IT UNIVERSITY OF COPENHAGEN

Master Thesis

Designing a Dynamic Node Based Shader Tool for OpenGL.

Author: Antonios Nestoridis
Supervisor: Morten Nobel-Jørgensen, Postdoc

IT University of Copenhagen
Copenhagen, Denmark

A thesis submitted in fulfillment of the requirements for the degree of Master of Science.

August 2018

Abstract

The area of computer graphics in the recent years has managed to produce incredibly complex and photo-realistic effects, that are being used by a great number of applications throughout different industries. All this was rapidly achieved with the introduction of programmable 3D hardware and subsequently the manipulation of the graphics pipeline, with the use of shader programs. However, the authoring of shaders is a challenging task that requires highly specialized knowledge, even for seasoned programmers. This thesis attempts to simplify this process, by proposing a system that makes use of visual programming techniques that will assist the user in quickly prototyping and modifying shaders. This tool exposes all of the modern shader stages that are available in the graphics pipeline and presents a node creation interface where the user can expand its functionality, by authoring custom nodes. The decign decisions, implementation details, as well as the difficulties that emerged during development, will be discussed in detail to provide insight in the creation of such an application.

Keywords: Computer Graphics, Shader Programming, Visual Programming, Graphs , OpenGL

Acknowledgments

Contents

List of Figures

List of Tables

1

Introduction

1.1 Problem description

The area of computer graphics has seen a tremendous rise in the quality of the effects that can be achieved, since the introduction of programmable 3D hardware. This breakthrough provided the processing power and the software tools to manipulate it, in order to achieve results that in the past would not be possible in digital applications. Subsequently, many other areas have been further influenced by this escalation, with the most notable ones being the gaming and the film industry. Modern software manages to create photo-realistic effects and complex visual simulations both offline, in the case of films, and in real time for the vast and complex worlds of modern video games.

But even if graphics applications are so widely used, the theoretical background that is required for understanding the graphics pipeline, as well as learning how to manipulate it, is considerable. Even experienced programmers need to spend a lot of time in order to specialize around this knowledge, and even then, the process of authoring shaders can be challenging.

1.2 Goal of the project - Research question

In order to address the above problem, this project proposes a system that can assist with the easier modification and authoring of shaders, by making use of visual programming techniques. The goal is to provide an application that will allow users to prototype and compare shader effects faster and in a more abstract way, by focusing on the higher level manipulation of data in the pipeline, instead of the low level details of code authoring. Furthermore, the system will attempt to expose all of the available shader stages that are part of the modern graphics pipeline in this environment, and address the difficulties that might arise with their use.

The implementation is based around the notion of using a graph structure and nodes to represent the different shader operations and data. These nodes will be contained in a library that is initially populated by the developer but can be further expanded by the user.

1.3 Result Summary

The resulting application offers a visual node editor where the user can easily create and connect nodes, in order to produce, modify and save different shader effects swiftly. Changes in these results can be previewed in real time on a 3D model that was loaded with the use of a simple custom rendering engine, that was created for the purposes of this project in OpenGL and C++. Additionally, there is a node creation interface where the user can author their own custom nodes.

The system is not a complete and polished application, and is still restricted in many ways. From a lack of general usability features, to the

rather few available operations due to the size of the node library, as well as a restriction in the authored shader language that is used by the nodes.

1.4 Chapter overview

Following this short introduction, the rest of the chapters will follow the following structure:

- Chapter 2 presents an introduction to the theory that is essential to the comprehension of computer graphics and shader programming, as well an overview of the modern graphics pipeline.
- Chapter 3 contains an outline of related research in visual shader programming and similar applications, that served as an inspiration for this implementation.
- Chapter 4 presents the overall design of the system and its many components, along with its usage capabilities.
- Chapter 5 provides an in-depth view of the underlying implementation details of the tool and the techniques that were used to create its features.
- Chapter 6 examines the resulting system and discusses its strengths and weaknesses in more depth.
- Chapters 7 and 8 include the features that will be implemented in future development and the conclusion of the thesis.

2

Theoretical Background

Before presenting in detail the system that was created for this thesis, it is important to provide an overview of the modern computer graphics pipeline and shader programming to the reader that has never explored this area before.

This chapter is dedicated to the introduction of basic concepts, theory and keywords that are used throughout the chapters and are crucial to understanding the purpose and functionality of the tool. It is not meant to provide an extremely detailed explanation, but more of an overview of the graphics pipeline. The reader that is already familiar with these topics, may skip directly to the next chapter which presents related work on visual shader programming.

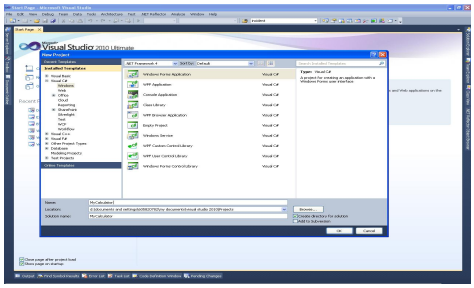
The following sections were authored using combined material from [4] [17] and [10].

2.1 The Modern Graphics Pipeline

A graphics pipeline is a model that describes the series of steps that are necessary for a graphics system to display something on the screen. That

can be either a 3D entity, like a digital model of a character in a game, or a 2D entity like a common window from the graphical user interface of an operating system.

(a) Screenshot from the AAA game title : God of War



(b) Visual studio application windows

Figure 2.1: Examples of computer graphics entities

In either case, the final goal is to convert these defined entities into a final digital image that can be displayed on a monitor. This process is called rendering and the abstract steps that describe it can be viewed in figure 2.4.

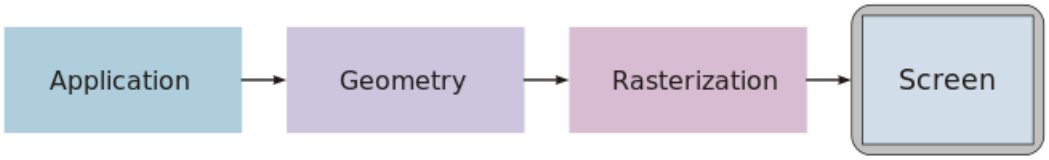


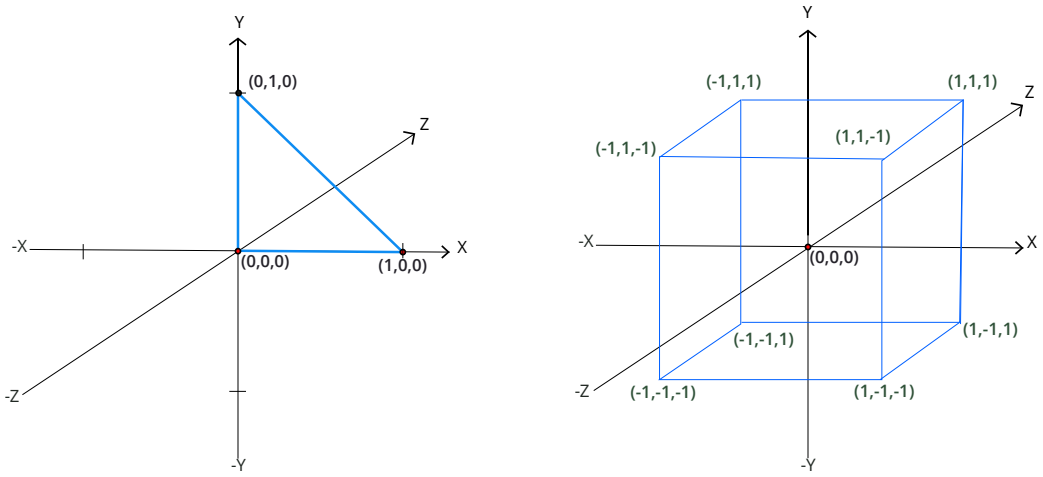
Figure 2.2: Application to screen overview

The geometry of those objects (along with other possible characteristics) is defined in a dedicated computer application which makes use of a graphics API, that can communicate with the hardware, mostly the GPU, and handle the rest of the rendering steps. A graphics programmer is responsible for handling this API and manipulating the rendering process to their desires and needs.

Game engines and 3D modelling tools are examples of such applications, that hide the complicated internal communication with the graphics API, by providing an easy to use interface for creating and manipulating objects.

2.2 Geometry definition

Objects are comprised of collections of data called vertices, that contain the geometric location of points in 3D space. If a user wanted to render a triangle or a cube on the screen, they would need to define the 3D coordinates of the points that define these shapes.



(a) Triangle in 2D space

(b) Cube in 3D

Figure 2.3: Geometry definition

In the example above, the triangle on the left was defined as a surface in 2D space by simply not making use of the z coordinate.

A collection of vertices may represent points but they can express different kinds of polygons or other simple geometric shapes which are called

primitives. Some of the most common are points themselves but also lines, triangles and quads. Ultimately, a rendered object is expressed in terms of primitives. The user apart from providing vertex coordinates also chooses the primitive (or topology) type in which the API will draw these vertices. The process of creating a primitive from a given set of vertices is called primitive assembly. It is handled by the graphics system automatically and may occur multiple times during the pipeline execution as we will see later.

After the geometry of a scene is defined, the graphics system will perform an automatic rasterization step in which it will convert primitives that are expressed in vector graphics, into discrete pixels that can be drawn on the screen.

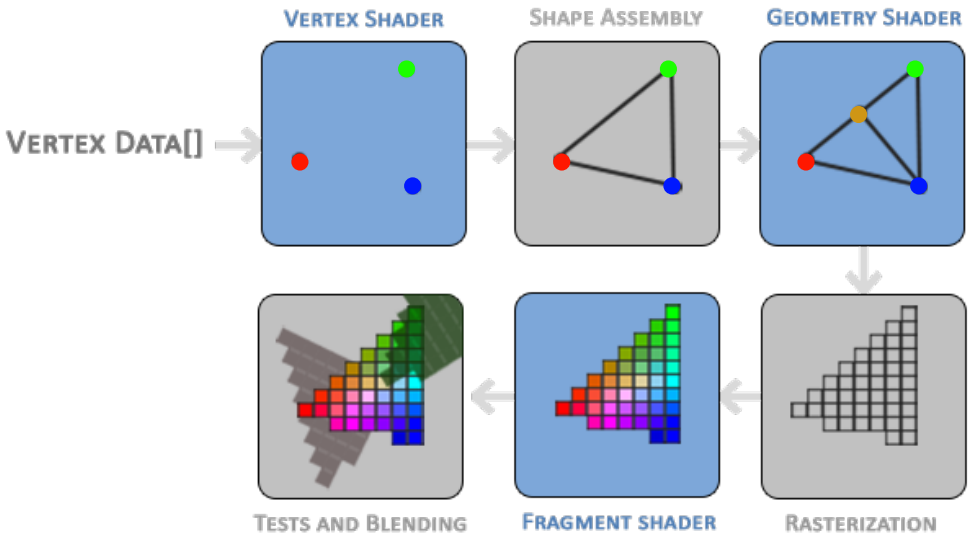


Figure 2.4: Application to screen overview

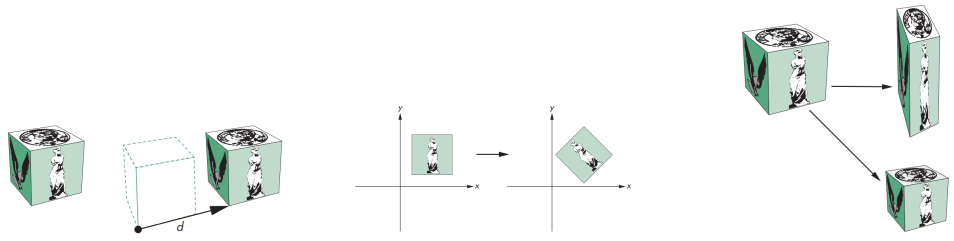
Apart from geometric information the user can define additional per-vertex data. These data are called vertex attributes. The most common type of data that is provided for each vertex is position,color information, a directional

vector which describes the orientation of a surface called a normal vector and is useful for many calculations and finally texture coordinates which map the color contents of an image to a primitive. Vertex attributes will be automatically interpolated for the interiors of primitives during rasterization.

2.3 Homogeneous coordinates

In order to avoid a possible confusion between a point and a vector, since both can be described with the current representation, graphics systems use a homogeneous coordinate representation instead, which is comprised of four components for the three dimensions that we are working on. The first three components x,y,z express the same information as before, while the fourth component w creates the distinction between vector or point with its value.

Homogeneous coordinates are important in computer graphics systems for various reasons. The most crucial transformations that are used for manipulating vectors and points, like translation, rotation and scaling, can be easily calculated with the use of 4-dimensional matrices and can be applied through simple vector-matrix multiplication. This of course means that the vector representation needs to be four dimensional as well for the operations to work.



(a) Translation (b) Rotation (c) Scaling

Figure 2.5: Common geometric transformations

The fourth component w is also important for the application of different viewing and projection methods on a scene, through an operation called perspective division. It is also crucial for the quick transformation between coordinate spaces. Both of these concepts are described in more depth in the next section.

2.4 Coordinate spaces

A coordinate system is the geometric space defined by a set of basis vectors (axis) and an origin point.

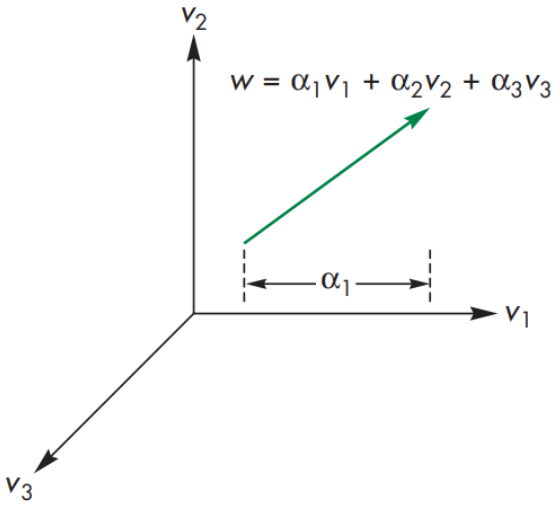


Figure 2.6: Vector derived from 3 basis vectors

An important aspect of vertices and primitives, is the coordinate system in which they are defined. A primitive will be rendered only if its coordinates fall inside the cube volume that has its center at the origin and expands from (-1,-1,-1) to (1,1,1), like the ones in example NAME. This is called the

Normalised Device Coordinates (NDC) space. All vertices that lie outside this volume will be discarded by the system. These coordinates are given to the rasterizer that will finally transform them to 2D pixels.

It is apparent that if a great number of primitives were to be defined in this coordinate system, either for simulating a complex surface or a complicated 3D scene that contains multiple entities, the task would become tedious and difficult. It would also mean that every user of graphics systems and 3D modelling application would have to adopt the exact same conventions when it came to axis representation (left or right hand system), orientations, origin etc.

