Simulating the GPU-based shaders in the graphics pipeline on a CPU-based language to allow code inspection at runtime

Masterthesis

at the University of applied science Ravensburg-Weingarten

by

Matthias Mettenleiter

September 2019

Student ID	29015
Supervisor	Daniel Scherzer
Secondary supervisor	Sebastian Mauser

Author's declaration

Hereby I solemnly declare:

1. that this Masterthesis, titled *Simulating the GPU-based shaders in the graphics pipeline on a CPU-based language to allow code inspection at runtime* is entirely the product of my own scholarly work, unless otherwise indicated in the text or references, or acknowledged below;
2. I have indicated the thoughts adopted directly or indirectly from other sources at the appropriate places within the document;
3. this Masterthesis has not been submitted either in whole or part, for a degree at this or any other university or institution;
4. I have not published this Masterthesis in the past;
5. the printed version is equivalent to the submitted electronic one.

I am aware that a dishonest declaration will entail legal consequences.

Weingarten, September 2019

Matthias Mettenleiter

Abstract

Debugging is an important part of the programming process. Most programming languages are supported by different debugging tools. For writing shaders within the graphics pipeline, the existing tools and support for debugging are scarce and most of the tools provided are limited to use with specific hardware and specific drivers for said hardware.

This thesis provides a solution to enable the use of different debugging tools for shaders in the graphics pipeline without the dependency on specific hardware or drivers while still providing the advantage of being executable on the GPU. To achieve this, the shader is written in a language supported by debugging tools while being executed in a simulated version of the graphics pipeline on the CPU. This shader is then translated to a regular shader language to run on the GPU.

Contents

1 Introduction

To introduce the content of this work the concepts of debugging and the behavior of shaders on the graphics pipeline are explained. After that the problem of debugging shaders in the graphics pipeline is shown and the objectives of this work are presented.

In this work a basic understanding of computer graphics and the related terminology is expected.

1.1 Explanation of debugging

"Debugging is the process of locating and removing faults in computer programs" according to [Collins 2014] . The steps that are part of the debugging process are reproducing the problem, identifying the source of the problem and fixing the problem. All of these steps can be done manually but there are methods to improve and accelerate this process.

It is possible to write a program with formal verification, where the program is based on a mathematical proof ensuring its correctness. For these programs debugging is not necessary but formal verification has its limits with programs getting too complex. [Bolton, Bass, and Siminiceanu 2013]

For debugging, to find problems there are the options of writing automated tests, inserting debug outputs on the console into the source code or writing states into log files. This enables the programmer to find anomalies before, while and after running the program.

Once a problem is reproducible, the source of this problem needs to be uncovered. The manual way of doing this, is to increase the amount of debug outputs around the problematic part of the code until the bug is found.

"Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?" according to [Kernighan 1982] . As this states, debugging is a quite exhausting and time consuming task. For that reason, tools exist to aid the programmer in narrowing down the source of the bug for most programming languages. Those tools do this with following methods:

- The user is enabled to set breakpoints at which the program pauses and he can inspect the values of the variables directly within the code. He can find the error by either continuing the program until the next breakpoint or he can go through the code step by step. It is also possible to be able to add conditions to the breakpoints describing a state that has to be fulfilled for the debugger to pause. [Undo 2019]
- The code throws an exception when unwanted behavior occurs and stops at this exception. By saving a stack of the calls which occurred before the exception was thrown the programmer can retrace where the error may be found. [Jetbrains 2019]
- Reverse debugging records all program activities and thereby it is possible to step backwards in addition to stepping forward from a set breakpoint and see the changes in the variables and the calls in the code leading to the problem. [Undo 2019]

After the programmer finds the problem manually or with the aid of debug tools the code can be corrected.

1.2 Explanation of shaders in the graphics pipeline

A shader is a program running on the GPU thereby mostly running as part of the graphics pipeline. [Richard S. Wright 2007] The exception for this behavior is the compute shader which is independent from the graphics pipeline. [Dave Shreiner 2013]

The main use of the graphics pipeline is to generate a graphical output in form of a raster image out of geometry data. The GPU is a hardware specialized for this process leading to improved performance compared to running the same calculations on the CPU. This is shown in more detail in the paragraph Performance differences between graphics calculation on a GPU and a CPU.

Common types of shaders on the graphics pipeline are the vertex shader, the tesselation shader, the geometry shader and the fragment shader. Their basic functionality is described in the following paragraph showing the sequence of events within the graphics pipeline.

Structure of the graphics pipeline The graphics pipeline, also known as rendering pipeline, differs from framework to framework but the basic structure of the pipeline within each of them is identical. [Peter Shirley 2009]

In the first stage the vertices are specified and set up in an ordered list. Attributes can be linked to the vertices adding data to them.

In the second stage the vertex shader is executed for each defined vertex with its associated data. For each vertex the vertex shader generates an output vertex with data.

After the vertex stage multiple optional stages can occur. Examples for these are the tesselation stage or the geometry stage where additional shaders are executed.

In the following stage there are multiple optimization steps followed by the primitive assembly and the rasterization. The perspective divide and the viewport transformation are calculated. Through clipping, which is optional, primitives that overlap the edge of the viewing volume are split into multiple primitives and primitives outside the viewing volume are removed. Primitives are assembled according to the primitive type, so a list of primitives is constructed out of the list of vertices where each primitive is constructed out of multiple vertices. Face culling can occur as additional optimization that removes primitives which face certain directions in the window space. The primitives are then rasterized into fragments within the raster of the render result. The data values linked to the vertices are set for each corresponding fragment by hardware based calculations.

The fragment stage executes the fragment shader for each single fragment. The output data is calculated and attached to the corresponding fragment.

After the fragment stage the output of the render process is calculated. For each sample there are multiple optional filters that can be activated to change which fragments are used within this stage. Examples for these filters are the scissor test where a fragment is ignored when its pixel lies outside of the rectangle of the screen or the depth test where the fragment's order is changed when its depth fulfils specific, user defined conditions compared to another fragment's depth within the same sample. After all active filters have been executed the color blending is executed where the color between multiple fragments within the same sample is calculated with a specific blending operation.

The final data is written to the framebuffer.

