and fetch program data. This may take a while for large log files.

Once paused, the `GLI Log`, `Program Info` and `Buffer Info` options become available. Navigation options include executing the next pass, go until the end of the frame, or execute a full frame (see figure 4).



Nau - Frame pass Information

Pipeline:

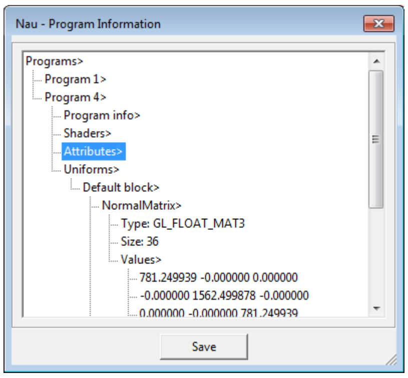
```
shadow#getShadowMap
shadow#computeShadowsToRT
shadow#showShadowMap
shadow#DebugViewFromGodsCamera
Next:> shadow#showComputedShadows
shadow#DebugViewFromLight
```

Figure 4. Nau's pass control.

The log window shows the function call log and stats per frame, see figure 7.

`Program Info` will provide information for each program including the actual uniform values, see figure 6. Composer already provides the values that uniforms *should* have according to the project file. Having the actual values allows developers to check if the uniforms are correctly being set.

The buffer information window allows the visualization of buffer contents. The user may specify the number of atomic elements and their type per struct of the buffer.



Nau - Program Information

```
Programs>
  Program 1>
  Program 4>
    Program info>
    Shaders>
    Attributes>
    Uniforms>
      Default block>
        NormalMatrix>
          Type: GL_FLOAT_MAT3
          Size: 36
          Values>
            781.249939 -0.000000 0.000000
            -0.000000 1562.499878 -0.000000
            0.000000 -0.000000 781.249939
```

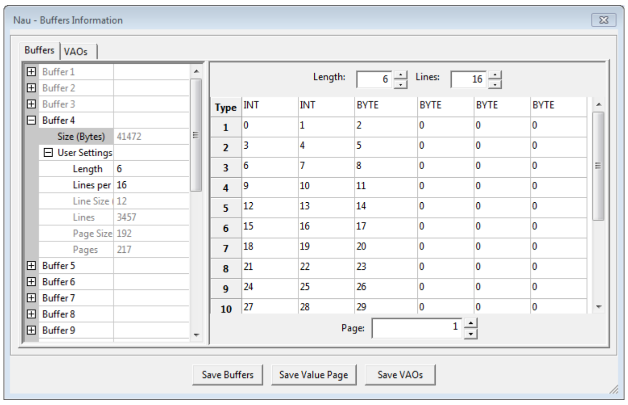
Figure 5. Nau's program uniforms.

4. CONCLUSIONS AND FUTURE WORK

Current Open Source debuggers, while not covering all desired features, can complement each other. While there is still a lot of work ahead, the debuggers presented in here are already very useful tools for OpenGL developers.

Nevertheless, actual shader debugging remains an issue. Proprietary debuggers like NSight or CodeXL, and GLSL Debugger are the only debuggers capable of this feature and even them are not fully up to date with the latest OpenGL version and extensions. A software implementation by Khronos could help to solve this issue.

Integrating GLIntercept with Nau turned out to be a much more simple task than expected. GLIntercept's code itself is simple to read once the programmer knows where


Nau - Buffers Information

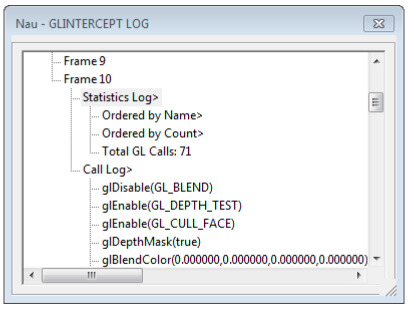
Buffer 1	
Buffer 2	
Buffer 3	
Buffer 4	
Size (Bytes)	41472
User Settings	
Length	6
Lines per	16
Line Size (	12
Lines	3457
Page Size	192
Pages	217
Buffer 5	
Buffer 6	
Buffer 7	
Buffer 8	
Buffer 9	

Length: 6 Lines: 16

Type	INT	INT	BYTE	BYTE	BYTE	BYTE
1	0	1	2	0	0	0
2	3	4	5	0	0	0
3	6	7	8	0	0	0
4	9	10	11	0	0	0
5	12	13	14	0	0	0
6	15	16	17	0	0	0
7	18	19	20	0	0	0
8	21	22	23	0	0	0
9	24	25	26	0	0	0
10	27	28	29	0	0	0

Page: 1

Figure 6. Nau's buffer information.


Nau - GLINTERCEPT LOG

```
Frame 9
Frame 10
    Statistics Log>
        Ordered by Name>
        Ordered by Count>
        Total GL Calls: 71
    Call Log>
        glDisable(GL_BLEND)
        glEnable(GL_DEPTH_TEST)
        glEnable(GL_CULL_FACE)
        glDepthMask(true)
        glBlendColor(0.000000,0.000000,0.000000,0.000000)
```

Figure 7. Nau's log information.

to start looking and all that was required was to export the configuration functions.

On Nau's side, due to the highly modular architecture of the 3D engine, it was painless to introduce all the new features. Although we have implemented a substantial set of features, much remains to be done to achieve a really helpful debugging environment. For instance, as we know which values the GL uniforms *should* have, and the values actually set on those uniforms, an automatic verification matching procedure is a goal to be achieved.

State inspection is another feature we are working on. The goal is to be able to configure which OpenGL state variables are relevant and to be able to inspect them when the application is paused.

Regarding 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.

5. ACKNOWLEDGEMENTS

Work partially funded by National Funds through FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project PEst-OE/EEI/UI0752/2014

References

[AMD 13] AMD. Codexl, November 2013. [Online - accessed 1 August 2014].

[Fonseca 13] José Fonseca. Apitrace. GitHub, November 2013. [Online - accessed 1 August 2014].

[Hanson 13] Hanson e Chris 'Xenon'. Glsl-debugger, October 2013. [Online - accessed 1 August 2014].

[Khronos 14] Khronos. Khr_debug extension spec, March 2014.

[Klein 10] Thomas Klein, Magnus Strengert, e Thomas Ertl. Glsldevil - opengl glsl debugger. GlslDevil. Http://www.vis.uni-stuttgart.de, February 2010. [Online - accessed 1 August 2014].

[Merry 07] Bruce Merry. Opengl software development kit. BuGLe. OpenGL, 2007. [Online - accessed 1 August 2014].

[Mesa.org 99] Mesa.org. Mesa 3d graphics library, January 1999. [Online - accessed 1 August 2014].

[NVIDIA 09] NVIDIA. Nvidia optix ray tracing engine, 2009. [Online - accessed 1 August 2014].

[NVIDIA 14] NVIDIA. Nvidia nsight visual studio edition. NVIDIA Nsight Visual Studio Edition. NVIDIA, 2014. [Online - accessed 1 August 2014].

[OpenGL.org 12] OpenGL.org. Debugging tools. OpenGL.org, March 2012. [Online - accessed 1 August 2014].

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

[Software 14] Valve Software. Valve opengl debugger, May 2014. [Online - accessed 1 August 2014].

[Trebilco 13] Damian Trebilco. Glintercept - opengl call nterceptor/logger. GlIntercept. Code.google.com, June 2013. [Online - accessed 1 August 2014].