For these reasons, the definition of coordinates is instead accomplished in a series of steps. These steps will transform the vertices through several intermediate coordinate systems before they end up in the NDC space, making the process more dynamic and ensuring that regardless of the conventions used, the final result will be the expected one.

Changing coordinate systems for a vector means to change its frame of reference in regards to the origin and the axis in which it was defined in. To put it more simply, it is its equivalent representation in a new system that has a different set of basis vectors. This is accomplished by simply multiplying the vector with a 4x4 matrix that represents the transformation from the old system to the new , just like the vector manipulation transformations in the previous section.

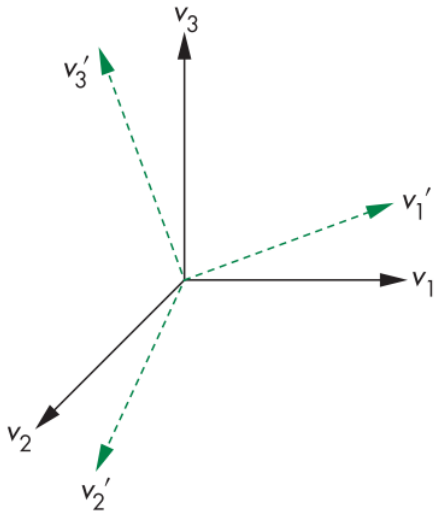


Figure 2.7: Rotation and scaling of a basis

There are six important coordinate systems commonly used in the graphics pipeline. Each of these intermediate representations offers advantages and easiness over certain operations/calculations. We will try to outline these advantages as we provide a short description for each one below. An overview of all the transformations can be seen in figure 2.8 below:

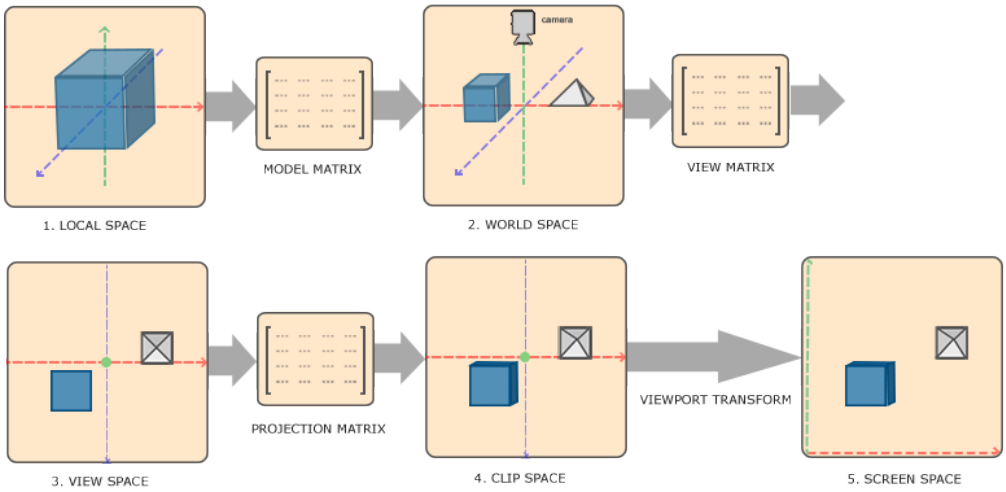


Figure 2.8: Overview of the coordinate systems involved in the graphics pipeline

- Object (or local) coordinate space

  These are the local coordinates in which a single model is created in, either from a computer application or by direct instancing in a file. Usually they are specified around the origin with a desired orientation and size. The range of the coordinates can be arbitrary and there is no need to limit it in any specific range, like in the NDC.

- World (or Model) coordinate Space

  The world space is the frame of an entire graphics scene, which may contain a great number of individual objects. These coordinates indicate where and how is an object placed in the world. Properties like position, orientation and scale can also be altered in this space, in order to fit the desired dimensions of the scene. For example, if we want to place a cube with a 45 ° angle at the position (5,5,0), we can either create a rotated cube at the origin in local coordinates and only perform the translation in the world coordinates, or perform both the rotation

and the translation of a non-rotated cube in world space. This separation of operations provides a lot of flexibility to the user and assists in the easier creation of scenes.

The matrix that performs the transformation from object to world coordinates is called the Model matrix.

- View (or Eye) coordinate space

  Graphics applications perceive the world through the eyes of a virtual camera. This camera views the scene from a specific position, angle and orientation which corresponds to a new coordinate frame called View or Eye space. The origin of this frame is the center of the camera's position and the axis are defined by its current rotation. This transformation places an object in the way that it would be seen from the point of view of the camera. The matrix responsible for this transformation is called View matrix.

- Clip coordinate space

  After bringing the vertices in eye space, a decision has to be made regarding which objects will be included in the final rendering and which ones will be clipped (omitted), hence the name of the coordinate system. This decision is made based on the inclusion or not of the transformed vertices inside the viewing volume of the NDC which was discussed previously.

  However, since this normalized viewing volume is not intuitive enough to use, we define a new viewing volume of chosen dimensions and shape. This volume is called a frustum and is represented by the Projection matrix which projects 3D coordinates from the view space to 2D coordinates in clip space.

  In order to convert these 2D coordinates in the normalized range that is expected by the system, an operation called perspective division is

performed to all vertices and divides their x,y,z coordinates by the w component of the vector, which effectively brings them to NDC. This operation is performed automatically by the graphics API.

- Screen coordinate space

  The final step in this pipeline, maps the normalized device coordinates to actual 2D positions on the screen, which are measured in pixel units, with the viewport transformation.

2.5 Shaders

The theory described in the previous section is fundamental regardless of the graphics system and the API used. This section will also provide crucial information in regards to programming the graphics pipeline but some parts might be more exclusive to OpenGL.

Computer graphics as a field is known for being able to create extremely photorealistic images and effects that have been and are included in many different mediums and applications. From complicated geometric animations and graphical user interfaces to almost realistic simulations of natural lighting for great AAA games in real-time. In order to achieve these effects, simply defining 3D geometry was not enough. Programmability of the graphics pipeline and communication with the hardware was essential. This led to the creation of programmable shaders.

Shaders are programs that rest on the GPU and have a really specific functionality. They are responsible for producing outputs from the inputs they receive, usually in the form of geometric information, but not exclusively. They are not allowed to communicate with other shaders in any other way than this input/output interface. They are injected in different parts of the

pipeline that was described in the previous section, in order to alter settings or perform new types of calculations during the conversion of the vertex data to final pixels.

There are different types of shaders that are used for the different possible locations in the graphics pipeline. Some of those are mandatory and need to always be present for the pipeline to work while some are optional. A brief introduction to the types and their functionality will be provided below.

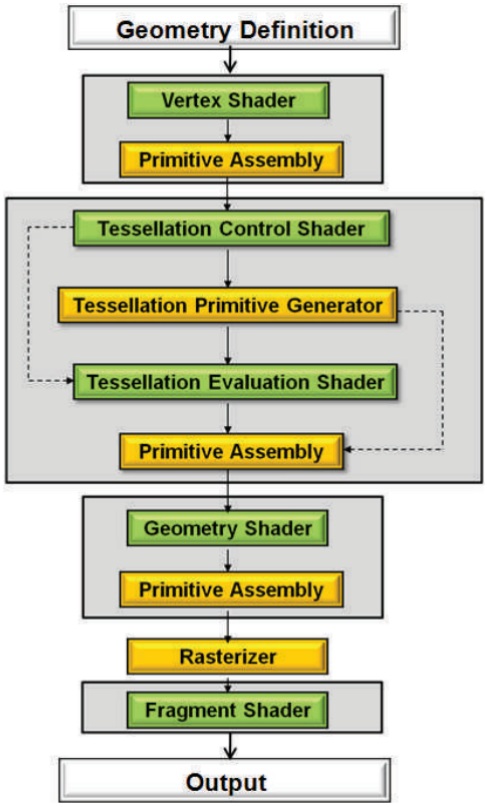


Figure 2.9: The expanded graphics pipeline

2.5.1 Vertex shader

The vertex shader is the first and one of the most important shader types in the program. It is one of the mandatory stages of the pipeline and its purpose is to perform transformations to the per-vertex attributes that it receives as input. This means that it operates on and outputs one vertex at a time and is invoked once for every vertex that was provided from the vertex data specification. Most of the 3D coordinate space transformations that were introduced previously take place during this stage, with the final vertex being converted from object to clip space, as depicted in figure before.

2.5.2 Fragment shader

The fragment (or pixel) shader is the last programmable stage of the pipeline before writing to the final buffer that will form the rendered 2D image. Along with the vertex shader, it is part of the minimum mandatory shaders that need to be provided by the programmer.

It operates on the fragments that are generated by the rasterization step of the transformed primitives and its goal is to calculate the final color of each fragment. Similar to the vertex shader, it operates on a single fragment per invocation and is invoked as many times as the number of fragments.

During processing, each fragment has data available from the previous vertex processing stage, which is usually the vertex shader, but not always as it will be described in the later subsections. These data are the interpolated per-vertex attributes that were described above.

2.5.3 Geometry shader

Geometry shaders are optional and are placed between the primitive assembly step that happens after the vertex shader or (the tesselation shader if it is used) and the rasterization step. Geometry shaders operate on a whole primitive and have full access to the vertex information that comprise it. They can output zero or a number of new vertices, and can modify the topology type of the received primitive. For example, even if the vertex shader generated triangles, the geometry shader can convert them back to points or lines.

It must be noted that a geometry shader cannot accept all of the available types of topology that can be generated either by the application or from previous shader stages. Therefore, they must be internally converted before they are given to the shader for processing. There are also restrictions regarding the type of topology that can be outputted by the geometry shader (points, line strips, triangle strips), as well as the number of new vertices that can generated.

2.5.4 Tesselation shaders

Tesselation is a process that subdivides existing geometry to smaller primitives, with the purpose of creating a more detailed and smoother surface. This step is optional and lies between the vertex shader and the geometry shader (or the fragment shader if the geometry shader is omitted). Tesselation shaders can create new geometry, much like the geometry shader, but they are not capable of modifying the received topology type and can only provide more of the same type.

Tesselation shaders require a new type of general purpose primitive called a

patch for inputs, which is basically a collection/grouping of a specific number of vertices. This number can be specified by the developer and it has an implementation based maximum limit that is at least 32. This allows the shader to work with a lot more vertices at the same time than the geometry shader. The output limitations are also a lot more flexible and the number of vertices that can be produced is much higher. This is why tesselation shaders are preferred over geometry shaders when it comes to level of detail (LOD) manipulation of the geometry.

Tesselation is divived in three stages, two of which are programmable shaders. They will be shortly described in the following subsections:

2.5.4.1 Tesselation control shader

The tesselation control shader (TCS) is responsible for setting the tesselation levels of a patch and preparing the final control points that will be used for the process during the execution of the tesselation evaluation shader (TES). Even if tesselation is used, the TCS can be omitted if the developer defines the tesselation levels manually from the main application.

The TCS outputs patches to the TES but the calculation process for each patch happens sequentially. Instead of outputting all the data in one invocation, like the geometry shader does, the TCS will be invoked once for every vertex in the output patch, calculating them one at at time. The size of the output is not necessarily the same as the input. Additionally, the whole input patch data is available for use on each invocation.

There are four outer tesselation levels and two inner that can be defined. Some or all of them will be used, depending on the type of tesselation pattern that the TES is using and will be described in the next subsection.

Between the TCS and the TES exists a fixed-function stage called tesselation patch generator (TPG) which creates the correct number of new tesselated geometry and provides parametric coordinates for them to the TES.

2.5.4.2 Tesselation Evaluation Shader

The TES uses the coordinates of the control points from the TCS output patches as well as the parametric coordinates that are provided by the TPG, to calculate the final 3D coordinates of the new geometry and possibly alter it. The TES also specifies some parameters for the tesselation, like the tesselation pattern (quads, triangles, isolines), the spacing between the tesselated segments and others.

2.5.5 Compute shaders

Even though compute shaders are not used in this project, we mention them for the sake of completion. They are the most recent shader stage to have been introduced in the pipeline and are mostly used for performing arbitrary calculations in the GPU which might not be directly related to the rendering process but may use or output information to the rendering pipeline.

2.6 Shader program object

Shaders are grouped together to form a shader program object, which when activated in the application, will use the pipeline created by the linked shaders to render objects. This program needs to have at least a vertex

and a fragment shader as stated above, and may use any of the optional stages. When linking shaders together the API is linking the outputs of a shader stage to the inputs of the next. Mismatch between those leads to linking errors.

2.6.1 Type Qualifiers

Apart from vertex attributes, there are two other important types of variables that circulate information.

Varying variables have been mentioned before, and are the data that are passed between shader stages through the input/output interface. They have to explicitly declare the direction of their information flow. Attributes for example, are essentially varying variables that are given as input to the vertex shader but from the application instead of another shader stage.

Uniform variables are special data that can be sent directly from the application to the shaders in the GPU, and are global and unique to the scope of an entire shader program. Every shader stage at any point has access to them and their values will not be changed unless they are updated by the application.

The concept of these variables are the same in all graphics APIs but the naming might differ.

2.7 Shading languages

Depending on the graphics API a developer is using, there is a corresponding programming language for shader development.

The most common shading languages and their corresponding APIs that are used in real time rendering can be viewed in the table below :

Graphics API	**Shading Language**	**Developer**
OpenGL	GLSL	The Khronos Group
DirectX	HLSL	Microsoft
NVIDIA	cg(discontinued)	NVIDIA
Playstation 4	PSSL	Sony
Metal	MSL	Apple
Stage3D	AGAL	Adobe
Vulkan	SPIR-V	The Khronos Group

Table 2.1: Shading Languages

3

Related Work

Creating a node based interface is not an original idea in computer graphics applications. The complexity and the steep learning curve of shader programming, created an early need for tools that make the use or learning process of shaders more approachable by all types of users. This is why there are many embedded or standalone nodal tools in the industry, as well as scientific research regarding visual shader programming.

The following sections will present some of the most notable research that has taken place throughout the years, as well as some of the most common commercial tools that served as an inspiration for this project.

3.1 Visual Shader Programming evolution in research

Robert L.Cook [8] was one of the first people to try and move away from fixed models in shading (which was the standard at the time) by proposing the idea of shade trees, shade languages and modular shading components. Shade trees were a flexible and dynamic tree structured model, that calculated the final output color for different types of surfaces. Basic operations like

dot products, normalization, reflection etc, were translated into nodes that receive and output appearance parameters to each other and serve as the building blocks for computational trees that can describe different surfaces or lighting effects. These parameters can be any type of geometric, material or environmental properties. One drawback of these methods was that the node connections and the creation of the trees had to be authored by hand with the use of a specific new shade language and therefore did not have the benefits of a visual interface.