The two most commom implementations of the graphics pipeline are found within the graphics libraries OpenGL and Direct3D. Their definitions of the graphics pipeline within them are shown in Figure 1.1 and Figure 1.2

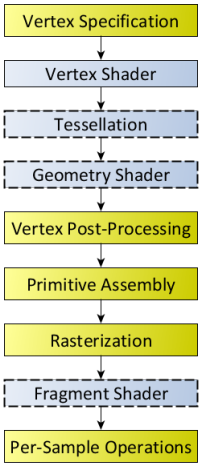

Figure 1.1: Graphics pipeline by OpenGL

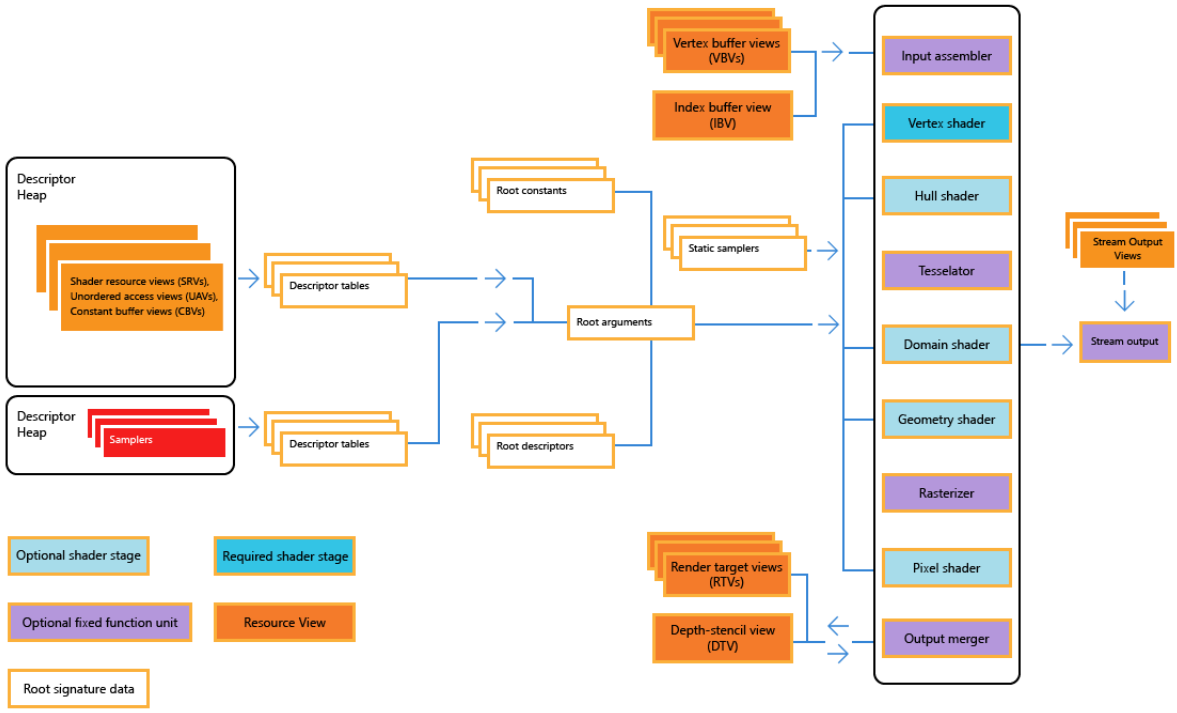

Figure 1.2: Graphics pipeline by Direct3D

Performance differences between graphics calculation on a GPU and a CPU The main advantage in using shaders in the graphics pipeline on the GPU lies in the increased performance compared to the CPU. This correlation can be observed if the project, which was implemented for this work [*Projektarbeit*] , is analyzed.

The chosen test case consists of a simple triangle covering half the space of a window with a resolution of 640 by 480 as shown in Figure 1.3. The test is run on a Nvidia GTX970 as GPU and a Intel i7-6700K as CPU. It has to be noted though that there are ways to further optimize the calculation on the CPU. For this test the program is not multi

threaded and is running in a JIT (Just in Time) compiler. But it should be enough to serve as a rough example for the difference in performance.

Rendering the described image on the GPU takes about 0.0015 ms while calculating the same output in the CPU with the implemented test takes about 800 ms. The rasterizing step with calculating the data for all fragments from the data of the vertices within the corresponding primitives alone takes over 300 ms in this test.

As shown with this test it is desirable to use the graphics pipeline on the GPU.

Figure 1.3: Example of a triangle rendered on a window with 640 by 480 pixels

1.3 Problem with debugging shaders in the graphics pipeline

As established in section 1.2 there are advantages in the use of shaders on the GPU. There are different tools to support in implementing shaders. [Jones 2019] [Scherzer 2019a] [Scherzer 2019b] For debugging shaders on the graphics pipeline though there are no general solutions that aid the user in the process.

While most CPUs are very broad in their functionality and support debugging by themselves a GPU is more specialized in the way it functions and usually does not have the option to pause the code to enable inspections at runtime and does not even have access to a console or logger to write states into. [Fox 2017]

As explained in more detail in section 2.1 it is possible to enable debugging on the GPU with specialized drivers for specific hardware. [Nvidia 2019a] [Microsoft 2016]

There are also approaches which enable debugging of compute shader code by translating it from another language. See more in section 2.2.

1.4 Objective of creating a general solution for debugging shaders in the graphics pipeline

The objective of this work is to create a general solution to enable debugging of shaders within the graphics pipeline. The goals this solution should fulfill are the following:

- Different methods to assist the programmer in debugging as shown in section 1.1 are usable.
- The solution is not dependent on the use of specific graphics cards or drivers.
- It is possible to switch between a mode where debugging is enabled and a mode where the shaders run as usual, so the program can run with the full performance and without interference of the debugger.
- The resulting output per render iteration of the debugger is close to the output of the program with the undebugged shader. It is close enough that the programmer can see what the rendering result without the debugger would look like. Errors like those resulting from the use of float variables with their inaccuracies are tolerable because there are tolerances within human perception where minimal changes in position or color within a rendered result do not matter. [Franz 2006]
- Performance is not a major requirement while running the debugger. It is possible to see the output and the values of the shader within each frame and iterate through the frames. It is not necessary to view the result in the speed of the final application while debugging. In existing debugging tools the programmer also has to wait until the program reaches the part where he wants to start debugging. This means the debugger does not have to be as fast as a user interface with the goal of a fluid user experience. It is sufficient that the user does not have to wait too long. According to [Nielsen 1993] there are the following three important limits:
  - "0.1 second is about the limit for having the user feel that the system is reacting instantaneously, meaning that no special feedback is necessary except to display the result."

- "1.0 second is about the limit for the user's flow of thought to stay uninterrupted, even though the user will notice the delay. Normally, no special feedback is necessary during delays of more than 0.1 but less than 1.0 second, but the user does lose the feeling of operating directly on the data."
- "10 seconds is about the limit for keeping the user's attention focused on the dialogue. For longer delays, users will want to perform other tasks while waiting for the computer to finish, so they should be given feedback indicating when the computer expects to be done. Feedback during the delay is especially important if the response time is likely to be highly variable, since users will then not know what to expect."

Therefore the targeted performance should enable the user to reach a debugging point within a maximum of 10 seconds.

2 Related Work

2.1 Existing methods for debugging shaders in the graphics pipeline

For debugging a shader program within the graphics pipeline there are the options to use workarounds to get the values of the variables within the code or to use special drivers provided by the producers of the hardware to get the option to debug on this hardware.

