Debugging Shader Programs

Arthur Préal
Faculty of Science)
University of Antwerp)
Antwerp, Belgium
arthur.preal@student.uantwerpen.be

***Abstract*—In the rapid evolving landscape of computer graphics, shader programs and shading languages play a vital role. They are used to create immersive and beautiful scenes that can be seen on a display. As is normal with other programs, bugs are likely to be written in these shader programs during development. Other languages have tools to help developers find these bugs quicker than inspecting source code. An example of such a tool is the debugger. A debugger is a program that is used to inspect another program's execution. It allows the programmer to step through the code and see the values of variables at certain points in the program. Last year, a new shading language (SRSL) was developed for the first research project that solved the code duplication issue that comes from supporting multiple graphics APIs (and thus multiple shading languages). This research project builds further on the language toolchain developed in the first research project by creating a debugger for the SRSL language.**

I. Introduction

This paper consists of a few different sections. In the first section, some terminology and important concepts will be explained. After that, some difficulties will be mentioned that comes with debugging shading languages followed by a section explaining the approach and architecture of the debugger. Also some interesting existing projects will be mentioned. Lastly, a small example will be given together with the workflow.

II. Definitions & Terminology

A. The GPU

The Graphics Processing Unit or GPU is a dedicated hardware circuit designed to render 3D computer graphics [12]. These devices are frequently much faster at rendering tasks than traditional CPUs, this is mainly due to their parallel architecture. Because of this advantage, the GPU recently saw a lot more usage outside the computer graphics field, mainly in the Machine Learning and Artificial Intelligence fields. Its architecture mainly consists of many small cores or execution units (somewhat comparable to the cores found in a CPU) together with more specialized hardware such as rasterization and texture mapping circuits. Developers can use this device through a graphics API.

B. The Graphics API

A graphics API is an Application Programming Interface used to send commands to the GPU and manage the different resources found in GPU memory (also called video RAM). It provides a uniform way of communication between an application and any (supported) GPU. This communication is done through the GPU driver which is tied to a specific GPU device. The different API commands range from controlling the graphics pipeline to managing different regions of GPU memory. The driver handles low level details of this management. There are a lot of different APIs but for the personal computer, the following three graphics APIs are the most common.

- OpenGL [9]
- Direct3D [16]
- Vulkan [10]

More obscure platforms such as gaming consoles have their own graphics APIs.

1) OpenGL: From the three mentioned APIs, OpenGL is by far the oldest one since it has been around since 1992. The Khronos Group is the main group for maintaining OpenGL although development is stopped in favor of Vulkan. The most recent version is version 4.6 which was released in 2017. OpenGL is designed in such a way that the driver takes a lot of the low level responsibility away from the programmer. This has the advantage that it is easy (under 100 lines of code) to get something to draw to the screen. But this approach has the drawback that the driver has to 'guess' what you are trying to do and therefore may not be as performant as other APIs. Because OpenGL is a mature and older API, it is supported on a wide range of different platforms. Since development has stopped in 2017, the latest advancements in GPU architecture such as dedicated ray tracing cores are not supported by this API.

2) Direct3D: Direct3D is part of the DirectX toolkit developed by Microsoft and tightly integrated into the Windows operation system. The current version is Direct3D 12.1 but in this paper, only version 11 is considered as it is still widely in use today. Thanks to close integration in the Windows operating system, this API is the most performant API on this platform. But the downside is that this API is only supported on Microsoft products (Windows and the Xbox gaming console.

C. Vulkan

Vulkan is also developed by the Khronos Group and is considered the successor to OpenGL, the codename for this API during development was GLNext. It was intended to

address the shortcomings of OpenGL, mainly the ability to fine tune the GPU to a specific application. The first release (version 1.0) was released in 2016, the most current version is 1.3. Vulkan is considerably more difficult to get something to draw (more than 1000 lines of code needed). This is because Vulkan is designed with minimal driver overhead which means that a lot of things that the OpenGL driver does for the programmer, must now be managed manually. The advantage from this decision is that experienced graphics programmers can fine tune the GPU to their specific application which can increase performance by a large amount.

D. Shaders

In 2004, an advancement was made in GPU architecture that would change the landscape forever. It became possible to write custom programs that can be executed by the execution units found on the GPU. These programs are also called shaders and are written in a shading language. There are a lot of different shading languages as nearly every graphics API defines their own. For the three considered APIs, GLSL is used by OpenGL and Vulkan (note that there is a distinction between OpenGL GLSL and Vulkan GLSL) and HLSL, used by Direct3D. These languages are based on the C programming language but extend the C syntax with keywords that let programmers access the other objects and constructs provided by GPUs. Examples of these constructs are 4D linear algebra math operations and texture sampling. A shader consists of multiple shader modules. There are a few different shader module types but in this paper, the focus is placed on the two most important ones. First, the vertex shader processes vertices and the fragment shader processes individual pixels. These two are the only ones needed to construct a valid shader program.

E. The graphics pipeline

The graphics pipeline is the collection of different GPU states and commands that specifies how the GPU must render an object. This can be seen as a large state machine. A visual representation can be seen in figure 1. Figure 1 has two distinct parts. First, there is the pipeline itself that describes how a set of input data. The second part is the different objects that reside in GPU memory. A few of these objects are:

- Constant buffers: Read only data located in video RAM that can be accessed by the different shader stages.
- Textures: Data that represents an image, can be accessed by the fragment shader.
- Samplers: Objects that specify how a texture should be mapped onto a surface.

The pipeline itself begins withe the input assembler stage. This stage will fetch the vertex data of the object from memory and pass it to the vertex shader stage. This stage executes the vertex shader module on vertex data of the input mesh. It transforms the mesh in the world and projects it to the screen. The processed vertices are then passed to the rasterizer who converts these vertices to screen coordinates or pixels. In this stage, the screen coordinates are given to the fragment

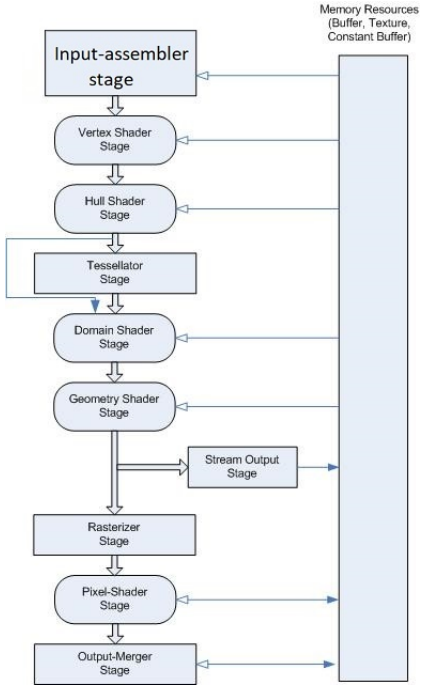

Fig. 1. The Graphics Pipeline

shader who will determine the color of the corresponding pixel. This set of pixels is then passed to the last stage: the Output Merger state which determines which pixels are part of the screen space and which pixels are not. There are a lot of other stages as well but these are optional. An example is the depth testing stage which is used to determine if a fragment is behind another fragment.