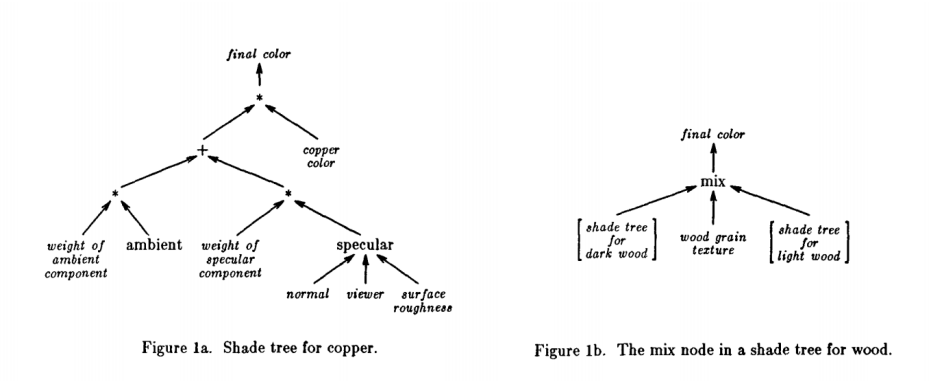


Figure 3.1: Shade Trees

Abram and Whitted [7] created the first visual implementation of Cook's shade trees that did not require code authoring. This implementation provided the user with a visual editor that could modify connections and parameters in a node network, in order to create complex shaders without exposing their inner implementation. It also provided quick feedback of the user's changes through a preview window that rendered the new shade effects on a sphere model when instructed.

Following that, Goetz et al [12] presented a graph system based on the above work but built for the modern technology of the time. Its goal was to provide the means for visually developing complex effects for the newly introduced vertex and fragment shaders of the OpenGL API. The resulting

topology of the graph that represents the final shaders, were stored in XML format with the intention of possible integration with XML based 3D formats like Extensible 3D (X3D). These results could also be later translated to the Cg shading language. The system offered a low abstraction level since the nodes represented functions that were more or less equivalent to operations from Cg.

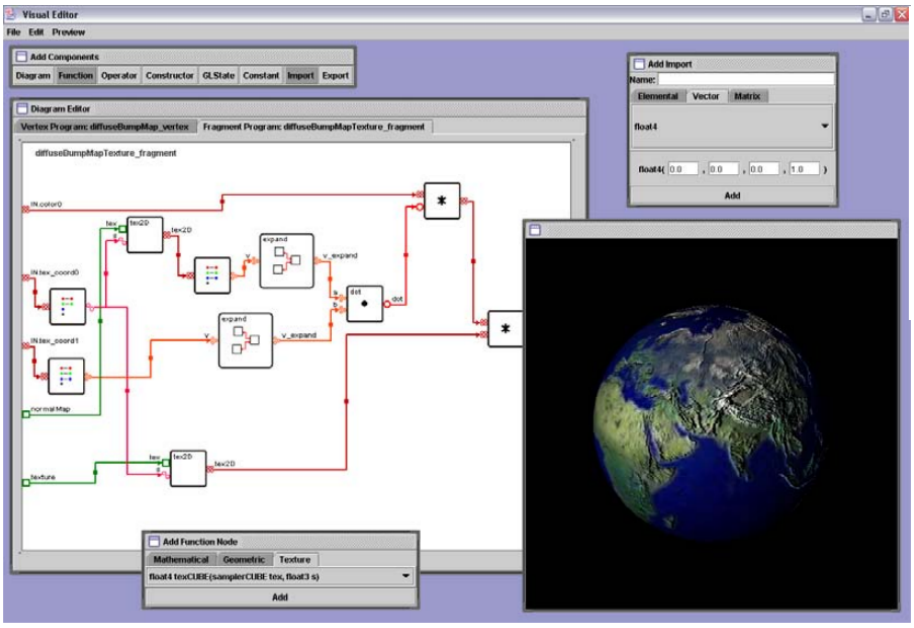


Figure 3.2: An XML-based Visual Shading Language for Vertex and Fragment Shaders (Goetz, 2004)

In order to address the problem of coupling between the visual elements and the low level programming functionality, McGuire et al [14] proposed the usage of abstract shade trees. Their aim was to create a system that does not require any programming knowledge, something that its predecessors did, but can be used by artists directly. It works with high-level rich types as nodes called atoms, that encapsulate information like semantics, transformation space, dimensionality, length etc. The user connects existing

effects together, which are sub-graphs comprised of said atoms, with a single edge that creates a data-dependency flow between them. The system will then automatically create appropriate internal connections between output and input components from those connected effects on the lower level, based on the type definitions of the I/O components and their semantic similarity. One drawback of this abstraction level is that the lack of details can make some effects more difficult to comprehend, since the user has no knowledge of the actual internal data connections.

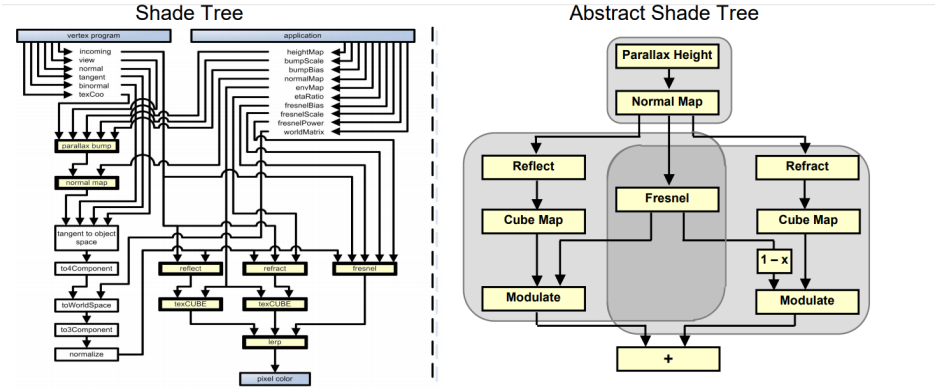


Figure 3.3: Abstract Shade Trees

Shortly afterwards, Jensen et al [13] created another system for interactive shader development through visual methods. This was closer to the implementation from Goetz et al [12] in the sense that the node slots that are exposed are closer to the lower level shader data (attributes, variables etc). Furthermore, the implementation tried to improve on previous difficulties/issues by using intelligent variables in the connection slots, that maintain information about the data type of the variable and the space in which it was defined. The editor will allow connections between variables of the same type and will automatically make the appropriate transformations if their mathematical space differs. The paper also proposed some shader optimization techniques for code placement and use of minimal transformation costs.

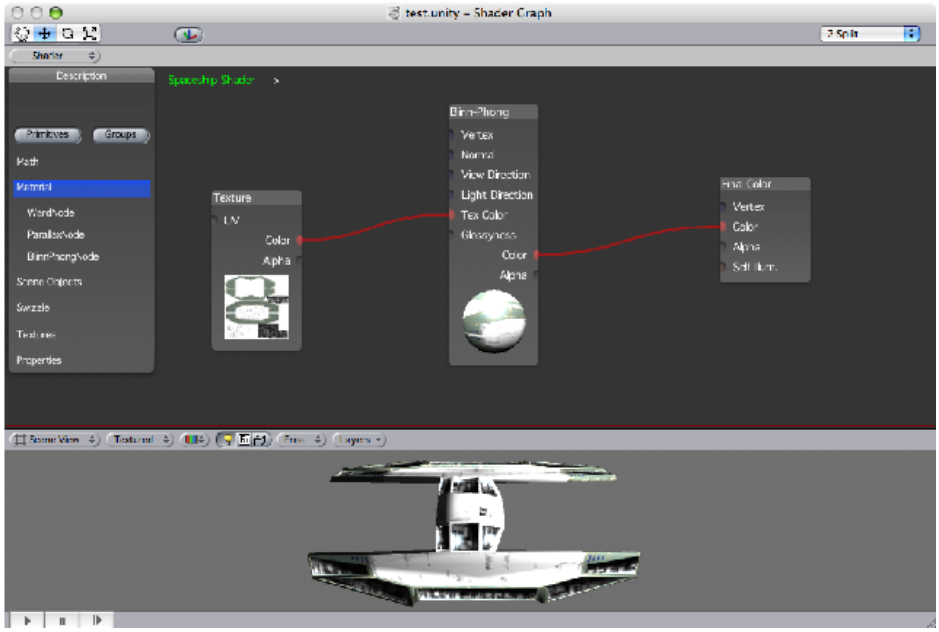


Figure 3.4: Interactive shader development

In his master dissertation thesis, Fitger [11] designed and implemented a similar system for the visual authoring of vertex and fragment shaders in OpenGL, that tried to improve the usability of the previous systems by non programmers. The system investigated the integration of the resulting shaders with third party applications in a more generalised way, since previous systems were highly coupled only with specific systems. It also provided a modular node implementation interface in which users can easily extend the current node library of the tool using XML. Moreover, it handled the issue of data type checking between connections, through an automatic type conversion library that checks if a conversion between the specific data types is possible or not and additionally locate the computationally cheapest.

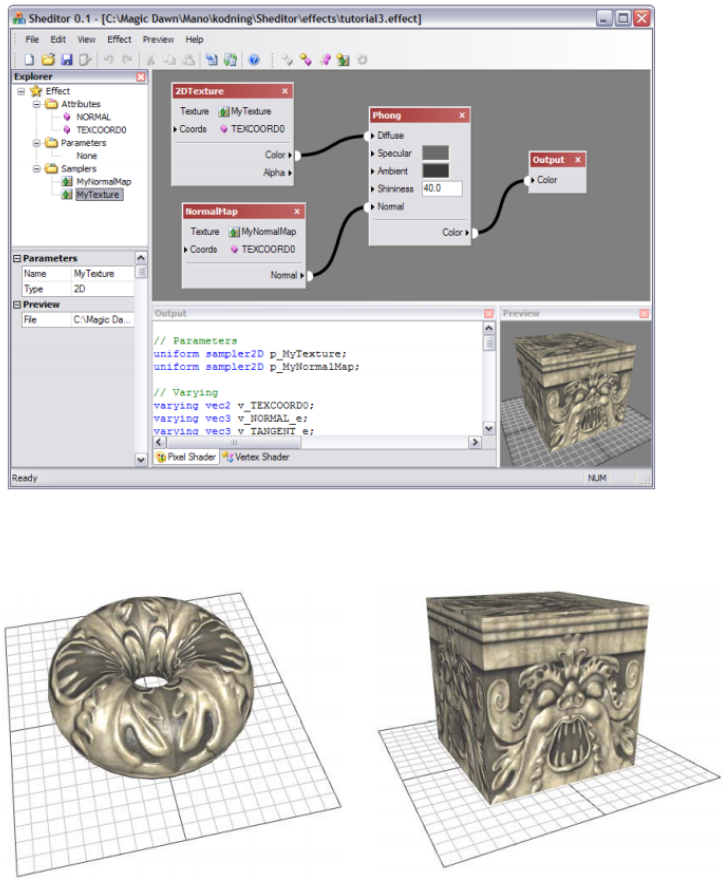


Figure 3.5: Visual shader programming

A more recent approach by Vergne et al [16] was Gratin. It is a programmable node-based system tailored to the creation, manipulation and animation of 2D/3D data in real-time time on GPUs. It targets different types of users, by allowing experts to customize shaders by hand in a lower and detailed level (generic nodes), while non-experts make use of the higher level visual environment that does not require deep technical knowledge of shaders to use. The node library is restricted, in the sense that each node encapsulates one or many shaders that perform complicated effects or higher

level data (e.g images), without providing different node types for low level parameters like float or custom vector values. Due to its modern development time, Gratin makes use of all modern shader stages as well both in the custom syntax editor and through the predefined effects in nodes.

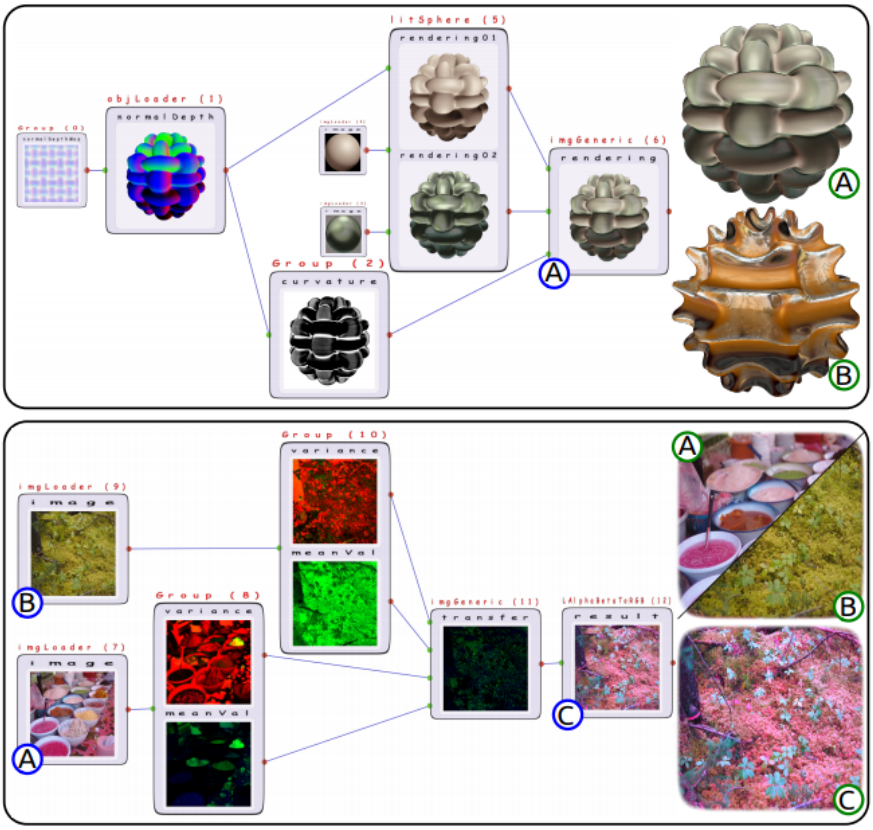


Figure 3.6: Developing Grating

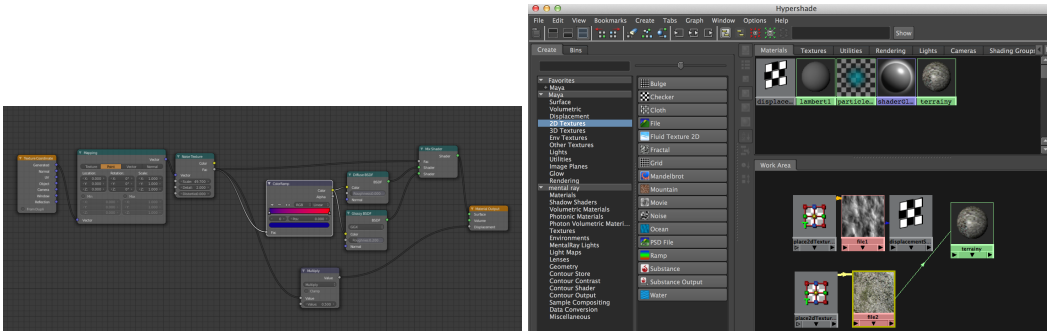
3.2 Commercial visual shader development tools

Apart from the research and the systems introduced in the previous section, there are many commercial tools that make use of real time visual shader de-

velopment. These are usually embedded systems in much larger applications, like 3D modelling software or game engines, with the purpose of assisting non programmers and artists to create shader effects faster and easier.