Manual debugging The manual way of debugging a shader is by creating outputs of the values within the shader program to see anomalies in their values. This can't be done by writing these values on the console or in a log file like it would be done in a CPU based application because as mentioned in section 1.3 there is no access to the console or a logger within a shader program. The workaround used here is to return the values projected on the rgba-color values on the resulting image of the program. In this way the programmer can get an approximation of the values on the direct output. The image can also be saved and inspected closer to get the exact values within the pixels of the resulting image. [Ciardi 2015]

Debugging with special drivers on certain hardware It is possible to install special drivers for certain graphics cards provided by their producers to enable debugging of shaders running on this hardware within dedicated environments. The two biggest suppliers of graphics hardware Nvida and AMD provide these debugging environments in the form of *Nvidia Nsight* and *GPU PerfStudio*. These tools can be included into different IDEs or downloaded as standalone applications. There are debugging tools for GPU debugging provided by other sources than the producers of the hardware themselves but for gaining access to the values in the hardware pipeline the drivers of this hardware have to implement specific debugging interfaces. [Microsoft 2016] Applying these tools enables the use of breakpoints and the inspection of variables within the shader code at runtime [*GLSL-Debugger*] or dumping the buffer into a file. [Microsoft 2019b] The disadvantage of this

method is that not all graphics cards are supported with such drivers and tools by their producer.

2.2 Approaches for debugging compute shaders on the CPU

As [Jukka 2012] states "Relatively little has been published about debugging of GPU programs in practice. Most of the best practice guides and lessons learned papers discuss GPU programming rather than debugging. Although exceptions exist". Among these exceptions are approaches of writing compute shaders in other programming languages. The version of the shader in the other language can be run on the CPU where it can be debugged. This version of the shader will then be translated to the actual shader language so it can run on the GPU with the full performance advantage. This enables the use of all tools the chosen language supports to aid debugging without depending on specific hardware or drivers. Examples implementing this for simulating CUDA shaders in C# are [*ILGPU*] and [*Campy*] .

The advantage compute shaders have in comparison with other kinds of shader programs is that they run independently of the graphics pipeline. For a compute shader a buffer of input data is defined. The compute shader is then executed, writing the results to an output buffer. The output can be accessed by copying this buffer. This process can be emulated within a program running on the CPU by applying the calculations of this shader to the data as a function.

In comparison, for a shader in the graphics pipeline, the inputs and outputs are handled by the pipeline. The user defines the input for the pipeline itself. The shaders define how the data is processed within their corresponding stage and which data flows in and out of this stage. In between the shader stages there are calculations applied to the data flowing through. These calculations are defined by the pipeline. Here the individual shaders are dependent on the pipeline and the shaders of the other shader stages. Because of this emulating shaders in the graphics pipeline is more complex than emulating compute shaders.

2.3 Translating shaders from other languages

To run code on the GPU usually the programmer has to use one of multiple low-level languages specialized in this task. To enable programmers to run code on the GPU while being able to stay at the language they are accustomed to there are multiple solutions to translate different languages into these low-level languages.

There are three different types of programming languages. The ones that are directly compiled ahead of time (AOT), the ones that are compiled to an intermediate language which then runs in a JIT compiler and the ones that are running in an interpreter. For these three types there are different approaches to translate them to a shader language. [Turton 2017]

For all of these kinds of languages, it is possible to write a compiler translating them directly to the shader language. This is what [*GLSLplusplus*] does with the AOT compiled language C++ translating it to the shader language GLSL. In this project created to change the way of writing shader code the possibility of using this to debug the shaders written in C++ is mentioned. [*SharpShader*] is another example which uses the Language C# that is usually compiled to the intermediate language MSIL. In this example C# code is directly compiled to a Shader languages HLSL or GLSL. In both cases the programmer can use his accustomed language with the limitation of having to write the code for the GPU within special syntax rules to be able to translate it into a shader for the GPU. An advantage that results of the direct translation is that variable names given in the written code can be directly retained in the shader code.

Another option with the use of a language that compiles to an intermediate language is to take the code in the intermediate language and further translate it into a shader language. Examples implementing this are [*ILGPU*] and [*Campy*] where the shaders are written in C#. This code is then compiled to the intermediate language MSIL. The MSIL code is further translated into the compute language CUDA which then runs on the GPU. Another example is [*SpirvNet*] which translates MSIL to Spirv. Spirv is an intermediate language for graphical shader code that can be interpreted as a shader on the GPU or translated further into other shader languages with tools such as [*SPIRV-cross*] which translates to GLSL. The translation in all of these cases is based on a uniform intermediate language which could be the result of multiple higher level languages. So by writing the translator for this intermediate language the option exists to write the shaders in different languages compilable to this intermediate language. It can occur that some variable names are lost by compiling to the intermediate language which would impede retaining these variable names in the shader code. [Miecznikowski and Hendren 2002]

To translate code to a shader code it is also possible to chain multiple existing tools together to translate the language, in which the shaders should be written in, to the desired shader language. An example for this process can be seen in the way the Interpreted language R is translated through multiple steps to run on the GPU in the example "Just-In-Time GPU Compilation for Interpreted Languages with Partial Evaluation" [Fumero et al. 2017]

3 Contribution

The proposed solution of this work is inspired by the approaches for compute shaders described in section 2.2. The code for the shaders is written in a source language which supports the different methods for debugging listed in section 1.1 while running on the CPU.

To run the shaders as usual on the GPU the shaders written in the source language are translated to the shader language and loaded on the hardware.

To enable the functionality while debugging, the steps in between the shaders on the graphics pipeline, usually already provided by the graphics hardware, will be simulated in the source language.

3.1 Steps for translating the shader code

In most cases the access of attributes within a shader is done by getting its location from its name. [Lighthouse3d 2013] Because of this it is advantageous to retain the variable names from the source language. As explained in section 2.3 by directly translating the language without compiling it to intermediate stages all kinds of languages can be chosen as source languages and the variable names can easily be retained.

Shaders within the graphics pipeline have a special structure, in order to allow the input and output within their respective stage to flow correctly through the shader. Because of this the code written in the source language meant to be translated to a shader should mimic the structure the shader language provides. This means when the shader language provides certain structures for input or output or a specific point of entry into the code an equivalent for these has to be given in the code written in the source language.

The translation is done by converting the source language into a syntax tree from which the syntax will be extracted in the shader language. Occurences of variable types also have to be translated to their correct equivalent.

It has to be considered that specific syntax, like header lines or special accessors, only existing in the shader language has to be emulated in some way. Examples for this are shown in chapter 4 for the translation of C# to GLSL.

3.2 Steps for simulating the graphics pipeline

To simulate the graphics pipeline to be able to run and thereby debug the shaders written in the source language on the CPU the shaders have to be written in a structure that they can be inserted and exchanged within the simulated pipeline. The simulated pipeline has to implement the steps usually running automated on the GPU in between the different shader stages.

It is determined which kinds of shaders should be able to be simulated and therefore which parts of the pipeline as described in section 1.2 have to be implemented.

Instance of a shader within the source language A shader in the source language should be contained within a fixed structure like a class. Thereby the execution of the shader within the simulated pipeline is easier to manage compared to having to compose the shader out of multiple parts. As already mentioned in section 3.1 it is optimal to have the code of the shader mimic the structure of the shader in the original language. Not only does this simplify the translation process, but it is also benefits the implementation of interfaces for the shaders similar to the actual shaders in the pipeline. To achieve this, for each access point the regular shader has in the pipeline there is an equivalent in the simulation. The following things have to be given:

- The possibility to set variables as attributes or uniforms
- The function starting the execution of the shader functionality
- The option to access the outputs of the shader after it has been executed

For the shader to be able to function on the CPU as on the GPU the types and functionalities existing in the shader language are implemented in the source language. For example there are the multiple forms of vector and matrix types based on the same base types. Also, the methods and operators desired to be used of the shader language are implemented. The difference between value based and reference based behavior in the languages has to be considered in this step. [Magyar 2017]

Implementation of the graphics pipeline The way the shaders are accessed and managed within the pipeline, as well as the calculations in the steps between the shaders, as described in section 1.2 are implemented. The different optional steps for optimizing in between the shader stages which can be enabled or disabled within the regular graphics pipeline like clipping, culling or the optional filters in the end are implemented if their use is desired. In the following the basic steps to be implemented are described. An overview of the described points is displayed in Figure 3.1.