F. The Debugger

A debugger is a computer program used to test and debug other programs (also called the target program). This program allows developers to see what the target program is doing at a certain point and view the state of the program. Features include executing instructions step by step, displaying the contents of different variables and changing these values directly when the target program is running. It has become an essential tool for programmers to help them find why a program in development is not behaving as expected. An example is the GNU Project Debugger or GDB [2]. It supports multiple programming languages such as C, C++, Fortran and Rust.

III. The SRSL Toolchain

The SRSL language is a shading language developed in the research project upon which this research is built on [23]. This language is a solution to the problems that come from supporting multiple graphics APIs. This research extends the usability of this language by providing a debugger.

A. Supporting Multiple APIs

The first research project states that when developing a graphics intensive application that needs to run on a variety

of platforms, multiple graphics APIs must be supported by that application. While OpenGL is supported on a lot of platforms, it comes with its own drawbacks such as performance (in comparison to newer APIs such as Vulkan and Direct3D 12) and missing features like the ability to control dedicated ray tracing hardware. Example programs who fall under this category of applications are The Unreal Engine [7] and Blender [3]. In particular, the Unreal Engine runs on a lot of different platforms including Windows, UNIX, Nintendo Switch, PlayStation and Xbox. Each of these platforms has their own set of graphics APIs. It is expected that running this application always results in the same output regarding which platform it is executed on. To achieve this, they configure each (supported) graphics API similary, this includes writing the same shader logic for each shading language. This results in a lot of code duplication which makes it hard to maintain large softwares such as the Unreal Engine. They solved this issue by sticking with a selected language (the HLSL language) and they developed a compiler that compiles HLSL to other languages including GLSL. The main problem with their approach is that this compiler is tightly integrated into their specific application and cannot be easily used by other programs. This is where SRSL comes in.

B. SRSL

The SRSL language is a shading language based on the C programming language and other shading languages such as GLSL and HLSL. This language was designed in the first research project and has a compiler ready that can export SRSL to GLSL and HLSL. The compiler is designed in such a way that it is easy to integrate in any project as a standalone library. While GLSL and HLSL are also used in general GPU compute, SRSL is specifically designed for 3D computer graphics and is therefore considered a domain specific language. The language is still relatively new and is developed by a single individual. Therefore not every aspect is supported in SRSL but it fully supports vertex and fragment shader modules. Besides only compiling SRSL, the compiler also performs some validation checks and the programmer can specify certain properties. For example, depending on the GPU hardware, the number of vertex attributes may be different and SRSL allows for constraining this number. Another constraint is placed on the maximum size that a constant buffer can be.

C. Example SRSL Program

A shader program in SRSL is valid if both the vertex shader and fragment shader modules are valid. As an extra constraint, they must also be compatible with each other meaning that the output of the vertex shader must be the same as the input for the fragment shader. The first line of a shader module must always be the shader type declaration. Explicitly stating this type helps the compiler verifying shader module specific checks. An example of this is the 'discard' keyword. This keyword halts shader execution and can only be used by the fragment shader. In this section, a small example will be explained. This basic example uses all important constructs such as constant buffers, textures, samplers and performs some basic 4D linear algebra math operations.

Listing 1. SRSL Example Vertex Shader

```
ShaderType = Vertex;

Input vsIn{
    float3 position: Position;
    float3 color: Color;
    float2 texCoords: TexCoord;
};

Output vsOut{
    float4 position: SRV_POSITION;
    float4 color: Color;
    float2 texCoords: TexCoord;
};

ConstantBuffer(slot = 0) transformationData{
    float4x4 world;
};

void main(){
    float4 extPos = float4(vsIn.position, 1.0);
    float4x4 copied = transformationData.world;
    vsOut.position = copied * extPos;
    vsOut.color = float4(vsIn.color, 1.0);
    vsOut.texCoords = vsIn.texCoords;
}
```

Listing 1 is the vertex shader module. The shader module expects the incoming vertices to have the following layout:

1) Position: A 3 dimensional coordinate
2) Color: A RGB color value
3) TexCoord: a 2 dimensional texture coordinate

A single constant buffer called transformationData is declared which holds a 4x4 matrix called world. This shader outputs a transformed vertex together with the color attribute and texture coordinates. The main function begins by extending the 3 component position attribute to 4 components and multiplying with the transformation matrix. The color attribute is also extended to 4 components and the texture coordinates are simply copied over.

Listing 2. SRSL Example Fragment Shader

```
ShaderType = Fragment;

Input fsIn{
    float4 position: SRV_POSITION;
    float4 color: Color;
    float2 texCoords: TexCoords;
};

Output fsOut{
    float4 color: SRV_TARGET_0;
};
```

```
Texture2D(slot = 0) albedo;
Sampler(slot = 0) sampler0;

void main() {
    float4 texel = sampleTexture(
        albedo,
        sampler0,
         fsIn.texCoords
         );
    if (texel.a < 0.1f){
        discard;
    }
    fsOut.color = mix(fsIn.color,
                      texel,
                      0.5);
}
```

Listing 2 represents the fragment shader module. It takes in the attributes outputted by the vertex shader and interpolated by the rasterizer. This fragment shader also takes in a texture found at slot 0 and a sampler, also found at slot 0. The main function begins by sampling from the provided texture and uses the texture coordinate attribute. The way this sampling is done is determined by the declared sampler. After that, the texel is checked if it is transparent or not. If it is transparent, the pixel is discarded and the fragment shader stops for this current pixel invocation. If it is not transparent, the resulting color is a mix between the texel and the incoming color attribute.

Although this example is small, it still uses all important constructs. This example will be used to develop the debugger.