Most of them are naturally not open source and access to their low level implementation details and architecture is not available. But useful knowledge can still be obtained from the development environment that is offered to the users.

3D modelling software like Blender [1], Autodesk Maya and 3ds max [2], or texturing software like Substance, offer similar functionality with their nodal tools who are mostly targeted towards artists and have a more high level approach. They offer a wide range of premade shader effects that the user can parameterize and mix together, without the need for creating complex low level pipelines, all while being easy to learn and use. In contrast to the Autodesk products, Blender is open source and free to use and edit.



(a) Blender (b) Maya

Figure 3.7: 3D modelling tools

These tools also offer good intergration of the produced shaders and effects, with most game engines.

Popular game engines, like the Unreal Engine from Epic games [6], Unity3D

[5] or the open source Godot3D also have embedded nodal tools for shader creation.

Unreal was one of the first commercial products to adopt visual graph techniques since its initial development, not only for shader/material editting purposes but for most of its other components through the use of the blueprint subsystem. Its visual material editor offers a wider range of options to the user, both high (predefined shader effects) and low level (math operations and numerical parameters), through its extensive node library. Some of the node effects and general settings that are offered, make use of modern shader stages like geometry and tesselation but on a higher level of abstraction. Unreal Engine made the source code available to the public a few years ago, even though it still remains a licensed product.

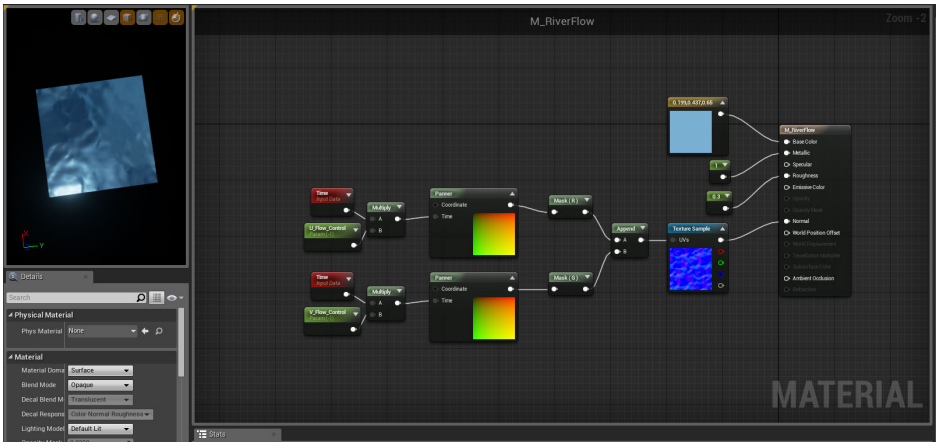


Figure 3.8: Unreal engine's blueprints

Unity3D only recently exposed more configuration over its rendering pipeline to the public and also introduced a node based visual shader editor, ShaderGraph. It is still in its early steps of development and does not offer as an extensive node library or options as the Unreal Engine, but it is still an easy to use practical tool that can provide insight for modern visual shader devel-

opment. Unity is free to use even for commercial use, up to a specific point, but the full source code is not available to the public. However, in the case of ShaderGraph and the new Scriptable Rendering Pipeline, the repository is available to the public.

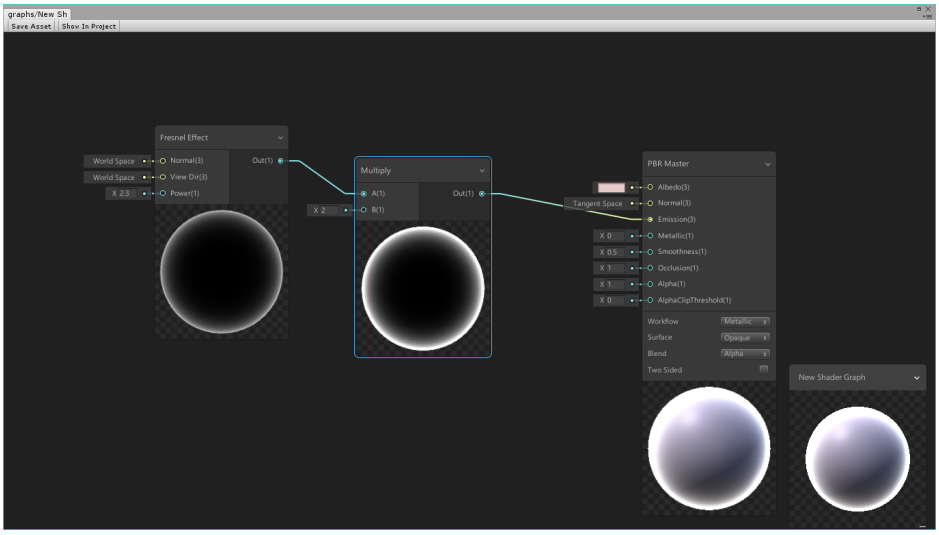


Figure 3.9: Unity's shadergraph

Godot's visual editor was initially introduced as a feature in one of its previous versions but was later removed and is not available in the current release. It is expected to return in future patches.

3.3 Non visual shader development software

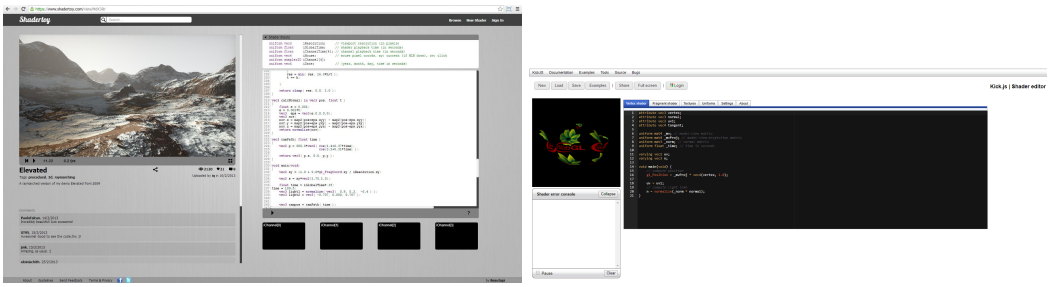
Apart from visual based methods, there is a plethora of tools for syntactical shader authoring. These are mostly targetted towards programmers, both amateur and expert, that have a level of understanding of the graphics pipeline and can effectively modify shader code to quickly prototype their

ideas. Most of these tools are web browser based and use the WebGL API, which is a cross platform simplified version of OpenGL but tailored for use in browsers.

Tools like ShaderToy [3] and GLSL Sandbox offer shader editing only for fragment shaders, while previewing the results in a different viewport. The user can choose to make their shaders public and add them in a huge library, where other users can view, rate or comment on them.

Other more sophisticated tools like KickJs [15] and Shdr, offer the additional option of editing vertex shaders, as well as other general API settings like blending mode, culling face etc.

Web 3D applications unfortunately do not have yet access to the modern shader stages of native APIs.



(a) ShaderToy (b) KickJs

Figure 3.10: Non visual shader editors

4

Design

4.1 Overview

Heavily inspired by the related work presented in the previous chapter, the design/implementation of the tool was made with a similar structure and functionality in mind. There are of course many differences, extensions or drawbacks/limitations which will be noted when necessary.

The general aim of the system is to offer flexible shader modification through the creation of a visual graph. Each node in this graph represents either data for use or operations to be performed in shaders, in the form of source code snippets (text strings). Connections between these nodes, manipulate the dataflow by feeding the results of one operation as inputs to others, resulting in chains of calculations that lead to more complicated effects. These effects can be reviewed and changed in real time with low effort.

Apart from the set of operations offered by the current version, the system also offers the possibility of extending its functionality through an implementation interface for creating custom nodes.

4.2 The Graph system

The structure that enables the desired modular shader modification mentioned above, is a number of nodes connected to each other to create a large graph that corresponds to chains of complex calculations.

Since this flow of data is directional and cycles in the graph are not desirable, because of logical and syntactical inconsistencies that might occur, it is categorized as a Directed Acyclic Graph (or DAG for short). This is not a surprise since DAGs are commonly used in most approaches towards visual shader programming. [11] [7] [16].

An example of a simple effect created with the visual graph interface can be seen in figure 4.1.

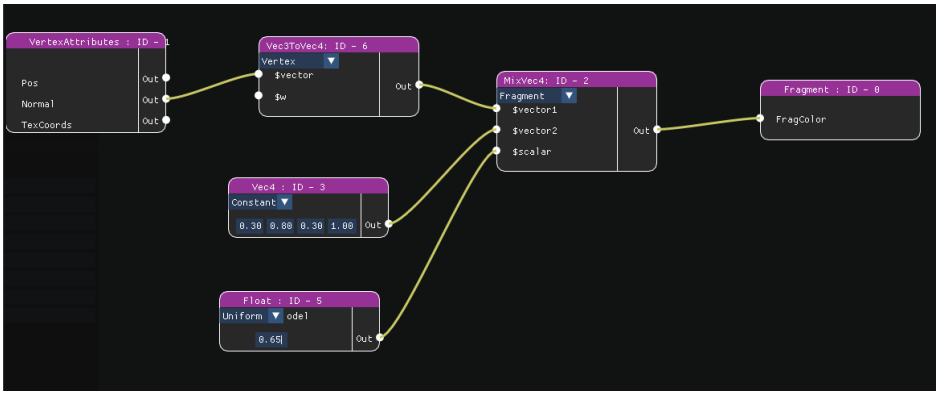


Figure 4.1: An example of a graph calculation

The example above indicates that information flows from the left to the right and therefore the nodes must be evaluated with an equivalent priority. This is expected since nodes that will pass an output value to other nodes need to be evaluated first. It should be noted that an output slot can be connected to multiple different inputs but since it is ambiguous to receive

an input value parameter from multiple sources, input slots have only one incoming connection.

In order to achieve the above notion of priority evaluation, the system performs a depth first traversal algorithm logic in the opposite direction, starting from the root node of the graph that lies at the farthest right and propagating all the way to the nodes to the left. This root node will always be the final entity to be evaluated and is responsible for accumulating all the source code that flowed from the graph operations and appending them in the appropriate segments of each shader code. This algorithm is examined in more detail at a later section.

4.3 Data types

Each node slot has a specific data type related to it. This either describes the data type that this slot will output or what kind of data it expects to be passed to it. The choice of which data types to include in the tool, was made based on the most useful and common types that are part of GLSL. These are the following :

Data Type	Description
bool	Conditional type
float	Floating point scalar
int	Integer value
vec2	Vector of 2 floating points
vec3	Vector of 3 floating points
vec4	Vector of 4 floating points
mat4	4x4 matrix of floating points
sampler2D	2D texture sampler
samplerCube	Cube map sampler

Table 4.1: Data types

Extending this list to support more types is possible but only from the backend of the system.

A connection between an output slot and an input slot is valid and is created only if they both have the same data type assigned to them. Otherwise the connection is simply rejected. An automatic type conversion system that can assist with transforming data types that are similar, like for example a float to an integer or vice versa, is not available in the current version.

Input slots that do not receive their value from another node, are automatically initialized to the default value of their data type, except for Sampler2D and samplecubes, which require an input node to be present.

4.4 Node architecture

All nodes inherit from a generic node interface, that contains information that is common to all of them, like the name of the node, its unique ID in the graph, the shadertypes that this node is allowed to be executed in and other.

The generic structure for a node can be seen in figure 4.2:

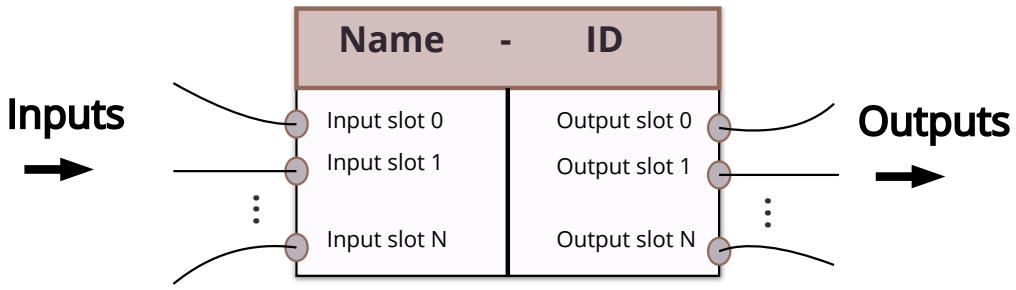


Figure 4.2: Generic graph node structure

The details of this structure are altered depending on the type of the node. There are four available basic types:

1. Input Nodes
2. Function Nodes
3. Output Nodes
4. Shader Nodes

4.4.1 Input nodes

Input nodes represent any kind of data that is used as input to a shader. They only have output slots since they act as sources of data. Their main purpose is to initialize and modify arbitrary variables that can be used as inputs to other nodes.

There are different input nodes for every data type from table 4.3 like float scalars or vectors, whose values and shader type qualifiers (e.g constant, uniform) can be directly modified by the user from the GUI. There are also

special input nodes that represent built-in attributes for each shader type, like the position or the corresponding normal for a vertex from a vertex shader or the invocation index from a tesselation shader. Finally, there are input nodes that expose useful global scene information from the application, like transformation matrices (Model, View, MVP etc) or the current time value of the system.

Input nodes that express local constant values have their current execution shader value change to the type of shader that they are connected to. This is not necessary for uniform variables, since they are globally scoped to the program and can be used by every stage, or for attribute nodes who are automatically assigned to the corresponding shader type that they expose.

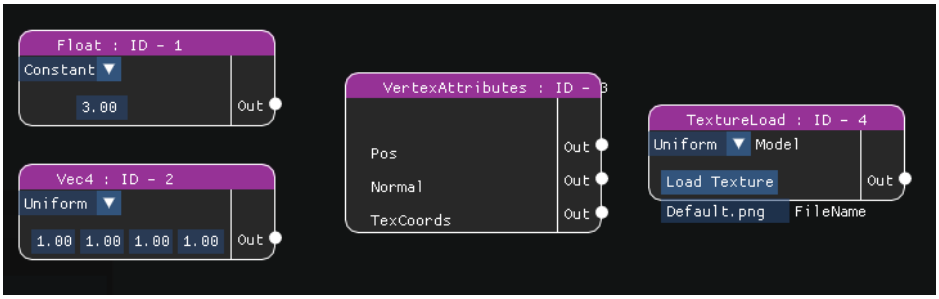


Figure 4.3: Examples of input nodes

4.4.2 Function nodes

Functions nodes are responsible for any kind of shader effect or formula implementation that the user wants to perform. They include everything from mathematical and logical operations for a specific data type, to geometric displacement, texture manipulation, lighting calculation algorithms etc. These operations will be performed either on the incoming data from the inputs slots or on the default values of the data types, and will produce one or multiple results on their output slots. The number of input and output slots can

vary depending on the operation.