The ordered list of the vertices as well as the attributes affiliated to them are saved in the pipeline.

To be able to set and get the attribute and uniform values by its variable names as it is done for actual shaders, a way to access the variables within the shader from a string containing the variable name is implemented. [Lighthouse3d 2013]

To calculate the vertex step, the vertex shader is executed for each vertex with the attribute values for the vertex being passed to the shader instance. The resulting vertex data is then gained as outputs of the shader instances.

The same process is done for each of the optional shader stages after the vertex step.

The perspective divide and the viewport transformation are calculated and applied to all vertices.

To implement the primitive assembly, the vertices are grouped together according to the primitive type which is defined within the pipeline.

A raster in the size of the resulting image is generated. Within the raster for each position, a list of fragments is created. For each primitive and all positions in the raster overlapping it a fragment will be generated and saved in the corresponding list of fragments within the raster. The attribute values of each fragment are calculated from the values of the vertices of their corresponding primitive and added to this fragment.

After all fragments are generated for each of them additional data like the depth and the stencil information are calculated and the fragment shader is executed with the attribute values being passed to the shader instance. The resulting fragment data is gained as outputs of the shader instances.

Desired evaluations like the depth test or a stencil test are applied and if enabled a chosen blending function is applied to the fragments.

The resulting color data of the fragments is finally added to a raster that can then be displayed as the render result.

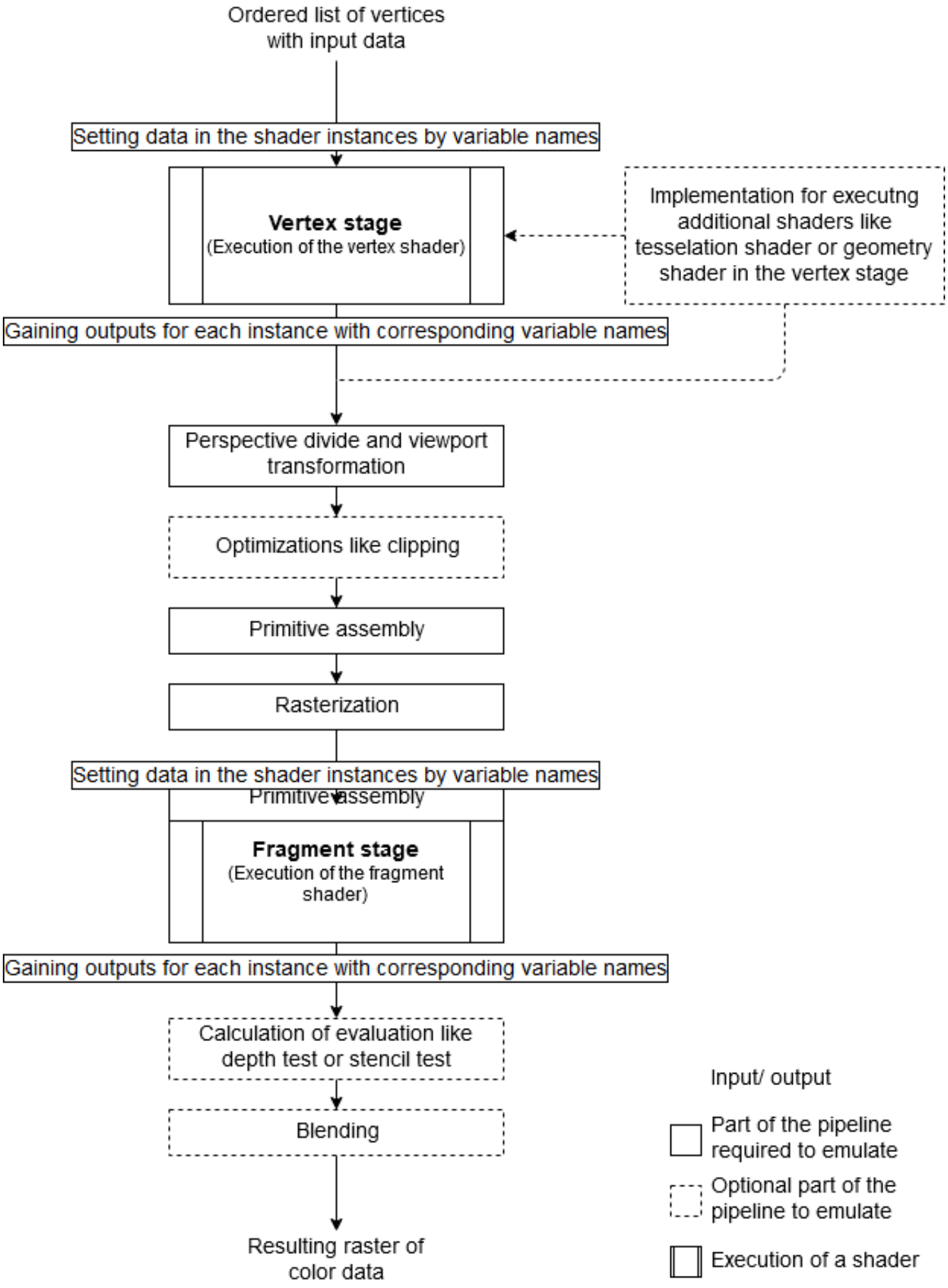

Figure 3.1: Overview of the required steps to implement in the simulation

4 Implementation

In the project [*Projektarbeit*] an example to simulate shaders in the graphics pipeline is implemented. The simulation and the shaders are written in C#. The shaders are translated to GLSL which is then run on the GPU within an OpenGL program.

C# is supported by multiple debugging frameworks including the VisualStudio Debugger which has a broad spectrum of tools. [Microsoft 2019a] GLSL is the main shader language for OpenGL which is one of the most used graphics frameworks. [Nvidia 2019b] Another reason those languages and frameworks are chosen is that I personally am very accustomed to them.

By writing the code in C# all debugging features coming with VisualStudio can be used for the simulated shaders.

Features implemented The project is a proof of concept where only parts of the functionality of shaders in the graphics pipeline are implemented. The implemented features suffice to run a basic example and evaluate the process as described in chapter 5. In the following the different features needed to fulfill this requirement are listed. The necessary steps to implement them are shown in the subsequent paragraphs.

- Only the two types of shaders necessary to calculate a graphical output are supported. These are the vertex shader and the fragment shader which process the vertex and the fragment data within the graphics pipeline.
- For the primitives only the type triangle is supported. Triangle is the most common primitive type and is sufficient to render examples.
- The result is an image with color outputs for each fragment. This is the usual output for a process in the graphics pipeline and used for the example.
- The viewport is always spanning the full output window to simplify the required calculations.
- A depth test only rendering the fragment with the smallest depth is implemented to enable the rendering of basic 3D scenes.