D. Why developing a debugger?

In 2010, a paper was written on a plugin for the Eclipse [4] IDE. This paper, titled The Spoofax Language Workbench [15] goes over the difficulties that come from developing text based Domain Specific Languages. It states that for a DSL to be easily usable, some quality tools must be available for creating programs in that DSL. This paper in particular focuses on generating IDE services based on the newly developed DSL. These IDE services are services such as automatic indentation, bracket insertion, reference resolving and content completion. This research project closely matches the intuition behind the Spoofax Language Workbench paper but focuses on a different tool. Having a debugger that allows developers to see what their programs are doing at a specific point in time while being in a specific state is invaluable to find and solve bugs. So with this second research project, the toolset for the SRSL language (developed during the first research project) is expanded upon. This research project is done concurrently with a Master Thesis in which another invaluable tool is developed, a testing framework. Testing frameworks for shading languages are seemingly nonexistent. The SRSL language is used to create a testing framework that supports auto generation of test cases. This project further extends the toolset available for the SRSL language.

E. Other Tools

There are of course a multitude of different tools that can be used to view and debug the graphics pipeline. In this section, a few interesting tools will be mentioned, mainly regarding shading programs.

1) RenderDoc: RenderDoc [1] is a stand-alone graphics debugger that allows developers to capture a frame of their application and inspect the GPU device and the GPU pipeline at that point in time. It is useful to see the contents of different buffers (constant buffers, vertex buffers, index buffers, ...) and textures (frame buffers, color attachments, depth buffers, ...) and see the different commands that the GPU executes during that frame. Errors destinated from the graphics API are also reported here. It (currently) supports the following APIs across Windows, Linux, Android and Nintendo Switch.

- OpenGL (and OpenGL ES)
- Direct3D (D3D11 and D3D12)
- Vulkan

Besides viewing pipeline states, it can also be used to debug HLSL shaders, but GLSL shader debugging is not supported. RenderDoc supports viewing variables and stepping through the code.

2) Microsoft PIX: Microsoft PIX [11] is a tool similar to RenderDoc but optimized for the DirectX family of graphics APIs, it therefore only runs on the Windows operating system. it also supports shader debugging but only for the DirectX APIs. PIX is generally more used for performance analysis instead of inspecting the GPU pipeline. It provides a lot of useful timing information for this purpose.

3) Shadertoy: Shadertoy [24] is a web service that allows shaders to be executed in the browser. While it does not support debugging features like inspecting variables and stepping through the different statements, it can be handy to quickly test your fragment shaders. This introduces a further drawback of using Shadertoy as some kind of debugging tool, it only supports fragment shader execution. So vertex shader debugging is not possible with this tool.

IV. Approach

Creating a debugger for a shading language is not as straightforward as it may sound. A few challenges surfaced when developing the SRSL Debugger. The first problem tackled was to find a way to execute the SRSL code in such a way that attaching/creating a debugger won't be too difficult.

A. The GPU Architecture and signals

A first problem is the GPU architecture itself. A traditional CPU features some systems that allow debuggers like GDB to function. Such a system are the CPU signals. Signals are standardized messages sent to a running program to trigger

specific behaviour. Most programmers will come across some signals such as SIGSEGV, meaning that the process made a segmentation fault. GDB uses signals to get information over the process it's attached to. SIGTRAP is used to stop the process at a certain point. This system works entirely different and the implementation varies between the different GPU vendors and architectures. Since this project is done individually, using GPU signals will be considered not feasible for this project although using these would be the preferred way.

B. Simulator Versus Emulator Versus Interpreter

Since the SRSL code won't be executed on the GPU itself, other ways must be considered to run the SRSL code (with the purpose of debugging in mind). The code would then be executed on the CPU which would not be as fast as on the GPU, but regarding the debugger, this is not an important concern (for now).

1) Interpreter: The first approach would be to write an interpreter that could execute the SRSL statements found in a program. Regarding SRSL, there are two valid options. The first one could be to develop a custom bytecode language and an accompanying interpreter. The existing SRSL compiler would receive an extra compilation target that is the newly designed custom bytecode. When a SRSL program is to be interpreted, the program would be given to the SRSL compiler which would be instructed to generate the corresponding bytecode. This bytecode can then serve as input for the bytecode interpreter. Since the SRSL compiler explicitly builds the abstract syntax tree (AST), the other approach would be to directly interpret the AST itself. This can be achieved to construct a tree walker that walks over the AST and execute the statements (or nodes) directly [13].

Bytecode is typically closer to machine code which allows for potential faster execution, but having another target to implement causes extra overhead. The debugging aspect would also be more difficult in comparison to the AST interpreter since some mapping logic must be available to map sections bytecode back to the original SRSL code. This is by far the main benefit of the AST interpreter since the SRSL statements are directly interpreted, there is no need for complex mapping logic. While the AST interpreter may not be as fast as the bytecode interpreter, it shines in its simplicity regarding development and execution.

Interpreting the SRSL language is a viable approach if the main interest was to provide a debuggable and runnable SRSL implementation. But since SRSL is designed specifically for 3D rendering, it makes sense that the debugger would also target this usecase. The input data for a SRSL shader would be something concrete. In case of the Vertex Shader, this would be actual vertex data and in case of the fragment shader, this would be fragments and their corresponding attributes. Creating meaningful data by hand can be rather difficult. It would be beneficial if the debugger would generate this data given a concrete input mesh.

2) Emulator: In order to debug a SRSL shader with meaningful data, a tool more powerful than an interpreter is needed. An emulator is a program that mimics the exact behaviour of a system. In most cases, an emulator is a software implementation of some actual hardware circuit. An example use case for emulators is mimicking the behaviour of old video game consoles, such as the Nintendo Entertainment System or NES [19]. In this case, the emulator is used to emulate the exact behaviour of the NES hardware when an input game is given. The important part here is that the emulation is exact, every line of code in the input program must be precisely executed.

In case of the debugger, the debugger would emulate the entire GPU. Not only the shader stages of the graphics pipeline are considered, but also stages like the rasterizer and input assembler are to be concerned. This solves the issue of using an interpreter for our debugger. If an input mesh is given, the data between the different shader stages is concrete and meaningful. But an emulator is a lot more work than an interpreter since not only a way for shader execution must be found, the other stages must be created as well.

3) Simulator: An emulator is needed when the exact execution matters such as in case of old console emulators where even the execution speed can have a huge impact. But for our debugger, this exact emulation is not necessary. The actual implementation of the other stages like the rasterizer and input assembler do not matter as the main interest is the shader execution. Even regarding shader execution, the main interest is to spot bugs in a SRSL program's logic. A simulator [25] tries to model the behaviour of a system, it aims to only replicate the essential characteristics of a real system. A simulator provides a more abstract view than an emulator In this case, the simulator would simulate the behaviour of the GPU, which is essentially to draw a given input mesh to a frame. How this process happens is not important as long as the resulting image is correct. Regarding our shader programs, this is the part of most interest; does our shaders generate the correct output? If not, use the debugger to try and see where possible logic errors are.