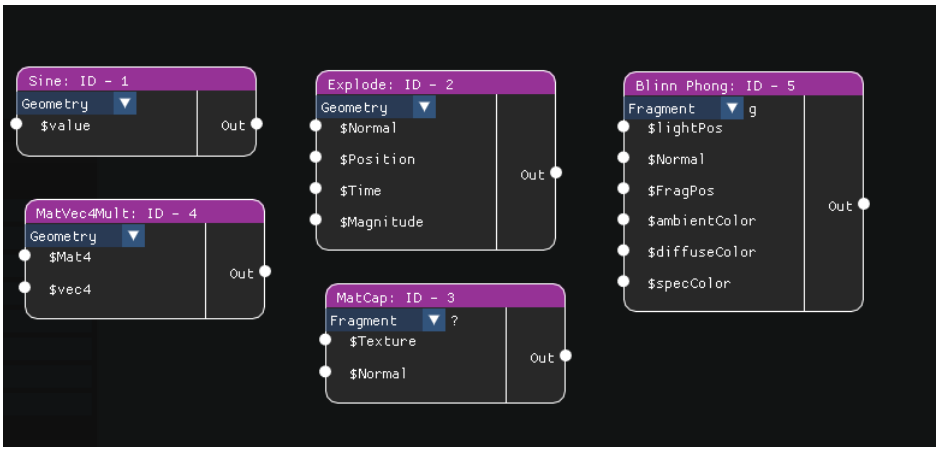


Figure 4.4: Examples of function nodes

Some operations may only be performed in certain shaders, since they require access to data that is exclusive to those stages, while others may be performed at any stage without issues. An example is the screen space normal effect which can only be used by the fragment shader, while mathematical operations like a vector3 addition can be performed at any stage.

For this reason, the user may change the current execution shader of the node in order to produce the desired result. Some possible issues with the shader stage order of connected nodes may arise. This will be discussed in more detail in a later chapter.

4.4.3 Output Nodes

Output nodes have the opposite functionality of input nodes. They are nodes that only receive information, therefore they only have input slots. They serve as converging ending points for calculations, where the exposed vari-

ables of the input slots will be matched to the incoming data.

4.4.4 Shader Nodes

Shader nodes are specialized output nodes that represent the different shader stages. They expose built-in variables and unique settings for the corresponding stage that the user may want to modify, like tesselation levels or the color output of the fragment shader. There can only be one of each of those nodes per graph.

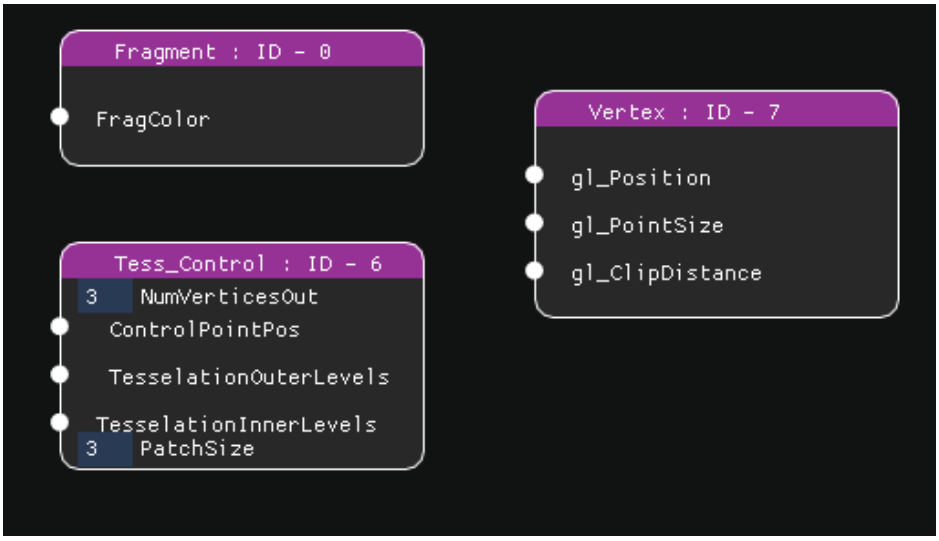


Figure 4.5: Examples of shader nodes

4.4.5 Node implementation interface

One desirable feature that was part of the design of the tool since the early stages, was the opportunity to customize and extend the functionality of nodes through the use of a simple node implementation interface, an idea

inspired by Goetz et al [12] and implemented in a similar fashion by Fitger [11]. Instead of using the XML mark-up language that was used in these systems, the interface uses the Javascript Object Notation (JSON) to represent the node structure parameters.

There is one JSON file in the project directory for every basic type of node, that contains a list of JSON objects with specifications for every different node of the said type. These lists have been initialized with nodes that were used during development and cover common operations.

The tool parses each of them once when the application starts and creates a library in the form of maps. When creating a node, the system will retrieve the appropriate data from this library at run-time.

In order to create a new custom node, the user is required to add a new object entry to the appropriate file and list, while following the format of that specific type. This must be performed before the application is launched, since the parsing of the JSON files happens only once as mentioned earlier.

An example of the structure of a JSON object can be seen below:

```
{
 "NodeType": "Function Node",
 "Name": "AdditionVec3",
 "Category": "Mathematical",
 "AvailableShaders": [
             "Vertex",
             "Tess_Control",
             "Tess_Eval",
             "Fragment",
             "Geometry"
  ],
  "Slots": [
        {
         "SlotType": "Input",
         "SlotName": "$a",
         "VariableType": "vec3"
        },
        {
         "SlotType": "Input",
         "SlotName": "$b",
         "VariableType": "vec3"
        },
        {
         "SlotType": "Output",
         "SlotName": "Vec3Add",
         "VariableType": "vec3"
        }
  ],
  "Code": [
   "vec3 Vec3Add = $a + $b ; "
  ]
}
```

Listing 4.1: Function node implementation interface

In the example above, we define the function node "AdditionVe3" which outputs the addition of the two vec3 inputs. This specific node will be listed in the subcategory "Mathematical" which contains math operations (see The function node subcategory list for all available categories) and can be performed at any shader stage in the pipeline.

Depending on the basic type of node, the fields of the corresponding JSON object are slightly different. For example, the field "Code" is not a part of the input node interface, since input nodes do not specify custom shader operations.

It is important to note that all of the key/value pairs must be filled out. It is also important that the user pays close attention to syntax, lower-upper case letters and correct name matching between variable references in the code to avoid potential errors and ensure that the operation behaves exactly as expected.

4.5 Saving results

The user can choose to save the effects that they have created with the tool at any point. The save/export button will simply save the current state of the shader program, by creating a folder in the project directory that contains five GLSL text files, one for each shader stage. No code optimizations or further modifications are performed to the saved result.

4.6 Visual user Interface

The GUI is pretty straightforward and simple to use but offers only a limited amount of actions to the user. The focus of the development was mostly towards creating the underlying system and less towards improving usability.

The interface consists of an application window that contains the sceneview viewport and a panel for the visual shader editor graph.

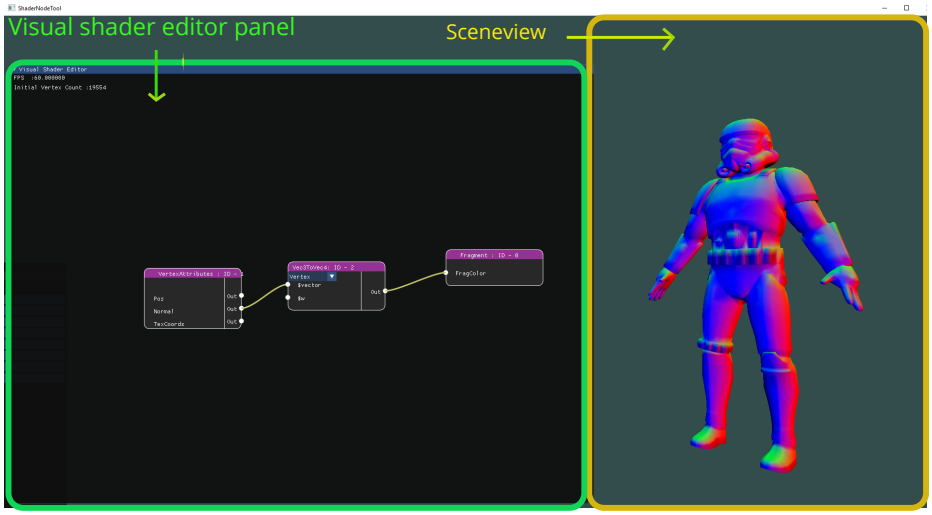


Figure 4.6: GUI overview

The sceneview is used to display a 3D model of choice using the active shader program.

In the visual editor panel the user can manipulate the graph by creating and connecting different types of nodes. Changes to the graph modify the shaders of the program that is currently used for rendering and the results can be viewed in the sceneview in real time.

Right clicking anywhere in the panel background, will display the creation

menu which the user can navigate to choose the specific node they wish to add to the system. This list is populated according to the maps that contain the names and initialization parameters of the nodes. The choice is made by simply left clicking on the appropriate menu item. Left clicking on the background will close the menu with no changes.

The newly created node will appear at the current location of the cursor. Nodes can be moved and reorganized inside the panel by clicking and dragging them from their title boxes

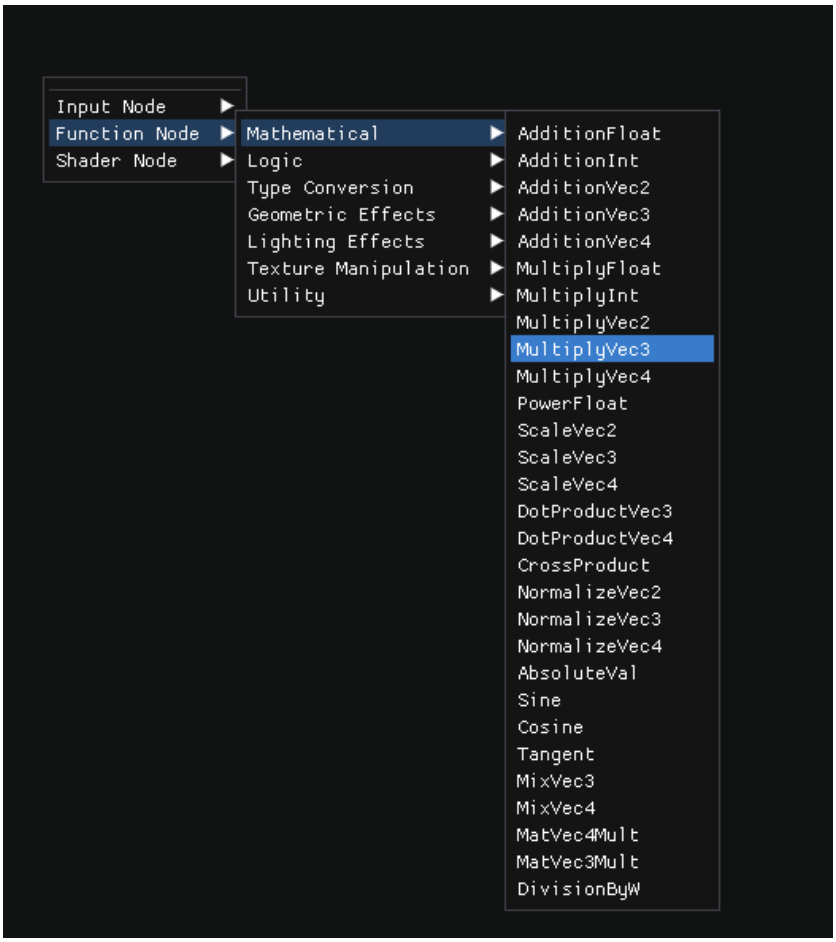


Figure 4.7: Node creation menu

In order to create a connection, the user has to simply click either on an input or output slot of a node, drag the cursor to the corresponding slot of the target node they wish to create the connection with and release the button. If the mouse is released anywhere on the panel other than the position of a slot, then no change will be made to the graph. As mentioned earlier, output slots can be used for multiple connections while input slots only have one incoming connections. This means that when trying to create a connection with an input slot that is already connected to a node, the system will replace the old connection with the new.

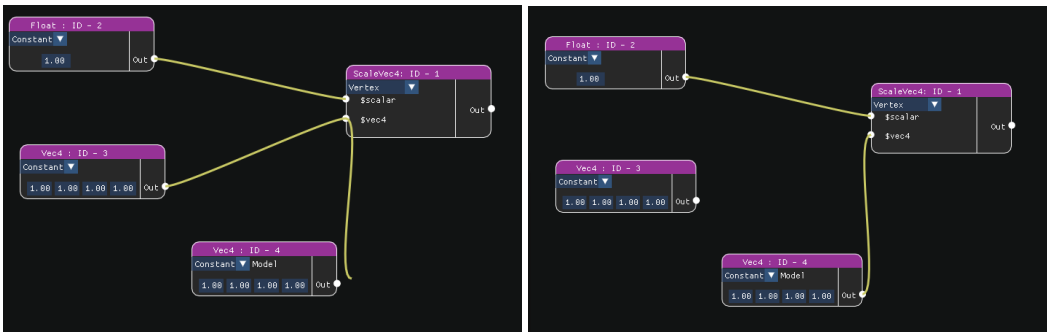


Figure 4.8: Replacing a connection

To delete an existing connection the user needs to click and drag starting from the input slot side of it and simply releasing the mouse anywhere on the background of the panel, replacing essentially the old connection with an empty one.

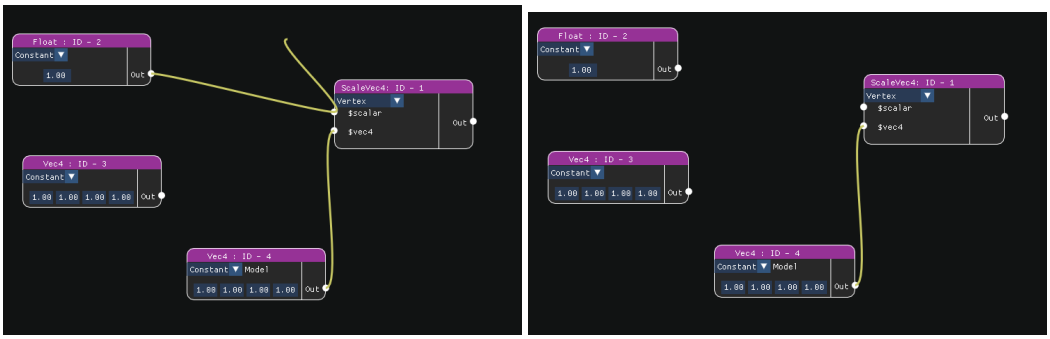


Figure 4.9: Deleting a connection

5

Implementation

This chapter will examine in more depth the actual implementation details of the system. We will start with an overview of the key technologies used for the development. Then we will present the system architecture and elaborate on the details of the shader structure and the compilation process of the graph. Lastly we will take a look at how the visual interface was created.