- Only a limited set of variable types and methods to use in a shader are supported. Which ones are supported, is explained in more detail in the paragraph Implementation of the variable types.

In the following paragraphs it is further explained how the translation and the simulation are implemented and which part of them is used to add these features.

Structure of a shader in C# A shader is written as a class, which inherits necessary features from a base class "Shader".

The base class "Shader" has following features:

- An abstract "Main" function which acts as the entry point for each shader and has to be implemented in the different shaders.
- An implementation of the different mathematical functions necessary for the basic examples.
- A method which allows to set the value of a property within the shader class by passing its name as a string and the value as a generic type. The properties are found, and the value is set by accessing it via Reflection. To differentiate between a property emulating a "in" or a "uniform" variable this method also gets an attribute type to be able to check if the property has a specific custom attribute attached to it.
- A method to return the names and values of all properties having a custom attribute of the type "OutAttribute" attached to them. These properties are found by using Reflection.

To implement the different behaviors of a vertex and a fragment shader, the resulting shaders are not directly inheriting the Shader class but one of two additional abstract classes implementing the base class "Shader":

- The "VertexShader" class having an additional property named "Position" to emulate the built-in GLSL variable "gl_Position" GLSL has [Dave Shreiner 2013] . This ensures the vertex shader always returns a position value for the output vertex.
- The "fragmentShader" class having an additional property named "Color" with an "OutAttribute" attached to it so there is always a Color variable to draw the fragments.

Translation of the shader class The shader class is contained within a single file. To translate it, the path to this file is given to the translator which will extract the syntax tree from the shader class. The different nodes are translated according to their type and defined translation rules.

The syntax for different nodes is implemented manually for these node types.

For each identifier within the nodes of the syntax tree, it is checked if a "TranslationAttribute" defining a term to replace it with exists. If this is not the case, the identifier is directly transferred to the shader code.

There are special patterns in the GLSL code that have no direct equivalent within C#. The following solutions are implemented within the project:

- Each GLSL shader has a line at the top defining its version. This line is simply added at the beginning of each translated shader and not emulated within the C# version of the shader. The version cosen in the example is the version 4.3 because the shaders in the example are known to work with this version.
- Each variable within a shader can have an accessor defining whether it is a writable attribute, a readable attribute or a uniform variable. This behavior is mimicked by having properties in the C# shader class that can have one of the three custom attribute classes "InAttribure", "OutAttribute" or "UniformAttribute" attached to them. This attribute is replaced by the corresponding "in", "out" or "uniform" tag when being translated to GLSL.

Implementation of the variable types To run the shader within the pipeline, the code within it has to be functional. For this reason the different variable types with their functionality have to be implemented in C#. For this project the implementation is limited on the variable types "vec2", "vec3", "vec4" and "mat4". For each of these a equivalent is implemented. By implementing them as structs instead of classes, the value based behavior they have in the GLSL code is reconstructed. These structs are implemented based on float values like their GLSL counterparts. The different forms of accessing their values are implemented in the form of properties and the necessary operators for the mathematical calculations with them are added.

Simulating the pipeline The stages are implemented with the functionalities described in section 3.2. Thereby the features listed in paragraph Features implemented are implemented.

Within the pipeline the attributes and uniforms for the shaders are set and the output values gained by using the methods implemented within the base "Shader" class. The exception is the "Position" property of the vertex shader which is accessed directly and used for the interpolation within the pipeline.

The final result is written into a "Bitmap" variable.

Surrounding structure The application is running within the OpenGL game loop.

The geometry to be rendered is prepared and manipulated in the update loop.

Within the render loop the data for the render process is generated and then either drawn by having the translated shaders run on the GPU or given to the simulated pipeline which is then executed and returns a Bitmap variable that is then rendered as a texture on a screen filling quad.

Ommited features In this paragraph a list of features not implemented in the project is presented. For each of these features the reason why they are not implemented and a basic concept for a possible way to implement it are given:

- In addition to the two shader types supported in the project there are other types of shaders like the tesselation shader and the geometry shader which can implement further manipulation, removal and addition of vertices. These were left out in the project because implementing their support is not necessary for the example application and would just inflate the code of the simulation. For implementing them corresponding sub classes of the class "Shader" with the respective inputs, outputs and usually hardware implemented functionality is implemented and their execution is added to the simulated pipeline.
- Other primitive types than the triangle type can be supported by expanding the primitive assembly. In the project only meshes consisting triangles were rendered, therefore this was not necessary.
- To enable the result being something different than an image consisting of a raster of color data or having additional data outputs a way to return all output values of the fragment shader in a raster to be used after the execution of the simulated pipeline would have to be added. This is not implemented because the presented example only produces an image as output.
- To support rendering in a specified viewport an option to define a viewport and the viewport transformation within the pipeline have to be added to the simulation.

- More evaluation types for the fragments than the depth test are not implemented. If features like the use of stencils with a stencil buffer or blending are desired these can be implemented after the execution of the fragment shader.
- If more variable types and methods in the shader are desired, their equivalents have to be implemented for the simulation. Not all methods and all variable types are implemented for the example to reduce the time needed for the implementation.

5 Evaluation

Used example geometry and shaders For a basic example a scene consisting out of a tetrahedron which is rendered in three instances is chosen. These instances are assigned different color and different transformations as shown in Figure 5.1.

The example for a vertex shader is applying the instance transformation to the vertex position and to its normal, applying a camera transformation to the resulting position and passing through the instance color.

The example for the fragment shader receives the position, the normal and the instance color and applies a light calculation based on a given directional light and the camera position.