Comparing the three approaches, the simulator approach seems the most viable since it does not concern low level implementation details of the GPU and is able to generate meaningful input data. Writing a simulator is still more work than an interpreter, but won't be as difficult as an emulator.

C. Executing SRSL Code

In case of both the emulator and simulator, a way to execute the SRSL code is still to be found. With the main benefit of a simulator in mind, the main interest is the behaviour

of the SRSL code and not the exact execution. One of the characteristics of SRSL is that it takes a lot of inspiration from the C programming language (also from GLSL and HLSL but these are also inspired by C). With the exception of the linear algebra aspect, converting SRSL to C should not be difficult. The SRSL compiler needs to be extended such that it can convert the abstract syntax tree to C. Another benefit of using C is that a C debugger such as GDB can be used to debug SRSL programs. This makes the project a bit easier as the only thing left is to create a line mapping from SRSL to C.

1) GLM: To solve the linear algebra math problem, existing solutions or libraries can be used. A noteworthy example is the OpenGL Mathematics library [5] [6]. GLM is a header only C++ library based on the official OpenGL GLSL specification. This last part is important regarding memory layout of the matrices (column-major) and vectors, random numbers, data packing, ... While it is designed with the official GLSL specification [14] in mind, it works perfectly for other use cases. The provided classes and functions are nearly all similar to their GLSL counterparts. Another benefit of GLM is that it can take advantage of SIMD instructions which would improve the performance of the simulator. Since this library is written in C++, the SRSL compiler will compile SRSL to C++ instead of C.

D. The Plan

To make this approach concrete, the following steps are identified. The first step is to extend the existing SRSL compiler such that the SRSL code can be converted to C++. This C++ code relies on GLM to carry out the linear algebra operations. Using g++, this code can be converted into machine code together with a runtime environment. This runtime will offer both an API similar to OpenGL and D3D11 and an implementation such that can execute the different API commands and shaders. This API will be structured as a library that will be used by the simulator itself. This simulator uses the library to load different meshes and call GDB to debug the shaders. Lastly, another small modification will be made to the SRSL compiler such that it can output a file containing line mappings from SRSL to C++.

V. Architecture

Figure 2 represents a high level view of all the different components of the Simulator. The SRSL Compiler relies on the ANTLR4 [21] library to generate the parser. The SRSL Graphics API is a graphics API inspired by OpenGL and D3D11. it is used to send commands to the Graphics API Implementation which is implements the algorithms found in GPU hardware. The shader implementation can dynamically compile and link SRSL shaders. It first calls the SRSL compiler to compile the input SRSL program to C++. The output in C++ is designed such that both the vertex and fragment shaders forms a small library. g++ Is used to convert C++ into an actual DLL library while linking GLM to this library. Calling g++ is done using the WIN32 API

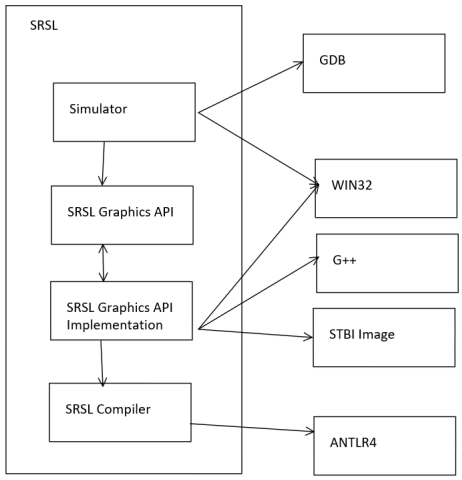

Fig. 2. SRSL Simulator Architecture

which is also used to load the DLL at runtime. STBI_Image is used for loading images into simulated texture units. The Simulator uses the SRSL Graphics API to create example scenes. It also uses the WIN32 API to attach GDB as a subprocess and send different commands to the GDB process.

Note that SRSL Simulator [22] is designed to work under windows. The WIN32 API is heavily used together with some other platform specific details such as DLLs and the STDCALL calling convention.

A. SRSL Compiler

Adding C++ as a new target in the SRSL compiler is straight forward. The SRSL compiler is designed in such a way that it is relatively easy to add a new target language that is similar to C. Compiling is done by walking over the Abstract Syntax Tree with a Writer Object. This writer object contains a set of string buffers that store the generated code in memory. When the tree is fully converted, the writer object can be instructed to either write the program to disk or to simply return it as an std::string. A lot of constructs are not different from the GLSL/HLSL targets and can be reused. These constructs include:

- For loops, while loops
- if, else and else if statements
- function declarations and function calls, except for the main function.

The constructs that have to be changed are:

- Type conversion: From SRSL to C++ (GLM)
- Intrinsic Functions
- Main function.
- Shader Interface
- Textures, Samplers and Constant Buffers.

1) Type Conversion: The abstract declaration of the Writer object contains a pure virtual function that converts the SRSL

type to the target language type. The C++ writer implements this function. For base types, the following mapping is used.

- void → void
- bool → uint32_t
- uint → uint32_t
- int → int32_t
- half → float
- float → float
- double → double

Note that boolean types are mapped onto 32bit unsigned integer types. on most GPUs, boolean values are 32 bit wide. Since the C++ bool type is 8 bits wide, this could potentially lead to weird memory errors. Using an unsigned integer will prevent this from happening. A point of issue is the half type. Since there is no native 16 bit floating point type in C++, this is mapped onto a 32 bit float type which could potentially cause some issues. It is therefore not recommended to use this type.

Scalar types are handled by GLM. GLM provides templates for these types, the syntax can be seen in listing 3.

Listing 3. GLM Scalar Types

```
// Vectors
glm::vec<NR_COLUMNS, BASE_TYPE>
// Matrices
glm::mat<NR_ROWS, NR_COLUMNS, BASE_TYPE>
```

2) Shader Interfaces: Shader interfaces are special blocks that describe the input data and the output data layouts. In case of the vertex shader, it describes the attributes of a single vertex and in case of the fragment shader, it describes a single fragment. The output C++ code is exported in such a way that it does not depend on any external dependencies (except for GLM). The input and output of a shader is represented by an unordered map, that maps a std::string to a byte array. When a variable declared in the interface is used throughout the program, the interface block layout is used to cast the byte array into the correct type. Listing 4 represents a simple SRSL vertex shader that both reads and writes from a shader interface block, while also extending the number of components in the scalar type.