5.1 Overview

The overall system was built using C++ and Microsoft Visual studio. For the custom made engine responsible for the rendering we used the Open Graphics Library API (OpenGL) and subsequently the OpenGL shading language (GLSL). The system does not have an abstraction layer between the source code functionality and the shading language, meaning that all the source code contained in the nodes is GLSL code. Since tesselation shaders are used, the required version of OpenGL is 4.3 for the user to run the application properly.

5.2 Node hierarchy and classes

All of the basic types of nodes implement the generic Node interface, as mentioned in the previous chapter. It is an abstract class that contains all the common attributes that a node of any type needs to have, like their name, unique ID, their node type, the allowed shaders that it can execute etc. Every derived node inherits and alters this information and has to override the virtual compile method that is responsible for the operational logic of the node when it is evaluated during the graph's compile phase. An overview of the class hierarchy for the existing nodes can be seen in the figure below:

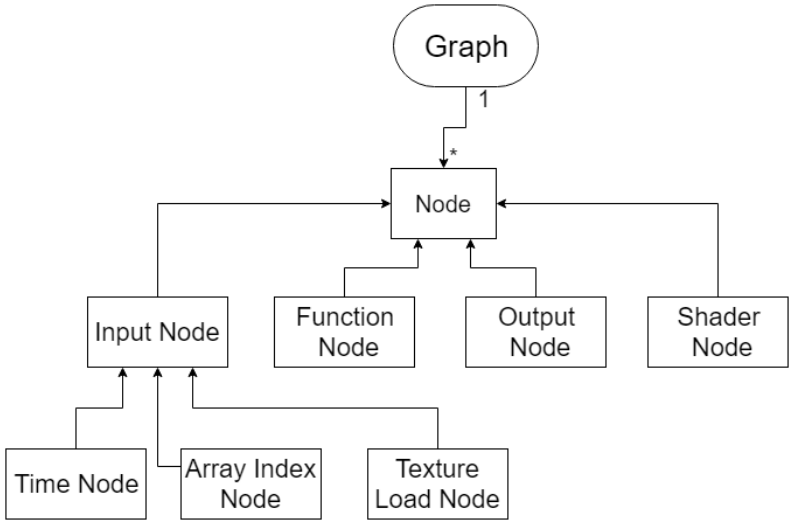


Figure 5.1: Node architecture

This structure attempted to avoid creating an individual class for each and every different node that exists in the system. This would create a lot of unnecessary overhead since most of the basic structure is common for nodes that belong to the same base type (Input, Function, Shadernode). For this reason, the individual functionality is embedded inside these classes and

handled accordingly.

For example, for the category of Input nodes, instead of having different classes deriving from the InputNode class for two different data types (e.g float and vec2) or for the different shader attributes, the system handles all these different cases by using instances of the same class but with the different parameters that determine the behavior of the node. This keeps all the information local and allows for better maintenance since changes will be applied either automatically to all input nodes, or will only affect a section for a specific case. The final parameters for each instance are still determined from the corresponding JSON file and are retrieved from the appropriate map at runtime.

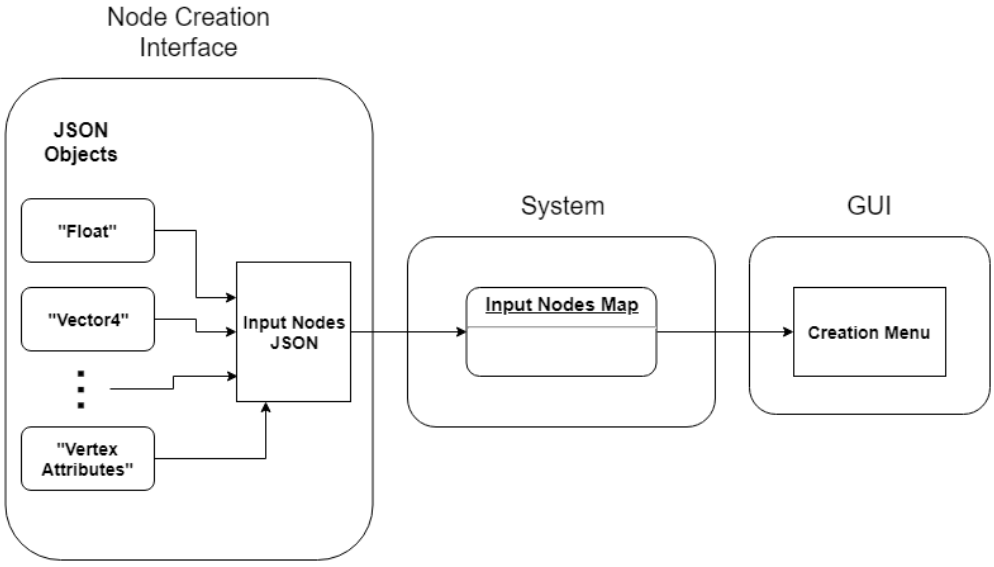


Figure 5.2: Node information retrieval

This method works well when the functionality of the node, as unique as it is for each node, still falls under the same format and overall structure of the corresponding base type. However, in the case of new and explicit operations

that differ from the base type pattern, creating a new derived class cannot be avoided. An example is the LoadTexture node which apart from outputting a texture sampler, similar to a float or a vec4 type, also encapsulates the operation of loading and storing a texture which requires special functions and possible communication with the renderer. Other special cases are the 'Time' node and the 'ArrayIndex' node.

5.3 Compilation process

The following section will describe in depth the compilation process of the graph and the details around the modification process of the shaders.

5.3.1 Graph traversal

As explained in an earlier section, the graph is a DAG since the dataflow is directional from output to input slots. The graph traversal though uses the opposite direction (input to output) to perform a depth first traversal. The node class retains connection information for both inputs and outputs, making this navigation possible. It is intuitive to use this direction since an input node has only one connection, effectively creating a tree structure.

Using the output shadernodes and the master root node as starting points, the traversal logic for a single node is the following :

```
void TraverseNode (Node CurrentNode)
{
  if (CurrentNode.HasCompiled == false)
  {
     for(all input slots in CurrentNode)
     {
          if (slot->ConnectedNode == null )
          {
              continue;
          }
          else
          {
              TraverseNode(slot->ConnectedNode)
          }
     }
     CompileNode(CurrentNode);
  }

 }
```

Listing 5.1: Node traversal logic

The above algorithm ensures that the current executed node will only be compiled once, even if there are multiple paths to it, and that all of the nodes that will provide their results as its inputs will be compiled first. Calling this algorithm from all output nodes will ensure that all nodes that are connected to the graph will be evaluated. Those without any connections will not be part of the process. Traversals are initiated from all shadernodes in order, with the fragment/Master node being the last.

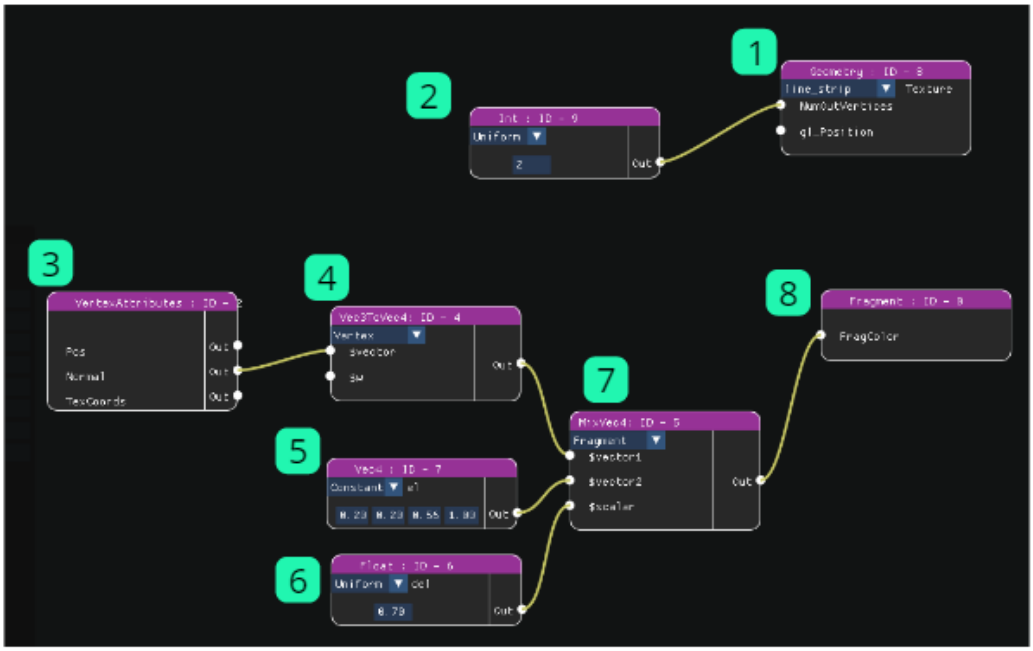


Figure 5.3: Example of the node compilation order

5.3.2 Node Compilation and variable naming

In order to ensure that every variable name in the final program is unique, the system makes use of two maps/dictionaries, one that maps output slot names with their associated variable names and one that maps variable names to their associated slots.

This allows for the quick retrieval of the name that is associated with a slot or the slot associated with a specific name, if those have already been assigned in the program. An output slot's identifier name in the map is unique and its value is assigned by the formula below:

$$OutputSlotName = NodeID + "->" + SlotIndex \tag{5.1}$$

where the NodeID is the unique ID of the node and SlotIndex is the index

of the slot in the array of output slots.

During compilation, a node firstly iterates over its input slots and replaces the predefined variable name related to it (e.g $normal , $color ,$vector) in its source code snippet. This name will be replaced either by the variable name that is received from another node, or the default value of its data type. In the case that an incoming connection exists, the variable name is retrieved from the slot to variable map, by checking the connected output slot's name entry.

After this process is done, the node updates the maps, by creating new entries for its own output slots' names and their associated variable name values. To avoid conflicts in new name declarations, the system will check if the names already exist. If they do, a new unique name will be generated by appending a unique integer identifier at the end of the predefined names (e.g _0, _12). This new name will also be correctly replaced in the source code, just like for the input nodes. We remind the reader that the initial predefined name for a slot is assigned from the node implementation interface and the JSON file.

The above process ensures that all the final variable names created for the shader program are unique. One drawback is that it does not check the names of local variables that are exclusively used for calculations inside a specific source code snippet of a function node. These local variables are not registered in the dictionaries since they are not part of the input/output slot operations and were written by the author of the code. Examples of such variables can be seen in figure 5.4 below:

```
"Code": [
  "vec3 normal = normalize($Normal); ",
  "vec3 lightDir = normalize($lightPos - $FragPos);",
  "float lambertian = max(dot(lightDir,normal), 0.0);",
  "float specular = 0.0;",
  "if(lambertian > 0.0) {",
  "vec3 viewDir = normalize(-$FragPos);",
  " vec3 halfDir = normalize(lightDir + viewDir);",
  "float specAngle = max(dot(halfDir, normal), 0.0);",
  "specular = pow(specAngle, 16.0);",
  "}",
  "vec4 BlinnPhong = $ambientColor + lambertian * $diffuseColor + specular * $specColor;"

]

},
```

Figure 5.4: Examples of local variables in the Blinn Phong node

In order to address this issue, a standard for the source code authoring of the nodes needs to be adopted, that will differentiate between a variable generated by a node and a local one, effectively avoiding the issue.

It is important to note that the modifications in the variable names and the source code is not permanent. The node creation library retains all the initial information and the node evaluation always begins from those default values. The slot to variable maps are also cleared and reset before every graph compilation.

After both input and output slots have been iterated and have modified the source code of the node appropriately, the final step of the node compilation is to append that source code in the appropriate shader and section. This process will be explained in detail in the next chapter.

5.4 Shader segmentation and modification

The idea of modulary constructing a shader by using an abstract syntax tree representation, that eventually compiles into actual shader code was

abandonded early in the process. The design and workload of creating such a system from scratch would be too great for the scope of this thesis, as was realised during the examination of relevant material [**?**].

Instead, shaders are modified through simple text editing. The initial shader program used for rendering, contains a default passthrough shader for each stage. This serves as the basic skeleton code which will be further populated with source code from the nodes. Whenever the system compiles the graph, the shaders always are always initialised to this default code.

Each of the five shader text files is split in six segments :

1. Version Segment
2. Varying Variable Segment
3. Uniform Variable Segment
4. Constant Variable Segment
5. Main Segment
6. MainLoop Segment

.

The start of each of these segments is indicated by a unique special identifier string in comments e.g //$Version$, //$Varyings$ etc. Their purpose is to serve as a shortcut to quickly navigate to that section of the shader. These identifiers need to be part of each shader text file, even if in practice they might not be used. An example of the default vertex shader with the segmentation identifiers can be seen below :

```
 //$Version$
#version 330 core

layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoords;
layout (location = 3) in vec3 aTangents;
layout (location = 4) in vec3 aBitangents;

//$Varyings$

//Uniforms-Standard
layout (std140) uniform Matrices
{
        uniform mat4 View;
        uniform mat4 Projection;

};
uniform mat4 Model;
uniform mat4 MV;
uniform mat4 MVP ;

//$Uniforms$

//$Constants$

void main()
{

        //Main - Default
    gl_Position = (Projection* View* Model* vec4(aPos, 1.0)) ;


        //$Main$


        //$MainLoop$

}
```

Listing 5.2: Default segmented vertex shader

This segmentation creates a clear organized structure that makes the modification process of a shader easier, as well as providing better readability to the final result.

The order in which source code is appended to the shader is important. Nodes that are compiled first need to have their code declarations higher in the shader text, to ensure that their calculations have finished, before another part of the code makes use of them further below. In order to ensure this fact, instead of writing directly to the final shader text, the master root node

of the graph accumulates all the code from the nodes during compilation and controls the writing priority.

For each segment, the master node has a list of strings that represents the code snippets that are meant for that segment. That list is populated in the order that the code is received, effectively achieving our purpose. Finally, the master node assembles the shader text by appending each list in the appropriate segment.

This process occurs every time the graph is to be compiled.

5.5 Varying creation

Communication between shader variables was an important and rather difficult task to face during development. In order to achieve the desired modularity and give the freedom to the user to change the shader stage of the executed code, some sort of subsystem that handles varyings needed to be implemented.

A variable needs to be declared and treated as a varying when the data that is passed to a node is derived from a previous shader stage than its own. Since every node has the current executed shader stage available as exposed information, it is easy to verify this condition.

We have adopted the following rule for naming varyings in the different shader stages.

Shader Stage	Prefix	
	In	Out
Vertex Shader	-	'v'
Tesselation Control Shader	'v'	'tc'
Tesselation Evaluation Shader	'tc'	'te'
Geometry Shader	'te'	'g'
Fragment Shader	'g'	-

Table 5.1: Variable name prefixes for each shader stage

A varying variable is created in the shader program during the evaluation of the output slots. This does not only append the appropriate prefix to the unique output name, but also declares the variable in the current shader stage and the ones following it, with the appropriate naming of course. The system ensures that this declaration only happens once per shader and once per dataflow direction (in/out).