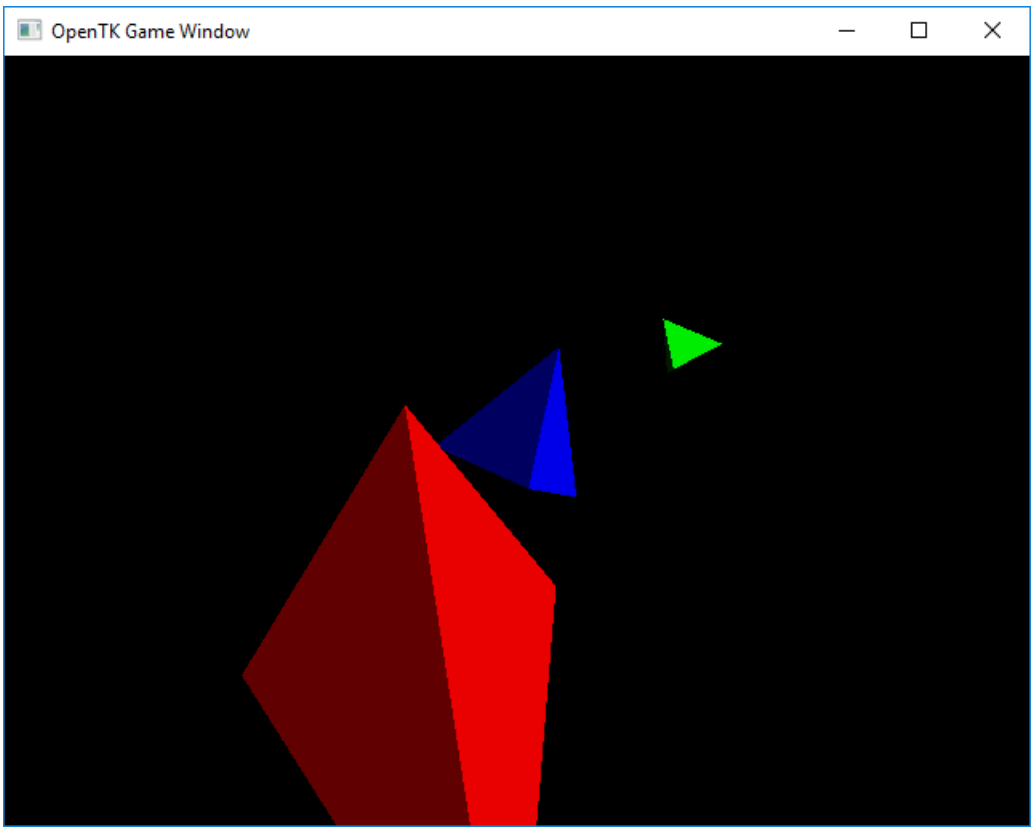

Figure 5.1: Project scene

Translation functionality For the chosen example shaders, all parts of the C# shader code are translated to a functioning shader in GLSL. The code examples in Listing 5.1,Listing 5.2, Listing 5.4 and Listing 5.5 show shaders written in C# and their respecitve translation in GLSL. As seen in the vertex shader all parts of the C# code have their equivalent in the shader code. For the fragment shader the result has a output variable for the color in addition to this. This additional output exists because this output is implemented in the base class for a fragment shader shown in Listing 5.3. In addition to the translation of the existing parts in the C# code the headline for the version number is added for both shaders in the translation process. In the example this version number is always set as "version 430 core".

```
class PassVertex : VertexShader
{
    [In]
    public Matrix4x4 InstanceTransformation { private get; set; }
    [In]
    public Vector3 Pos { private get; set; }
    [In]
    public Vector4 Color { private get; set; }
    [Out]
    public Vector4 Col { get; private set; }

    public override void Main()
    {
        Position = (InstanceTransformation * new Vector4(Pos, 1));
        Col = Color;
    }
}
```

Listing 5.1: Vertex shader written in C#

```
#version 430 core
in mat4 InstanceTransformation;
in vec3 Pos;
in vec4 Color;
out vec4 Col;

void main()
{
        gl_Position = (InstanceTransformation * vec4(Pos, 1));
        Col = Color;
}
```

Listing 5.2: Vertex shader translated in GLSL

```
public abstract class FragmentShader : Shader
{
    [Out]
    public Vector4 Color { get; protected set; }
}
```

Listing 5.3: Base class of a fragment shader in C#

```
class PassFragment : FragmentShader
{
    [In]
    public Vector4 Col { private get; set; }

    public override void Main()
    {
        Color = Col;
    }
}
```

Listing 5.4: Fragment shader written in C#

```
#version 430 core
out vec4 Color;
in vec4 Col;

void main()
{
        Color = Col;
}
```

Listing 5.5: Fragment shader translated in GLSL

Result of the simulation Running the translated GLSL shaders on the GPU with the example described above result in outputs that can't be distinguished as shown in Figure 5.2 and Figure 5.3.

The only noticeable difference between rendering the translated shader in comparison to rendering the simulated version is the performance. As shown in Figure 5.4 rendering a single frame in the simulation on the CPU takes about 1500 ms while rendering a single frame on the GPU with the GLSL shaders takes only 0.0015 ms for this example. These values are gained while running the example on a Nvidia GTX970 as GPU and a Intel i7-6700K as CPU. But as described in section 1.4 this is tolerable for the purpose of debugging.

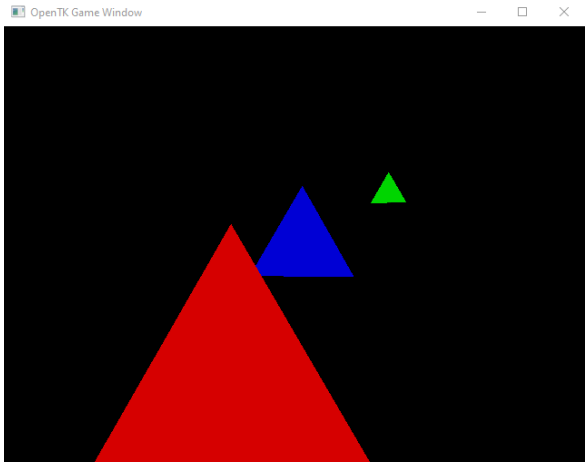

Figure 5.2: Result of rendering with the translated shaders

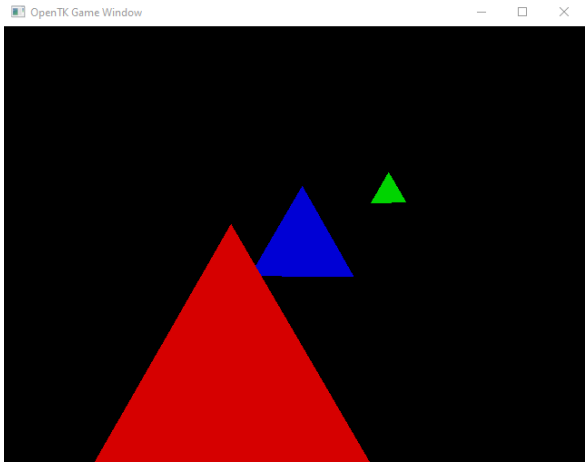

Figure 5.3: Result of rendering with the C# shaders on simulated pipeline

Debugging of the C# shaders The shaders written in C# and running in the simulated graphics pipeline can be debugged as any other C# code in VisualStudio.

It is possible to set breakpoints and add conditions to them. Setting such a condition could be used to debug a specific fragment or vertex. From these breakpoints it is possible to step through the code, inspect the variables and even navigating backwards through the call stack.

In the example no exceptions are thrown but if any of the underlying methods would throw a exception VisualStudio would stop the execution and show the problem in the debugger.

Most of the debugging tools listed above can be seen in Figure 5.6

Behavior when using non supported features in the C# shader If the code for a non supported method or value type would be written in the C# shader code the program would not compile and neither the simulation nor the translation would start.