Listing 4. SRSL Interface Block

```
ShaderType = Vertex;

Input vsIn{
    float2 position: Position;
};

Output vsOut{
    float4 position: SRV_POSITION;
};

void main(){
    vsOut.position = float4(
        vsIn.position, 1.0f
    );
}
```

This reading and writing results in the following C++ code, found in listing 5.

Listing 5. C++ Interface Block Translation

```
vsOut["SRV_POSITION"] = glm::vec<4, float>(
    *reinterpret_cast<glm::vec<2, float>*>(
        vsIn["Position"]
    ),0.0f,  1.0f
);
```

This code may seem a bit cryptic at first, but the data just gets casted around a lot. The variable is first casted into the datatype described by the input block. Then the variable is extended from a 2 component vector to 4 components. lastly, it calls the 4 component vector construction from GLM to create a 4 component vector and writes this value to the output block. Note that when a variable is accessed from an interface block, it is not accessed with the variable name but with the semantic name. This semantic name is also used by other stages in the pipeline such as the input assembler and rasterizer. This system is heavily inspired by D3D11. The rasterizer works on the output of the vertex shader and should be able to always access some data such as the position. The semantic name for this is the SRV_POSITION name.

3) Intrinsic Functions: Intrinsic functions are considered the standard library for shading languages. They contain some functions that calculate some common mathematical operations. A few examples are:

- Lerp: Linear Interpolates a vector.
- normalize: Normalizes a vector.
- sin, cos and tan: Calculate the sine, cosine and tangent angles
- transpose: Transposes a matrix

Nearly all these functions are also implemented in GLM. Whenever a SRSL program uses an intrinsic function, it will be replaced with the GLM counterpart during C++ code generation. There are a few functions not supported by GLM such as determinant which calculates the determinant of a matrix. There is one important intrinsic function which is not directly implemented in GLM; sampleTexture. This function allows sampling from a texture given a sampler and some texture coordinates. This function will be handled in section V-A6.

4) Main function: The output C++ code is designed such that both shaders can be compiled into a dynamic library. But this requires some extra precaution when declaring the main function, listing 6 represents the main function of a vertex shader.

Listing 6. C++ Main function

```
extern "C" __declspec(dllexport)
std::unordered_map<std::string, glm::vec4>
```

```
__attribute__((stdcall)) main__Vertex(
std::unordered_map<std::string, char*>& vsIn){
    // rest of code
}
```

This long definition breaks down into the following parts:

- extern "C": Use C-style linkage to prevent name mangling, this makes loading the functions much easier.
- __declspec(dllexport): A Windows specific extensions used to indicate that this name should be exported from the DLL, this object's name will appear in the export table of the DLL.
- std::unordered_map¡std::string, glm::vec4¿: The output from the main function is an unordered map of 4 component vectors.
- __attribute__((stdcall)) : Use the STDCALL calling convention, this is the standard calling convention under Windows.
- main__Vertex: The name of the main function. The shader type is always added to the name as simply calling the function 'main' conflicts if there are multiple shader modules in the same DLL.
- std::unordered_map¡std::string, char*¿& vsIn: The input for the main function, used when a variable declared in an interface is accessed V-A2.

5) Constant Buffers: Constant buffers are easily converted as they simply describe a structure of data with a binding point (or slot) attached. Converting this construct to C++ results in a struct definition together with a static global pointer to the name of that structure. Constant buffer data can be accessed globally throughout the program. Using a global static variable will mimic this behaviour. To fill these pointers with actual buffer data, the main function has an extra parameter called __CBUFFERS__ which is a vector of byte arrays. The constant buffer slot is used as an index into this vector. The corresponding byte array is then casted using the struct declaration. This cast operation happens in the main function before any other statements. Accessing data stored in a constant buffer is replaced with the global pointer, followed by the $->$ operator and then the member variable.

Listing 7. SRSL Constant Buffer

```
ConstantBuffer(slot = 0) cbData{
    float4x4 world;
};
```

Listing 7 provides an example SRSl constant buffer declaration. This will be replaced with the equivalent C++ code, found in listing 8.

Listing 8. C++ Constant Buffer

```
struct TYPE_cbData{
        glm::mat<4,4,float> cbDataCONCworld;
};

static TYPE_cbData* cbData = nullptr;
```

6) Textures and Samplers: Textures and Samplers are tackled in the same way as Constant Buffers. Since both samplers and textures also are external resources and have a binding point or slot associated with them. They are also passed as a vector of byte arrays in the main function. This is only in the fragment function since only fragment shaders have access to samplers and textures. The Texture and SamplerState types in SRSL are abstract representations for these resources, so in C++, they have to be made concrete. Currently, only the Texture2D type and SamplerState is supported. Their definition can be seen in listing 9

Listing 9. C++ Texture and Sampler Defintion

```
struct SamplerState{
        float borderColor[4];
    float mipLODBias;
    float minLOD;
    float maxLOD;
    int filter;
};
struct Texture2D{
    unsigned int width;
    unsigned int height;
    unsigned int channels;
    std::vector<unsigned char> data;
};
```

Although the SamplerState object has some members defined that specify how the texture should be sampled, these are not used in the current implementation. Currently, only linear repeat texture filtering is implemented, directly in the sampleTexture intrinsic function. Another limitation is the supported data type in the Texture2D implementation, which is currently only an unsigned byte array. So more detailed images such as HDR images (image of 32 bit floating point values) are not supported. The sampleTexture function is defined in listing 10

Listing 10. C++ sampleTexture function

```
glm::vec4 sampleTexture(
Texture2D* tex,
SamplerState* sampler,
glm::vec4 uv){
    unsigned int x = fmod(uv.x, 1.0f);
    x *= tex->width;
    unsigned int y = fmod(uv.y, 1.0f);
    y *= tex->width;
    unsigned int index = (y * tex->width + x);
    index *= tex->channels;
    float color[4];
    for (int i = 0; i < tex->channels; i++){
        color[i] = tex->data[index + i];
    }
    return glm::vec4(
        color[0] / 255.0f,
        color[1] / 255.0f,
        color[2] / 255.0f,
        color[3] / 255.0f
```

```
    );
};
```

This function first normalizes the UV coordinates to ensure they fall within the range [0, 1) and then converts them to texture space by multiplying with the texture's width. Using these adjusted coordinates, it calculates a 1D index into the texture data array, considering the number of channels in the texture.