During the input slot evaluation, the only thing that is required is the addition of the appropriate prefix in the incoming variable's name. It should be noted, that the prefixes are not permanently appended to the variable names to allow for flexibility. They are simply added in the name only before modifying shadercode.

5.6 GUI implementation

The visual node editor was created with the external library Dear ImGui created by [9] which uses immediate mode graphics for the creation of modular graphic user interfaces in C++. It is responsible for the visual shader editor panel window, as well as the rendering of the individual nodes and the connection curves between them.

For every node instance that is created in the graph, the system will automatically create a unique visual node associated with it. The visual node is aware of the graph node it is connected to but the information does not flow the other way. Changes in the GUI send information to the backend in order to call the appropriate operations. For example, changing a constant value in an input field will trigger the graph to recompile or creating a visual connection between nodes will result in the creation of a graph node connection.

There is one visual node class that handles the appropriate drawing details for each base node type. All drawing is dynamic in regards to the amount of inputs/outputs that a node has, meaning that its vertical size will adjust accordingly. This however is not the case for the horizontal size which is fixed.

This interface is also responsible for handling special cases for subcategories and types which require additional fields or drawing widgets, like the dropdown menu for the geometry shader node or the transformation matrix input node. These widgets are not present for the rest of the nodes of the same type.

6

Analysis and results

6.1 Target users and usability testing

The system was not designed with a unique set of users in mind. It was targeted both towards programmers and artists, but not exclusively to one of the two. In hindsight, it is safe to observe that since the visual implementation has a low abstraction level for its components, it is more accessible to graphics programmers and people with more technical knowledge around shader programming and its concepts. The custom node creation interface also reinforces this observation, especially regarding the function nodes which require actual GLSL syntax to author.

In order to confirm the above, it would be interesting to evaluate the usability of the tool by asking testers of various skill levels and affiliation with graphics programming to create specific effects with the visual editor. An experiment like that could not only provide useful feedback on the current state of the system, but also inspire new features for future development. Due to time constraints these tests were not conducted, as the focus of the development was targeted towards creating a functional version of the system.

6.2 Limitations

The possible effects that the user can create with the resulting system, are of course limited to the node library that is available, which at the moment is pretty basic. A direct comparison with the capabilities of normal code authoring, only reinfornce this observation. Furthermore, the library's size is misleading, as it is populated with a lot of nodes that perform the same operation but for different data types, due to the absence of the automatic data type conversion system that was discussed in previous chapters.

The application can only one use and edit one shader program at all times. This effectively means that the user cannot implement multipass shader effects or even choose a different rendering target, like for example a texture.

Another drawback is that all of the shader stages are always present in the program, regardless of their usage. This may prohibit the use of specific options because of the restrictions that are introduced by the rest of the shaders, as well as adding some unnecessary overhead to the performance. For example, when tesselation shaders are active in the pipeline the chosen drawing topology from the application must always be in patches.

General usability features that can enhance the user experience are also absent from the current version. Secondary elements like deleting or grouping nodes, resetting and saving/loading existing graphs, creating new scenes and many other would significantly extend the user's capabilities but their development was not a priority.

6.3 Performance

The system runs smoothly at around 58-60 fps for rendering even the highest available detailed 3D models. There seems to be an initial reduced performance after the model loading and until the first graph compilation. After that, the frames stabilize at the acceptable rate. The following models have been used for testing :

Model	Cube	Suzanne Low	Bunny Low	Stormtrooper	Suzanne High	Dragon
Vertex Count	36	2904	14904	19554	125952	300000

Table 6.1: The different 3D models used during development

A major factor that can reduce the performance significantly are the tessellation levels of the model, which currently are not bounded by any value. Should the user choose a high value, the vertex count can increase excessively, making the interface almost non responsive and therefore unusable. It has to be noted that no particular optimizations have been implemented regarding the rendering process since there is only one scene with a unique object that is loaded once.

Another aspect that might influence performance is the frequency of the graph compilation. A single compilation process does not have significant overhead, since the graphs that can be created in the editor will realistically never be too large to prove a computational hazard. However, frequent calls to the process can slow down the system. For this reason, a recompilation of the whole graph is triggered only when a non-uniform value is changed in the visual editor or connections are altered in the graph. This ensures that the number of compilation triggers is the necessary minimum.

An example of severe performance overhead that occurred due to the frequency of compilation calls, was the introduction of the Time node. Since it

exposes a value that is updated constantly, the older version of the system triggered graph compilations on every frame, making the tool unusable. The issue was resolved with the use of uniform variables, that are simply updated for the program every frame without the need to recompile the graph. A node that represents a uniform variable is added in a list which is traversed and evaluated separately.

The program at the current version also mishandles memory, resulting in the application to crush after a certain point in time.

6.4 Intermediate effect preview

A very useful feature that modern tools like Gratin [16] and Shadergraph [] offer is the preview of a shader effect in intermediate stages. This gives the user the opportunity to inspect how specific nodes will affect the calculations up to a certain node, apart from having the preview of the whole graph calculation.

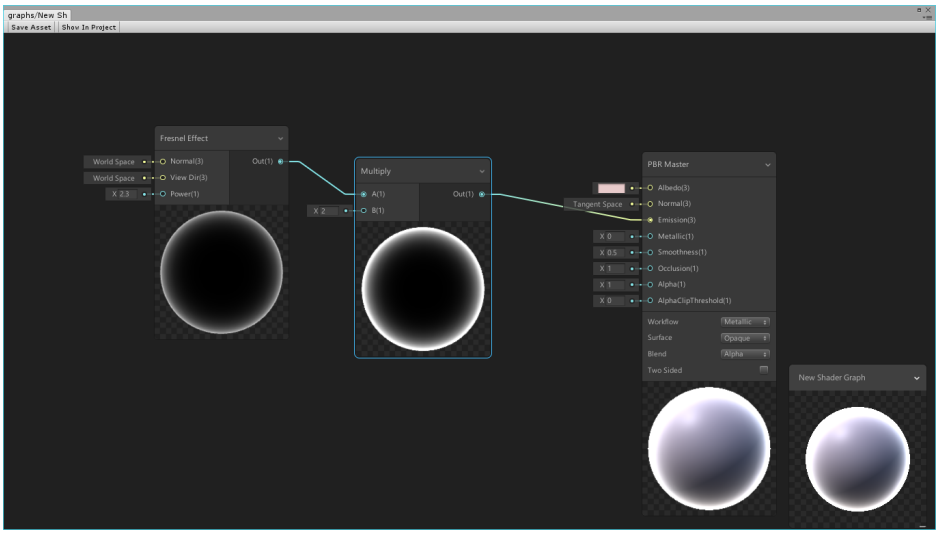


Figure 6.1: Examples of the preview feature in Unity's Shadergraph

Even though this feature is not available in the current version of the tool, it was part of the considerations that took place during the design of the backend. The initial idea was to keep track of the actual values of the inputs and the outputs for every node, in order to quickly display them to the user when requested. The obvious problem with this approach is that the calculations that take place in function nodes do not actually happen during the compilation process. They will only be performed in the GPU by the shaders themeselves, so an output value can never be calculated on the spot and displayed in the application.

A possible way to implement it, is to have different "hidden" shader programs for every node that offers such a preview window, which will accumulate and execute the shader code that has been created by the graph up to that point. This is essentially the same process that is performed for the whole graph by the final master node. However, since different shader stages are exposed as nodes and are an essential part to the traversal of the graph, displaying a specific intermediate effect might not be a trivial task. Calcu-

lations from different branches of the tree might be required for the correct display of the intermediate effect, that could alter the current algorithms of system significantly. Due to these difficulties and seeing that this feature is also of inferior importance, its implementation was left for the future work.

6.5 Loops

Using geometry and tesselation shaders introduced an interesting challenge during development, which is the use of loops and arrays. Both of these shaders receive data in the form of arrays of structs that represent all the vertex information from either a patch or a different type of primitive and their use is crucial to the calculation of the new geometry that will be generated. Additionally, the output geometry is not calculated one element per shader invocation, like in the cases of the vertex and fragment shader. In a typical usage case of the geometry shader for example, every invocation emits at least the same amount of vertices as the ones contained in the primitive that was provided, if not more. Two possible ways can be viewed below:

```
void main() {

  int NumOutVertices = 3;

  gl_Position = gl_in[0].gl_Position;
  EmitVertex();

  gl_Position = gl_in[1].gl_Position;
  EmitVertex();

  gl_Position = gl_in[2].gl_Position;
  EmitVertex();

  EndPrimitive();
}
```

Listing 6.1: Sequential Method

```
void main() {
  int NumOutVertices = 3;
  for (int i=0; i<NumOutVertices ; i++)
  {
    gl_Position = gl_in[i].gl_Position;
    EmitVertex();
  }
  EndPrimitive();
}
```

Listing 6.2: Loop method

The challenge/question is what is the best way to expose this information

to the user, both for accessing the different structs from the input and calculating the emitted vertices. There are a few different options to consider for each case.

Regarding the output calculation from the geometry shader, implementing the sequential method that is depicted on the left in figure xx , results in the most customizable solution. The user can calculate every single emitted vertex individually, providing the same opportunities as authoring the code. The obvious flaw in this method is the decrease in usability due to the great number of input slots of the resulting shadernode (one for each generated vertex). Should that number increase to the double digits, the node becomes less and less usable, especially in the current version of the system where the GUI does not provide any resizing (zoom in/out) or collapsing for the node editor.

Another method, which is the one currently used by the system, is to provide a formula for calculation for each point inside a loop, as shown on the right side the previous example . This has the opposite effect of the previous method, in the sense that it restricts the position calculation to a specific formula for each vertex but offers a much simpler node interface with only one slot to be evaluated. The loop is part of the default code of the geometry shader and it is the reason why the MainLopp segment exists and is used.

In both of the cases above, using data from the input primitive is important to the calculation of the new geometry. Providing access to the input array of structs can be implemented with a similar logic. A sequential method would expose all indices individually (e.g gl_in[0].gl_position,gl_in[1].gl_position) while a more generic method exposes one slot with a variable index (gl_in[i].gl_position). The first method might be a manageable feature for the input of the geometry shader, since the primitive with the maximum amount of data has only

6 vertices but for tesselation shaders the patches may grow dramatically in size. For this reason, the generic method was preferred in the current version.

It is important to note that arrays are not handled by the system as a different slot data type. The variable indexed struct data gl_in[i].gl_position is nothing more than the name of the variable in the output slot, which is of the same data type as the exposed variable (vec4 in this case). This means that connecting this slot to any kind of input will result in errors in its present form, since the index variable cannot be evaluated by the shader code. A direct connection will only work in the case of the geometry shader which, as was mentioned above, will append the code inside a loop. Even in this case, if the number of input-output vertices does not match, the geometry will result in errors.

To address the above issue and have some control over the array, a special ArrayIndex node was introduced, which receives data in the form of an array and replaces the index with a value specified by the user. This of course introduces the problem of requiring multiple of these nodes to effectively index every incoming value, but it offers a starting point to addressing the issue.

The implementation details that were introduced in the previous paragraphs, are but a first attempt in exposing these new shader stages to the visual environment. The complexities involved with the data handling were discovered late in the development process, resulting in the above constrained usage.

An attempt at designing an alternative system for handling loops as nodes, could assist at the implementation of a more generic system that does not require so many predefined components in order to operate. This matter will be discussed briefly in the future work in the next chapter.

6.6 Shaders that have been created with the system

The following screenshots showcase different shader effects that were created with the resulting system:

6.6.1 Blinn Phong lighting

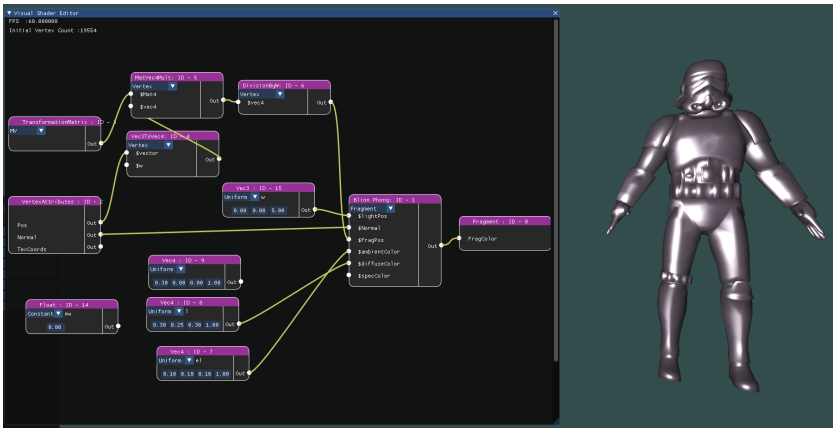
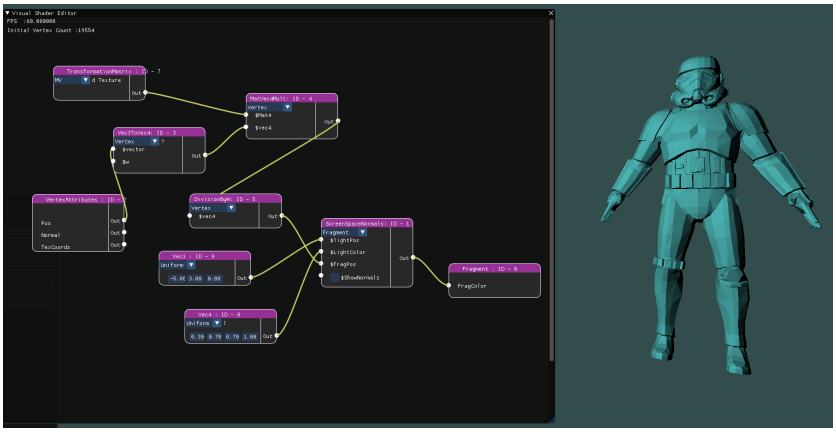
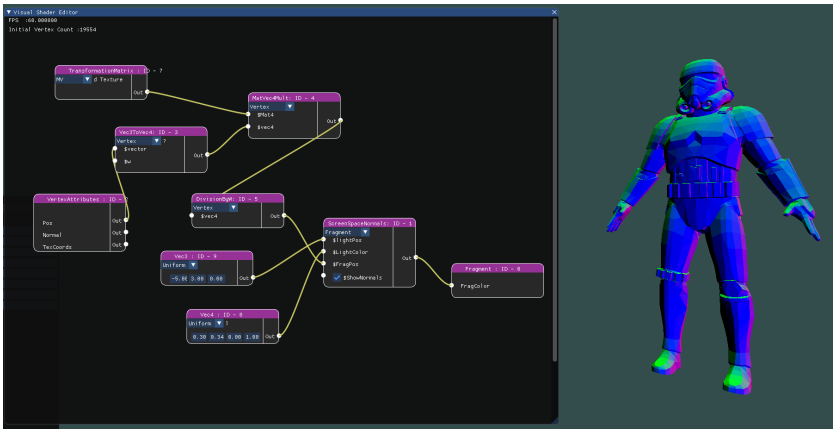