It would be possible to give the path of a file not used in the project as a shader to the translator. If this code would contain a C# shader written in the correct syntax using methods of value types not supported in the example the translator would run and return a GLSL shader. The translator would simply keep the names of every method call and value type that is not defined by a "TranslationAttribute". If these kept names match the names of the methods and value types in GLSL the shader could be executed. If any of the names would not have a match in GLSL, the resulting code would not be compilable.

```
Setting uniforms
Uniforms set. Duration: 1,0674ms
Calculating vertex step
Vertex step calculated. Duration: 4,6791ms
Assembling primitives
Primitives assembled. Duration: 1,3903ms
Rasterization
Rasterization complete. Duration: 1374,6003ms
Calculating fragment step
Fragment step calculated. Duration: 201,6613ms
Cleanup
Cleanup complete. Duration: 0,859000000000151ms
TotalRenderTime:  1585,0868ms

Setting uniforms
Uniforms set. Duration: 0,1021ms
Calculating vertex step
Vertex step calculated. Duration: 0,841ms
Assembling primitives
Primitives assembled. Duration: 0,0931ms
Rasterization
Rasterization complete. Duration: 1244,9009ms
Calculating fragment step
Fragment step calculated. Duration: 175,0174ms
Cleanup
Cleanup complete. Duration: 0,252799999999979ms
TotalRenderTime:  1421,2686ms

Setting uniforms
Uniforms set. Duration: 0,0534ms
Calculating vertex step
Vertex step calculated. Duration: 0,7265ms
Assembling primitives
Primitives assembled. Duration: 0,7277ms
Rasterization
Rasterization complete. Duration: 1278,0488ms
Calculating fragment step
Fragment step calculated. Duration: 180,0066ms
Cleanup
Cleanup complete. Duration: 1,21229999999991ms
TotalRenderTime:  1460,9924ms
```

Figure 5.4: Permormance of rendering with the C# shaders on simulated pipeline on CPU

```
TotalRenderTime:  0,001ms

TotalRenderTime:  0,0012ms

TotalRenderTime:  0,0016ms

TotalRenderTime:  0,0012ms

TotalRenderTime:  0,0015ms

TotalRenderTime:  0,0015ms
```

Figure 5.5: Performance of rendering with the translated shaders on GPU

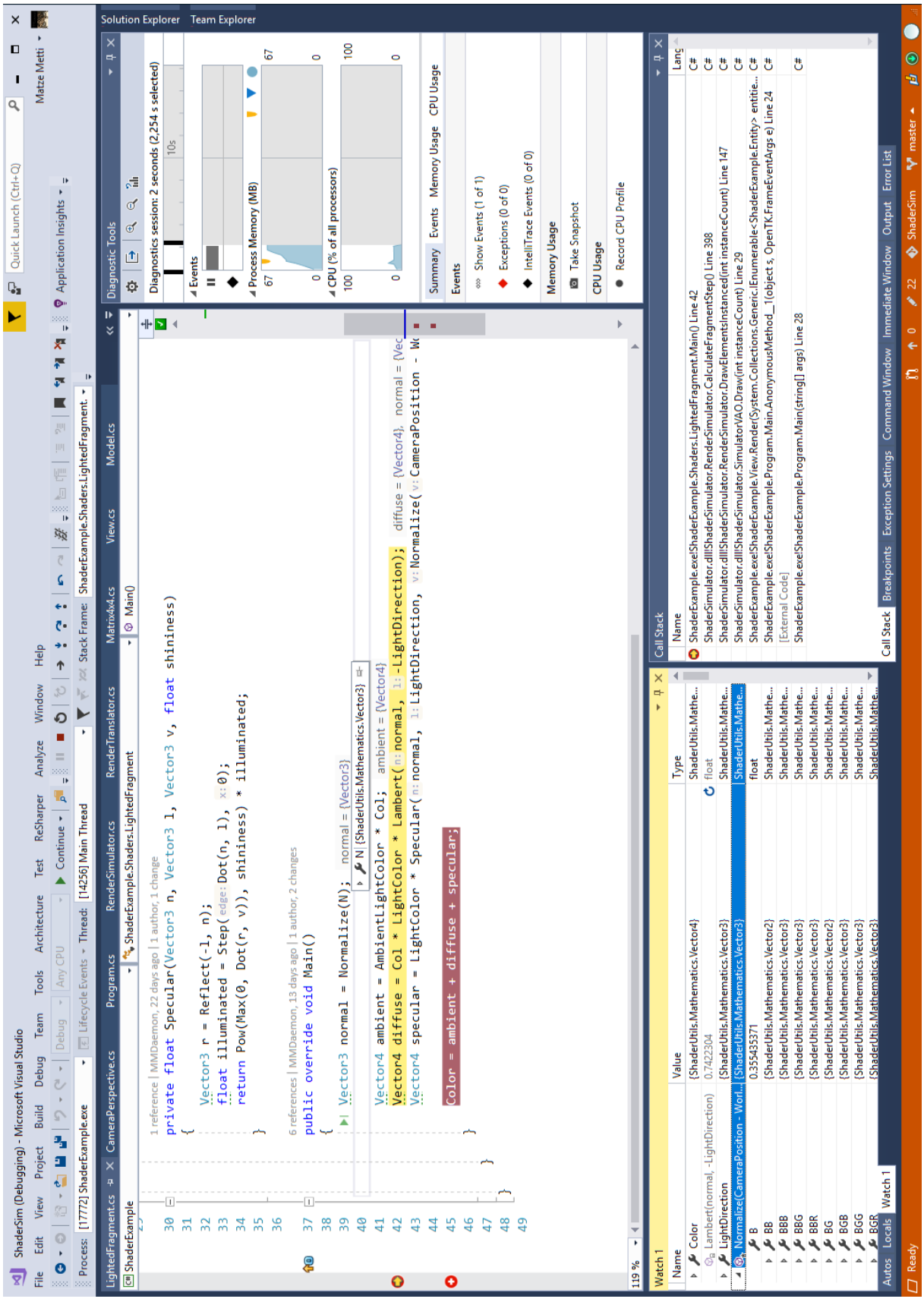

Figure 5.6: Debugging shader with different tools in VisualStudio: setting breakpoints, breakpoints with conditions, stepping through the code, variable inspection, navigating through the call stack

6 Conclusion

As shown in chapter 5 it is possible to enable debugging of shaders in the graphics pipeline by simulating them on the CPU. All objectives enumerated in section 1.4 are met:

- The different tools VisualStudio provides to aid in the debugging process can be utilized for the shader code written in C#. Thereby a large number of tools to assist debugging are usable.
- No specific graphics card or drivers are needed for this method.
- It can be switched between a mode where debugging is enabled and a mode where the full performance of the GPU is utilized.
- The render result per iteration does not have any noticeable differences.
- Calculating the full graphics pipeline with the shaders in the given example takes about 1.5 seconds per iteration. Consequently all desired debug points occurring in a frame can be reached within this time. This performance is acceptable for a user interface as stated in section 1.4.

The project implemented as part of this research is limited in its functionality, but it serves as concept for the practicability of the presented methods. It is a functioning method which could be refined and realized in a full tool supporting different source and output languages.

Possible modifications and improvements As stated in chapter 5 the biggest issue with simulating the graphics pipeline is the performance. This could be improved through multiple ways. The program could be implemented using multiple threads and thereby being able to calculate the simulated steps more parallel on the CPU. Even more improvement would be achieved by implementing parts of the simulated pipeline as compute shaders and running them optimized on the GPU again. For example the performance of the rasterisation process would benefit from this.

To get to the points in the shader faster where it is desired to be able to debug it would also be possible to limit the functionality of the simulated pipeline to this specific part.