B. The SRSL Graphics API Implementation

The SRSL Graphics API is a graphics API based on the pipeline model found in D3D11 and OpenGL. This model functions as a state machine, objects can be bound and unbound to configure this state machine. The way a frame is drawn depends on the current bound objects. For this project, only the necessary objects that are needed to draw a frame are implemented. This includes:

- VertexBuffer: Stores vertex data.
- IndexBuffer: Stores an array of indices
- VertexLayout: Specifies the attribute layout of a single vertex, attached to a vertex buffer.
- ConstantBuffer: Stores a block of data that can be read from in the shaders.
- Shader: Compiles and stores a shader program
- Texture2D: Stores a 2D texture that can be accessed in a fragment shader
- Color Attachment: A special 2D texture that can be drawn to.
- FrameBuffer: Stores a list of color attachments and configures the viewport.
- Context: Provides an interface for creating these resources.

Besides these objects, the context internally manages some other objects such as the pipeline object. It is not possible to directly interact with these objects. Some important but optional objects are missing such as the depth testing state.

1) The Shader Object: The shader object is an object that manages shader modules and assembles a shader program. It makes use of the SRSL compiler. As the SRSL compiler currently only supports vertex and fragment shaders, this Shader object only concerns itself with vertex and fragment shaders. To create a Shader, the vertex shader module and fragment shader module, both written in SRSL, are parsed by the SRSL compiler and then converted into C++. This C++ output is then compiled with g++, together with GLM, to create a small Dynamic Library. On Windows, this is a DLL. This DLL is then loaded into memory using the LoadLibrary [18] function found in the WIN32 API. If this step succeeded, the main function of both shaders are queried using the GetProcAddress [17], also part of the WIN32 API. After that, the Shader object can be bound to the pipeline and the shaders themselves can be executed.

2) The Pipeline Object: The pipeline object is an object managed internally by the Context. Objects from the list above can be bound to the pipeline. The pipeline represents the hardware and driver from the GPU. When a draw call is made, the bound objects to the pipeline are executed. The result of this draw operation is likely that something will appear on the screen. The pipeline internally has a rasterizer state and an input assembler state which are necessary to correctly execute the pipeline.

Before any bound objects are executed during a draw call, the current configuration of the pipeline is checked ensuring that all necessary objects are present. If this is the case, the vertex shader is executed for every vertex present in the currently bound vertex buffer. This is achieved with a simple loop. Per vertex, after it has been processed by the vertex shader, system values such as SRV_FRAGCOORD are added to the vertex. Each vertex is now considered to be an unordered map from strings to byte arrays. These strings are the semantic names which can be defined by the currently bound VertexLayout object or by the system. In which case, they all start with SRV_.

The processed vertices are then passed down to the rasterizer which will first generate triangles using the index buffer. The rasterizer will operate of the resulting list of triangles. Then each triangle is rasterized independently. This process starts by calculating the bounding box of the triangle followed by a check verifying that the current pixel is part of the triangle. This check makes use of the Barycentric coordinates which are also used by the interpolation algorithm. This algorithm loops over the attributes of each of the vertices of the current triangle, generating an interpolated vertex which is from that point called a fragment. The resulting fragment is then passed into the fragment shader which will write to any of the color system values. Currently, only writing to SRV_TARGET_0 will result in a pixel write in the color attachment at index 0 of the currently bound framebuffer.

Fragment shader execution does not always mean that a value will be written to the bound color attachment since it is possible that a fragment can be discarded. To simulate discarding a fragment in C++, a throw-catch construct is used. Whenever the SRSL compiled finds a discard statement, it will be replaced with the code seen in listing 11.

Listing 11. C++ Discard

```
throw int(0x87A9);
```

The fragment shader function call is wrapped in a try-catch block that only checks if this integer value is thrown. This approach works because it can circumvent the return statement found in functions in C++ although this comes at a performance cost.

If all triangles are drawn, the result is a color attachment that contains the drawn mesh. This attachment can be exported

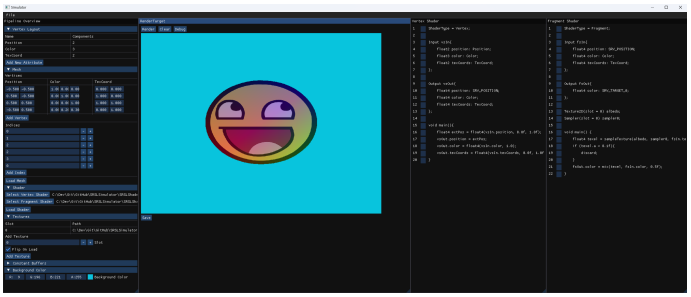

Fig. 3. The SRSL Simulator

as a PNG file.

C. The SRSL Simulator

The SRSL Simulator is a small program with a graphical user interface that uses the SRSL Graphics API. This program provides an easy way to create different environments in which SRSL shaders can be executed and debugged. These environments consists of a set of pipeline objects which can be saved as a JSON file which makes debugging easier. This application will also attach GDB to the pipeline and start a debug session. Figure 3 has an example environment loaded into the SRSL Simulator.

1) ImGui UI: The UI is rendered with OpenGL and the ImGui framework. This UI framework is an immediate mode UI that outputs a set of vertex buffers that can be easily intergrated into any application. This library has backends available for most platforms and Graphics APIs. It is used to quickly create basic UI widgets for debugging and visualization tools, optimized for graphics related projects. This framework was selected as creating a UI is not an important part of the project but thanks to its easy integration, provided a helpful tool in creating the simulator. ImGui is configured to use the OpenGL and WIN32 backend for this project.

The UI consists of 4 panels. From left to right, the first panel is the Pipeline Overview panel. This panel allow configuring the graphics pipeline. It has widgets to create vertex and index buffers, uploading textures and loading shader modules. The second panel is the Render Target Panel. This panel will always display the color attachment bound to index 0 of the default framebuffer. The pixel data of this attachment is uploaded to a OpenGL Texture2D object which ImGui uses to draw the attachment to the screen. The two last panels display the source code of respectively the vertex shader and the fragment shader. Each of these shader panels have a selection box attached to each line of the shader program. When such a line box is selected, the line mapping is used to determine the corresponding C++ line number. This is then used to instruct GDB to break at that line.

Fig. 4. AwesomeFace image

VI. Example Debug Session

A. Environment

To demonstrate the workings of the simulator, an example scene is drawn and debugged. This basic example will render a rectangle that consists of two triangles and a texture will be mapped onto this rectangle. The vertex shader from listing 1 is used together with the fragment shader found in listing 2. Their C++ equivalents can be found in appendix A.