Figure 6.2: Blinn Phong

6.6.2 Screen Space Normals



(a) Standard color



(b) Normal color

Figure 6.3: Screen space normals

6.6.3 MatCap shading

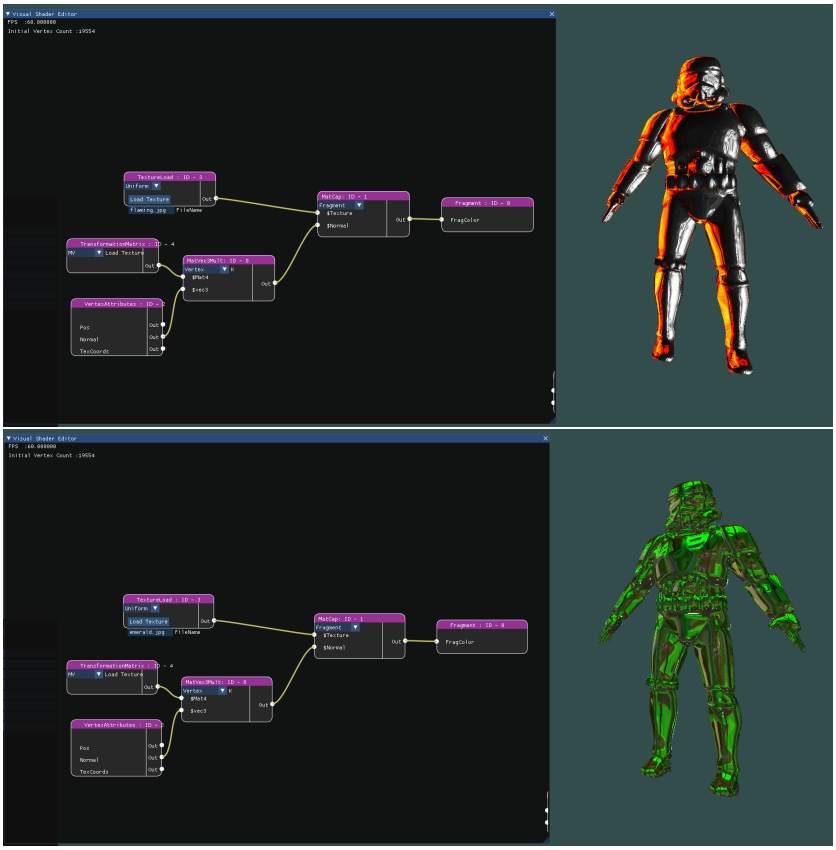


Figure 6.4: Matcap shading for different materials

6.6.4 Texture sampling and color mix

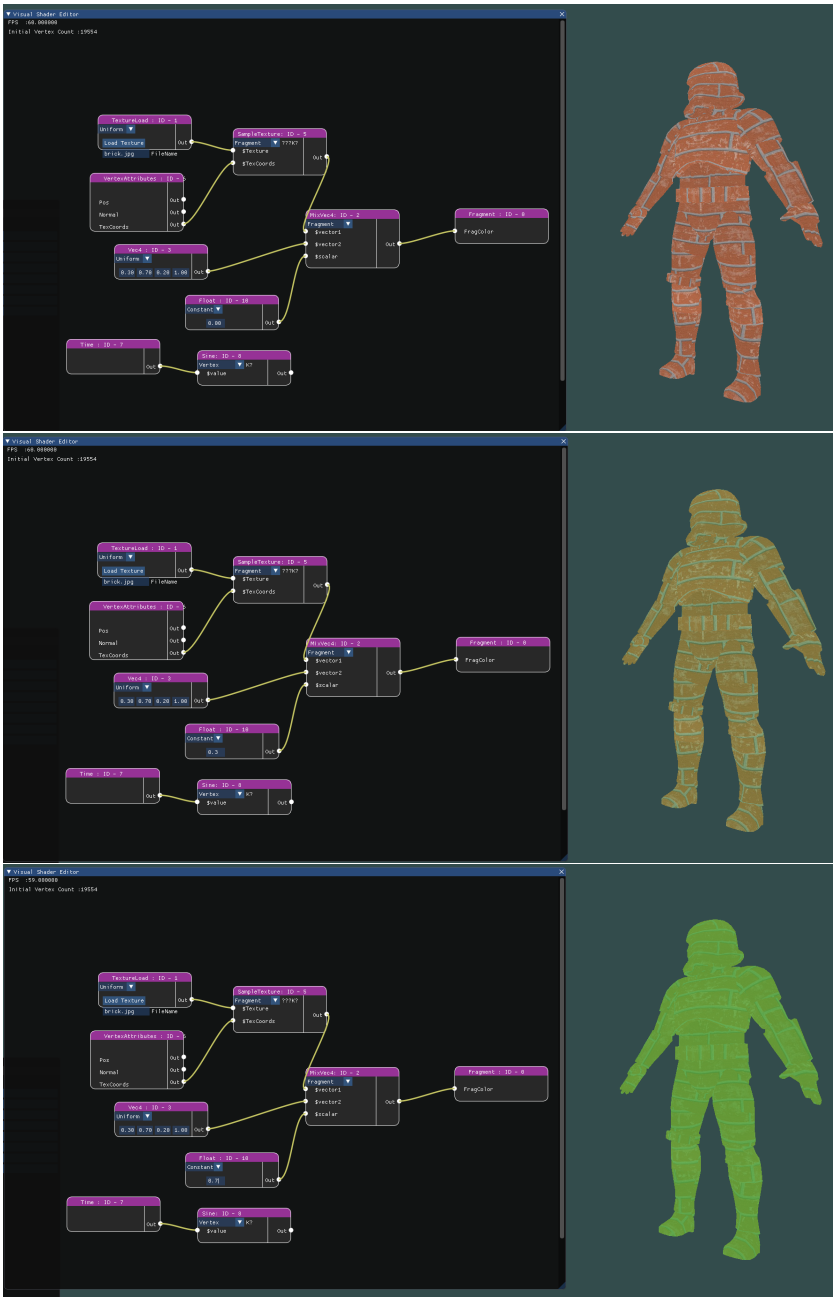


Figure 6.5: Texture-Color mix with different blend values

6.6.5 Explosion shader

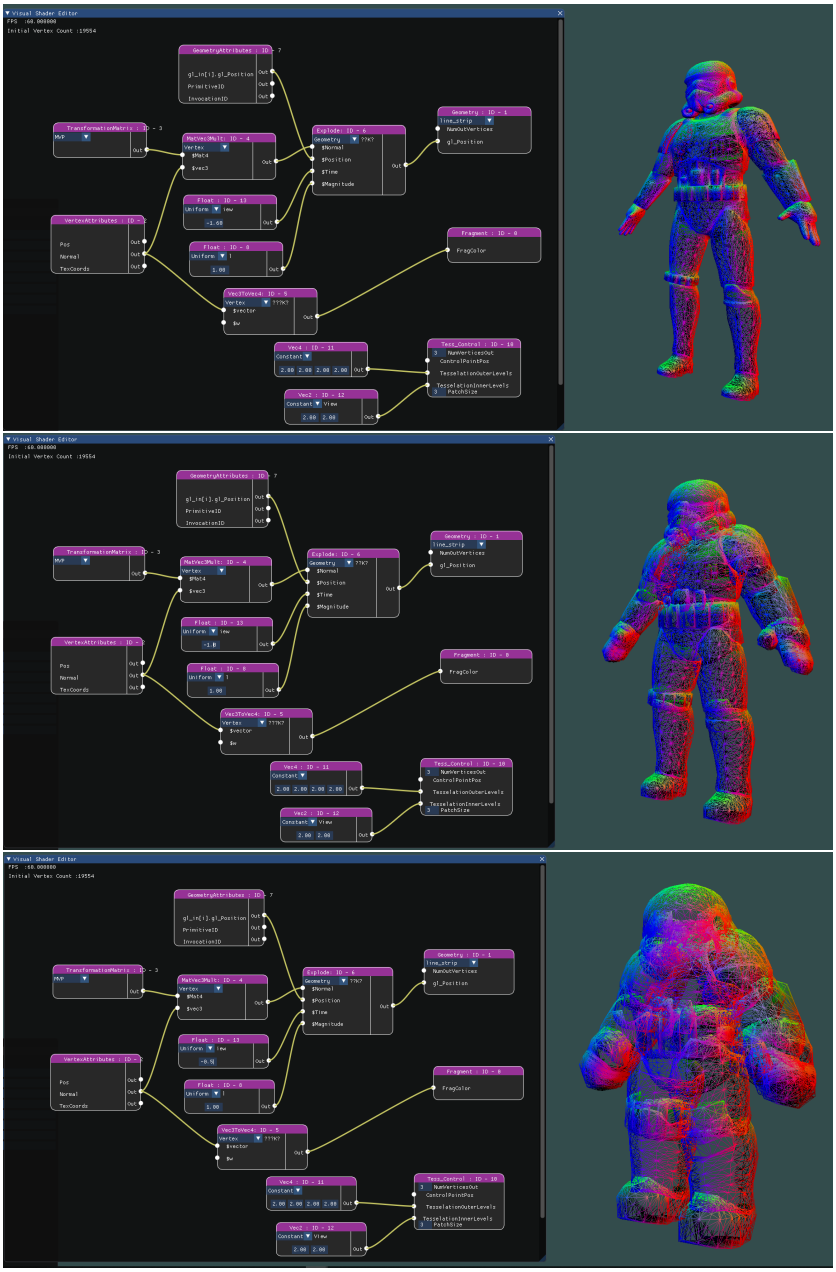


Figure 6.6: Stages of explosion shader with triangle strip topology

Figure 6.7: Stages of explosion shader with point topology

7

Future work and improvements

This chapter will summarize the desirable features and thoughts around matters that are part of the future development of the system. Some of these have already been mentioned in previous chapters. The order of the elements in the following paragraphs, does not represent the priority or importance of the discussed component.

Designing a new system for generating and handling loops is one of the most interesting prospects of the future work. An inspiration for implementing such a task can be found in Unreal Engine [6], where loops are fully programmable nodes that can be connected to other function nodes. This connection is different from the typical data input/output slot interface that has been used so far. Instead, there is a special flow control input slot that manages the execution of the node's function.

A similar technique could be used for our purposes, where shader code could be wrapped in any type of flow control statements. This of course introduces the difficulty of defining what does it mean for a node to be executed multiple times and is that definition the same for all types of function nodes ?

The flow control system could also assist in improving the access and iteration over data arrays, which in the current version is pretty constrained

and only allows access to a singular element.

The introduction of an automatic data type conversion system will remove any unnecessary usage of conversion nodes that currently perplexes the graph. This system might remove some of the direct control that the user has over these type conversions, but simplifies the usage of the tool significantly. Possible issues with this automation could be prevented, by giving the user the opportunity to choose the conversion they want with GUI widgets on every connection, instead of providing a general lookup rule table which is common for all cases.

Usability can also be improved by adding general workflow options to the editor like deleting nodes, zooming in/out for a better overview of the graph, multiple node selection and moving, grouping and collapsing nodes etc.

Generalizing the use of the shader stages is an important step towards making the current structure more modular. Instead of constantly making use of all available shader stages, the user should be able to choose when to activate and include each stage in the pipeline. The intermediate effect preview that was mentioned in the previous chapter could also benefit from this modularity.

Apart from the visual editor, the system could also expand the capabilities of the scene editor. Some of the most typical features that could be added are camera manipulation, the additional loading and manipulation of objects, saving and loading scenes and other general rendering options. This will enable the creation of more complex scenes that contain many different shader effects at the same time.

Choosing a final render target for the results from fragment shaders can lead to the use of multipass effects and also serve as a possible link between different shader pipelines (different graphs in this case).

Finally, overall optimizations in code and bug fixes will ensure that the tool operates at its full potential and the functionality of the tool is the expected one.

8

Conclusion

The purpose of this thesis was to create a node based tool that can perform real-time modification of computer graphics shaders using visual programming.

This is managed through the creation and manipulation of a graph, whose nodes represent shader operations while the connections between them represent dataflow. Apart from the standard vertex and fragment shaders, the user may additionally create effects for geometry shaders, as well as the modern tessellation shaders.

In order to allow for additional flexibility and modularity, apart from the predefined node library that is provided, the system provides a custom node creation interface where the user can define their own node operations.

The design and implementation of the tool, provided some insight to useful techniques for achieving communication between the different shader stages in the pipeline and assembling a shader program in a modular way. Additionally, this report outlines the complications that come along with this modularity and proposed partial solutions for them.

The creative possibilities, as well as the usability of the resulting system are still pretty limited in the current version. The resulting shader code

has not been subjected to any optimizations that could simplify the code placement or increase performance. Furthermore it is exclusively authored in GLSL, which restricts the integration with other applications that make use of other shading languages.

All in all, the implementation of the system serves as a good basis, that can be further built upon to create a consistent application for the easier creation and testing of computer graphics shaders.

Bibliography

[1] Blender's material editor.

[2] Maya material editor.

[3] Shadertoy.

[4] Thekhronosgroup.

[5] Unity shadergraph.

[6] Unreal engine.

[7] Gregory D Abram and Turner Whitted. Building block shaders. In *ACM SIGGRAPH Computer Graphics*, volume 24, pages 283–288. ACM, 1990.

[8] Robert L Cook. Shade trees. *ACM Siggraph Computer Graphics*, 18(3):223–231, 1984.

[9] Omar Cornut. Imgui. *Retrieved on April*, 4:2018, 2014.

[10] Angel Edward and D Shreiner. Interactive computer graphics: A top-down approach using opengl. *Addison-Wesley, ISBN*, pages 978–0, 2003.

[11] Martin Fitger. *Visual shader programming*. Skolan för datavetenskap och kommunikation, Kungliga Tekniska högskolan, 2008.

[12] Frank Goetz, Ralf Borau, and Gitta Domik. An xml-based visual shading language for vertex and fragment shaders. In *Proceedings of the ninth international conference on 3D Web technology*, pages 87–97. ACM, 2004.

[13] Peter Dahl Ejby Jensen, Nicholas Francis, Bent Dalgaard Larsen, and Niels Jørgen Christensen. Interactive shader development. In *Proceedings of the 2007 ACM SIGGRAPH symposium on video games*, pages 89–95. ACM, 2007.

[14] Morgan McGuire, George Stathis, Hanspeter Pfister, and Shriram Krishnamurthi. Abstract shade trees. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 79–86. ACM, 2006.

[15] Morten Nobel-Jørgensen. Kickjs.

[16] Romain Vergne and Pascal Barla. Designing gratin, a gpu-tailored node-based system. *Journal of Computer Graphics Techniques*, 4(4):54–71, 2015.

[17] Joey De Vries. Learnopengl.

Appendices

A

Cool Stuff