If for example within the pipeline only the fragments resulting in a specific pixel of the output would be calculated the rest of the rasterisation step could be omitted resulting in far less calculation and time before the desired point to start debugging is reached. The output image of the pipeline would no longer be similar to the resulting image of the shader running on the GPU but the values for this pixel would still be calculated correctly.

It would also be possible to just implement parts of the pipeline. For example it would be possible to debug only a fragment shader. The input values for this shader could be generated without calculating the vertex data. A way to generate these inputs would be to simply give the fragments an input depending on the position of the fragment within the output raster.

Acronyms

CPU	Central Processing Unit
GPU	Graphics Processing Unit
IDE	Integrated Development Environment
JIT	Just in Time
AOT	Ahead of Time
MSIL	Microsoft Intermediate Language (also known as IL)

List of Figures

Listings

Bibliography

Bolton, Matthew L, Bass, Ellen J, and Siminiceanu, Radu I (2013). "Using formal verification to evaluate human-automation interaction: A review". In: *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 43.3, pp. 488–503.

Campy. URL: http://campynet.com/ (visited on 09/06/2019).

Ciardi, Francesco Cifariello (2015). URL: https://computergraphics.stackexchange.com/a/101 (visited on 09/06/2019).

Collins, Harper (2014). *Collins English Dictionary – Complete and Unabridged, 12th Edition*. URL: https://www.collinsdictionary.com/dictionary/english/debugging (visited on 09/06/2019).

Dave Shreiner Graham Sellers, John Kessnich (2013). *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3 (8th Edition)*. Addison-Wesley Professional. ISBN: 978-0321773036.

Fox, Alexander (2017). *MTE Explains: The Difference Between a CPU and a GPU*. URL: https://www.maketecheasier.com/difference-between-cpu-and-gpu/ (visited on 09/06/2019).

Franz, Dr. Maren (2006). "Die Subjektivität der Wahrnehmung". In: URL: http://www.nlp-hh.de/img/Subjektivitaet_der_Wahrnehmung.pdf (visited on 09/06/2019).

Fumero, Juan et al. (2017). "Just-in-time gpu compilation for interpreted languages with partial evaluation". In: *ACM SIGPLAN Notices*. Vol. 52. 7. ACM, pp. 60–73.

GLSL-Debugger. *GLSL-Debugger*. URL: http://glsl-debugger.github.io/#features (visited on 09/06/2019).

GLSLplusplus. URL: https://github.com/AlexanderDzhoganov/GLSLplusplus (visited on 09/06/2019).

GPU PerfStudio. URL: https://gpuopen.com/archive/gpu-perfstudio/ (visited on 09/06/2019).

ILGPU. URL: http://www.ilgpu.net/ (visited on 09/06/2019).

Jetbrains (2019). *Debugging Exceptions*. URL: https://www.jetbrains.com/help/rider/Debugging_Exceptions.html (visited on 09/06/2019).

Jones, Tim G. (2019). *HLSL Tools for Visual Studio*. URL: https://marketplace.visualstudio.com/items?itemName=TimGJones.HLSLToolsforVisualStudio (visited on 09/02/2019).

Jukka, Julku (2012). "Debugging GPU code". In: URL: https://wiki.aalto.fi/download/attachments/70779066/jjulku_debugging_gpu_code_final.pdf?version=1&modificationDate=1357205536000&api=v2 (visited on 09/06/2019).

Kernighan, Plauger (1982). *The Elements of Programming Style, 2nd edition*. McGraw-Hill. ISBN: 978-0070342071.

Khronos (2019). *Rendering Pipeline Overview*. URL: https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview (visited on 09/06/2019).

Lighthouse3d (2013). *GLSL Tutorial – Attribute Variables*. URL: https://www.lighthouse3d.com/tutorials/glsl-tutorial/attribute-variables/ (visited on 09/06/2019).

Magyar, Szilard (2017). *All you need to know on by reference vs by value*. URL: https://www.freecodecamp.org/news/understanding-by-reference-vs-by-value-d49139beb1c4/ (visited on 09/06/2019).

Microsoft (2016). *Debugging GPU Code*. URL: https://docs.microsoft.com/de-de/visualstudio/debugger/debugging-gpu-code?view=vs-2019 (visited on 09/06/2019).

– (2018). *Pipelines and Shaders with Direct3D 12*. URL: https://docs.microsoft.com/en-us/windows/win32/direct3d12/pipelines-and-shaders-with-directx-12 (visited on 09/06/2019).

Microsoft (2019a). *First look at the Visual Studio Debugger*. URL: https://docs.microsoft.com/de-de/visualstudio/debugger/debugger-feature-tour?view=vs-2019 (visited on 09/06/2019).

– (2019b). *Using automatic shader PDB resolution in PIX*. URL: https://devblogs.microsoft.com/pix/using-automatic-shader-pdb-resolution-in-pix/ (visited on 09/06/2019).

Miecznikowski, Jerome and Hendren, Laurie (2002). "Decompiling Java bytecode: Problems, traps and pitfalls". In: *International Conference on Compiler Construction*. Springer, pp. 111–127.

Nielsen, Jakob (1993). *Usability Engineering*. Morgan Kaufmann. Chap. 5. ISBN: 978-0125184069.

Nvidia (2019a). *Debugging Solutions*. URL: https://developer.nvidia.com/debugging-solutions (visited on 09/06/2019).

– (2019b). *OpenGL*. URL: https://developer.nvidia.com/opengl (visited on 08/28/2019).

Nvidia Nsight. URL: https://developer.nvidia.com/tools-overview (visited on 09/06/2019).

Peter Shirley, Steve Marschner (2009). *Fundamentals of Computer Graphics 3rd Edition*. A K Peters/CRC Press. ISBN: 978-1568814698.

Projektarbeit. URL: https://github.com/MMDaemon/ShaderSim (visited on 09/06/2019).

Richard S. Wright Benjamin Lipchak, Nicolas Haemel (2007). *OpenGL SuperBible: Comprehensive Tutorial and Reference (4th Edition)*. Addison-Wesley Professional. ISBN: 978-0321498823.

Scherzer, Daniel (2019a). *GLSL language integration*. URL: https://marketplace.visualstudio.com/items?itemName=DanielScherzer.GLSL (visited on 09/06/2019).

– (2019b). *NotepaddPP-glsl-integration*. URL: https://github.com/danielscherzer/NotepaddPP-glsl-integration (visited on 09/06/2019).

SharpShader. URL: https://github.com/Syncaidius/SharpShader (visited on 09/06/2019).

SPIRV-cross. URL: https://github.com/KhronosGroup/SPIRV-Cross (visited on 09/06/2019).

SpirvNet. URL: https://github.com/Philip-Trettner/SpirvNet (visited on 09/06/2019).

Turton, Lawrence (2017). *Interpreters, compilers, JIT & AOT compilation*. URL: https://medium.com/@avelx/interpreters-compilers-jit-aot-compilation-753fb2f9014e (visited on 09/06/2019).

Undo (2019). *What is Reverse Debugging, and why do we need it?* URL: https://undo.io/resources/reverse-debugging-whitepaper/ (visited on 09/06/2019).

Appendix

DVD containing the project