The rectangle is compromised of 2 triangles totaling 6 vertices to render, 2 of these vertices overlap so the index buffer is used to create the triangles. Each vertex consists of three attributes with the following semantic names:

- Position: A 3-component vector indicating the position of the vertex.
- Normal: A 3-component vector representing the normal coordinate of the vertex.
- TexCoord: A 2-component vector representing the texture coordinate of the vertex.

The image mapped onto the rectangle can be seen in figure 4. This image measures 512 by 512 pixels, has 4 color channels each 8 bit wide. The fourth channel is the alpha channel and this image contains some transparent pixels. This makes it the ideal image to test texture mapping and the discard statement since the fragment shader will discard texels that have an alpha value lower than 0.1f. This check in the fragment shader will create the transparent effect.

B. Debugging

Suppose there is a bug when sampling from the texture in the fragment shader. The shader stores the sampled texel in a variable called ”texel” and the contents of this variable must be printed to the console. This texel variable is declared at line 17 in the SRSL shader which is mapped to line 39 in the C++ equivalent. GDB will be instructed to break 1 line after the variable declaration, 40 in the C++ code. First, attach GDB to the simulator using the command:

```
gdb -p <ProcessID>
```

The process ID of the simulator is displayed under the render target. Alongside the process ID, the temporary cpp file that stores the compiled shader is also displayed which can be used to set breakpoints in the program. As the fragment shader is executed for all pixels that make up a triangle, setting a break condition is a smart idea. In this example, GDB is instructed to break if the x-value of the Texel variable is not equal to 0. This is done using the following command:

```
break comp-fs.cpp:40 if texel.x != 0
```

The print or display commands can be used to inspect the contents of the Texel variable. If GDB is terminated, the simulator will resume drawing the frame as usual. After that, the render target panel should contain the rectangle with the texture mapped to it.

VII. Testing Framework

This project is done for the Second Research Project course. This project is done in parallel with a master thesis which also builds upon the SRSL compiler created in the First Research Project. In this thesis, research is done for developing a unit testing framework for the SRSL language and for automatic test generation using large language models or LLMs. This project further extends the toolchain available for the SRSL language. With this project, the difficulty here lies with retrieving data back from the GPU, which is an essential part for a testing framework. This project aims to utilize Shader Storage Buffer Objects (or SSBO's) to capture test data. These SSBO's are special buffers that can be both read and written to by both CPU and GPU. The main downside of this approach is that this requires explicit graphics API support as they are relatively new (only OpenGL 4.6). The SRSL language will be extended with syntax to write explicit testing statements for functions. The SRSL compiler will be responsible for converting this syntax into executable GLSL and HLSL code that fills in the test data SSBO. The other approach was to use this project's Simulator to execute the tests on the CPU.

The LLM part of this thesis aims to automatically analyze and generate appropriate test cases. The past year, some quality LLM's were released such as ChatGPT [20] and GitHub Copilot [8]. Another approach was to use a semi trained model and train it further on SRSL programs, but the difficulty with this approach is that there are currently insufficient SRSL programs written. Further research will be done how these models can be used in the testing aspect of the software engineering.

VIII. Further Improvements

Since this project is done individually, some parts are incomplete or can be improved. This project is currently designed to be a feasibility study so some parts received more attention than others. This section will go over a few important shortcomings.

A. Sampler Specification

The simulator does currently not support creating sampler states. These states define functions and properties how a texel should be sampled from a texture. They specify what should happen in the supplied UV coordinates are bigger than 1.0f. Should the texture be repeated or mirror repeated? Should a colored border be applied? Which filter should be used and so on. The generated sampleTexture function found the C++ fragment shader is hard coded to simply repeat the texture if the UV is bigger than 1.0f and the texture is filtered linearly. Despite an explicit SamperState object present in the exported C++ code. The SRSL graphics API does not have an interface ready for this object.

B. Shader Types

This limitation is constrained by the SRSL compiler itself. The SRSL compiler only supports vertex and fragment shader compilation and therefore, the SRSL simulator only supports simulating vertex and fragment shaders. Since SRSL is a domain specific language with the constraint domain being graphics shaders, there is currently no support planned for compute shaders. Other shader types such the tessellation shaders and geometry shader could be supported in the future by the SRSL compiler. These types can then also be supported by the simulator.

C. Textures

Textures have two important limitations. The first limitation is regarding the texture type, currently only 2D textures are supported. Cubemaps, 1D and 3D textures are not supported. They are already supported by the SRSL compiler but since this project is considered a proof of concept, 2D textures are sufficient for now. 3D textures and cubemaps are also more difficult to implement. They are, however, an important part of 3D computer graphics.

The second limitation comes from the supported texture types. Only image data consisting of 8 bit (unsigned chars) channels are supported. This means that more detailed images such as 16 bit or even 32 bit floating point image types are not supported. Uploading HDR images is thus not supported. This should not be too difficult to implemented but for now, 8 bit channels are sufficient.

References

[1] Baldurk. "RenderDoc's Early History". In: Accessed: 09/01/2024. June 6, 2017. URL: https://renderdoc.org/.

[2] The GDB Developers. *GDB: The Gnu Project Debugger*. Accessed: 15/12/2023. Dec. 22, 2023. URL: https://www.sourceware.org/gdb/.

[3] The Blender Foundation. *Blender*. Accessed: 06/01/2024. 2024. URL: https://www.blender.org/.

[4] The Eclipse Foundation. *Eclipse: IDE*. Accessed: 09/01/2024. 2023. URL: https://www.eclipse.org/.

[5] G-Truc. *GLM: OpenGL Mathematics*. Accessed: 11/01/2023. Aug. 16, 2017. URL: https://glm.g-truc.net/0.9.8/index.html.

[6] G-Truc. *GLM: OpenGL Mathematics*. Accessed: 11/01/2023. Jan. 5, 2024. URL: https://github.com/g-truc/glm.

[7] Epic Games. *The Unreal Engine*. Accessed: 06/01/2024. 2024. URL: https://www.unrealengine.com/en-US.

[8] GitHub. *GitHub: Copilot*. Accessed: 20/01/2023. URL: https://github.com/features/copilot.

[9] The Khronos Group. *OpenGL: Open Graphics Library*. Accessed: 06/10/2022. URL: https://www.opengl.org/.

[10] The Khronos Group. *Vulkan: Vulkan 1.4*. Accessed: 06/01/2024. 2024. URL: https://www.vulkan.org/.

[11] Shawn Hargreaves. "Introducing PIX on Windows". In: Accessed: 09/01/2024. Jan. 17, 2017. URL: https://devblogs.microsoft.com/pix/introducing-pix-on-windows-beta/.

[12] Intel. *Intel: What are GPUs?* Accessed: 06/01/2024. URL: https://www.intel.com/content/www/us/en/products/docs/processors/what-is-a-gpu.html.

[13] Javatpoint. *Difference between Emulation and Virtualization*. Accessed: 11/01/2023. URL: https://www.javatpoint.com/emulation-vs-virtualization#:~:text=An%20interpreter%20is%20required%20for,virtual%20machines%2C%20emulators%20are%20sluggish..

[14] Randi Rost John Kessenich Dave Baldwin. "The OpenGL® Shading Language". In: ed. by Google John Kessenich. Accessed: 11/01/2024. May 9, 2017. URL: https://registry.khronos.org/OpenGL/specs/gl/GLSLangSpec.4.50.pdf.

[15] Lennart C. L. Kats and Eelco Visser. "The Spoofax Language Workbench. Rules for Declarative Specification of Languages and IDEs". In: *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2010)*. Ed. by Martin Rinard. Reno, NV, USA, Oct. 17, 2010.

[16] Microsoft. *DirectX: Direct3D 11*. Accessed: 06/10/2022. Jan. 27, 2022. URL: https://learn.microsoft.com/en-us/windows/win32/direct3d11/atoc-dx-graphics-direct3d-11.

[17] Microsoft. *GetProcAddress*. Accessed: 18/01/2023. May 24, 2022. URL: https://learn.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-getprocaddress.

[18] Microsoft. *LoadLibraryA*. Accessed: 18/01/2023. Sept. 2, 2023. URL: https://learn.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-loadlibrarya.

[19] NesDev. *NES: Nintendo Entertainment System*. Accessed: 11/01/2023. Jan. 13, 2023. URL: https://www.nesdev.org/wiki/Nesdev_Wiki.

[20] OpenAI. *OpenAI: ChatGPT*. Accessed: 20/01/2023. URL: https://openai.com/blog/chatgpt.

[21] Terence Parr. *ANTLR4: ANother Tool for Language Recognition version 4*. Accessed: 17/01/2023. URL: https://www.antlr.org/.

[22] Arthur Préal. *SRSLSimulator Repository*. URL: https://github.com/SyriusEngine/SRSLSimulator.

[23] Arthur Préal. "The SRSL Shading Language". In: *The SRSL Shading Language*. Antwerp, Belgium, June 28, 2023.

[24] *Shadertoy*. Accessed: 09/01/2024. URL: https://www.shadertoy.com/.

[25] Sreevatsa Sreerangaraju. *What Are Simulators and Emulators? An Examination of Emulation vs. Simulation*. Accessed: 11/01/2023. Mar. 14, 2023. URL: https://www.perfecto.io/blog/emulation-vs-simulation#:~:text=Emulation%20vs.-,Simulation,environment%20of%20a%20real%20device..

.

APPENDIX

A. EXAMPLE C++ SHADER PROGRAM

The C++ code found in this appendix is generated by the SRSL compiler. Since the generated code generates long lines, some additional backspaces are added to make the Latex representation more readable. This causes the line numbers to be different from the generated line numbers which is important when trying to debug these shaders.

Listing 12. C++ Vertex Shader

```
1  #include <string>
2  #include <unordered_map>
3  #include <vector>
4  #include <glm/glm.hpp>
5  extern "C"
6  __declspec(dllexport)
7  std::unordered_map<std::string, glm::vec4>
8  __attribute__((stdcall)) main__Vertex(
9  std::unordered_map<std::string, char*>& vsIn,
10 std::vector<char*>& __CBUFFERS__) {
11 std::unordered_map<std::string, glm::vec4> vsOut;
12     {
13         glm::vec<4, float> extPos = glm::vec<4, float>(
14             *reinterpret_cast<glm::vec<2, float>*>(vsIn["Position"]), 0.0f, 1.0f
15         );
16         vsOut["SRV_POSITION"] = extPos;
17         vsOut["Color"] = glm::vec<4, float>(
18             *reinterpret_cast<glm::vec<3, float>*>(vsIn["Color"]), 1.0
19         );
20         vsOut["TexCoord"] = glm::vec<4, float>(
21             *reinterpret_cast<glm::vec<2, float>*>(vsIn["TexCoord"]), 0.0f, 1.0f
22         );
23     }
24     return vsOut;
25 }
```

Listing 13. C++ fragment Shader

```cpp
#include <string>
#include <unordered_map>
#include <vector>
#include <glm/glm.hpp>
struct SamplerState{
    float borderColor[4];
    float mipLODBias;
    float minLOD;
    float maxLOD;
    int filter;
};
struct Texture2D{
    unsigned int width;
    unsigned int height;
    unsigned int channels;
    std::vector<unsigned char> data;
};
glm::vec4 sampleTexture(Texture2D* tex, SamplerState* sampler, glm::vec4 uv){
    unsigned int x = fmod(uv.x, 1.0f) * tex->width;
    unsigned int y = fmod(uv.y, 1.0f) * tex->width;
    unsigned int index = (y * tex->width + x) * tex->channels;
    float color[4];
    for (int i = 0; i < tex->channels; i++){
        color[i] = tex->data[index + i];
    }
    return glm::vec4(
        color[0] / 255.0f,
        color[1] / 255.0f,
        color[2] / 255.0f,
        color[3] / 255.0f
    );
}
static Texture2D* albedo = nullptr;
static SamplerState* sampler0 = nullptr;
extern "C"
__declspec(dllexport)
std::unordered_map<std::string, glm::vec4>
__attribute__((stdcall)) main__Fragment(
std::unordered_map<std::string, glm::vec4>& fsIn,
std::vector<char*>& __CBUFFERS__,
std::vector<char*>& __TEXTURES__,
std::vector<char*>& __SAMPLERS__) {
    std::unordered_map<std::string, glm::vec4> fsOut;
    sampler0 = reinterpret_cast<SamplerState*>(__SAMPLERS__[0]);
    albedo = reinterpret_cast<Texture2D*>(__TEXTURES__[0]);
    {
        glm::vec<4, float> texel = sampleTexture(albedo, sampler0, fsIn["TexCoord"]);
        if (texel.a < 0.1f)             {
            throw int(0x87A9);
        }
        fsOut["SRV_TARGET_0"] = mix(texel, fsIn["Color"], 0.5f);
    }
    return fsOut;
}
```