

Politecnico di Milano
Facoltà di Ingegneria Industriale e dell'Informazione
MSc COMPUTER SCIENCE AND ENGINEERING

Procedural Shader Generation in Unity3D

Master of Science thesis of:

Elio Sasso

Matricola 928061

Advisor:

Prof. Pierluca Lanzi

Co-advisor:

Prof. Daniele Loiacono

Academic Year 2021/2022

Abstract

In recent years, the demanded quality of 3D visual applications by the public has been growing at a steady rate. While the new tools at the disposal of artists and designers allow them to produce high quality visual experiences, there are still some tasks which are both critical and time consuming when producing a high quality render. One of these tasks is often programming a dedicated shader which can render the produced asset in the best way possible. Currently, programming a new shader requires technical knowledge of shader languages such as High Level Shading Language (HLSL) and OpenGL Shading Language (GLSL), which leads development teams to either look for artists with technical knowledge for shader programming or to hire dedicated individuals under the role of technical artists. The task of shader programming, therefore, is costly and its price often cannot be payed by smaller development teams. Iterating on the shader program can also be a time consuming task given by the fact that, apart from some input values which can be changed, shader programs favour performance over versatility.

In the latest years, some companies tried to bridge the gap between the artists and the technical knowledge needed through the use of visual programming tools that represent shader programs through a graph model. While these tools are certainly a step in the right direction, they still require artists to acquire expertise over the rendering pipeline as well as technical knowledge which would not be required outside of programming shaders. The work presented in this document thus tackles the issue of requiring technical knowledge from artists and designer, by designing a tool in the Unity3D rendering engine. The tool, called Shader Graph AutoGen (SGA), leverages the graph model implemented by Shader Graph, a visual programming tool for shaders implemented into Unity. The graph model is reused as part of the chromosome representation of a Genetic Algorithm (GA) implemented into the developed tool and tailored towards producing working shaders which can have real practical use during production. Previous work done on implementing Genetic Algorithms for producing shader programs has been limited to producing code for just sections of the rendering pipeline or to produce shaders specialized for a single 3D application. This project aims at producing shaders which fully leverage the entire rendere pipeline and can be used more generally.

A system for interfacing with Shader Graph's model is thus implemented together with a set of heuristics that guide the Genetic Algorithm when generating new shading programs. Ultimately, a User Interface (UI) has also been implemented in order to let users with little technical knowledge interact with such tool, by setting values that dictate the Genetic Algorithm's behaviour such as the kind of mutations allowed and the number of specimens generated per each generation. The document is structured like the following:

- Chapter 1 introduces the problem and the reasons why such work may be useful to the industry of Video Games as well as 3D application development.
- Chapter 2 goes over the technical and theoretical preliminary knowledge for the project as well as describe the Shader Graph tool and its graph representation of shaders.
- Chapter 3 describes the current state of the art, looks at what work has been done thus far in order to make shader programming a more visual task and ultimately analyzes the previous work done on implementing GAs for procedural generation of shader programs.
- Chapter 4 shows how the tool has been implemented, the challenges that have been faced and what choices have been made in order to tackle such challenges.
- Chapter 5, finally, describes the tool that has been produced and how the user can generate shaders with it.
- Chapter 6 sums up the presentation of the work done and introduces future work that can expand on what has been done thus far.

Sommario

Negli ultimi anni, la qualità grafica richiesta dal pubblico per le applicazioni grafiche 3D è andata aumentando costantemente. Nonostante ci siano strumenti a disposizione di artisti e designers che permettono di produrre esperienze visive di alta qualità, ci sono ancora parti della produzione di queste esperienze che, oltre ad essere critiche per la qualità finale, richiedono una grande mole di tempo.

Una di queste attività è spesso quella di programmare uno shader che possa produrre un render nella maniera migliore possibile. Al momento, creare un nuovo programma di shading richiede conoscenze tecniche di linguaggi di shading quali High Level Shading Language (HLSL) e OpenGL Shading Language (GLSL). Questa richiesta porta spesso i team di sviluppo a cercare artisti che abbiano anche competenze tecniche oppure ad assumere figure dedicate a questa attività, chiamati technical artists.

Programmare shaders, quindi, è un processo costoso il cui prezzo spesso non può essere pagato da team di sviluppo più piccoli. Questo è anche esacerbato dal fatto che iterare su un programma di shading è spesso un processo altrettanto lungo in quanto il codice di shading è specifico per essere più performante che versatile.

Negli ultimi anni, alcune compagnie hanno provato a ridurre il divario che c'è tra gli artisti e le conoscenze tecniche richieste, implementando strumenti di programmazione visuale che rappresentano i programmi di shading tramite un modello a grafo. Questi strumenti sono certamente un passo nella giusta direzione ma richiedono comunque che gli artisti acquisiscano competenza su come funziona la rendering pipeline di un motore grafico e conoscenze sul funzionamento interno di uno shader, che non verrebbero richieste al di fuori di questa attività.

Di seguito, dunque, viene presentato del lavoro che è stato fatto per affrontare il problema del requisito di conoscenza tecnica da parte di designers e artisti. E' stato dunque creato uno strumento in Unity3D che possa aiutare a tal proposito. Questo tool, chiamato Shader Graph AutoGen (SGA), sfrutta il modello a grafo implementato da Shader Graph, uno strumento di programmazione visuale per shaders implementato per Unity. Il modello a grafo viene dunque riutilizzato per la rappresentazione dello shader all'interno di un algoritmo genetico, implementato all'interno dello strumento prodotto e specifico per produrre shader funzionanti

con una applicazione pratica.

Lavori precedenti sull'applicazione di algoritmi genetici alla creazione di shader si sono limitati a parti dell'intera render pipeline e non hanno reso possibile la modifica dei programmi prodotti se non tramite linguaggio di programmazione. Questo progetto si pone come obiettivo di produrre shader che sfruttino la render pipeline per intero e che siano rappresentabili tramite lo Shader Graph di Unity per migliorare l'accessibilità. Al fine di produrre questo strumento in Unity, quindi, un sistema per interfacciarsi tramite C# con il modello a grafo in Shader Graph è stato implementato con un insieme di euristiche che guidano l'algoritmo genetico nella generazione di nuovi programmi di shading.

Infine, un'interfaccia grafica è stata implementata al fine di permettere agli utenti con limitata conoscenza tecnica di interagire con questo tool, impostando dei valori che dettano alcune delle impostazioni dell'algoritmo genetico tra cui il tipo di mutazioni permesse ed il numero di esemplari generati ad ogni nuova generazione. Il documento è strutturato come segue:

- Il Capitolo 1 introduce il problema e le ragioni per cui il lavoro presentato può essere utile all'industria dei videogiochi e della produzione di applicazioni 3D.
- Nel Capitolo 2 vengono spiegate le conoscenze tecniche e teoriche preliminari al lavoro che è stato prodotto, viene inoltre descritto il funzionamento di Shader Graph ed il modello a grafo.
- Il Capitolo 3 descrive lo stato dell'arte corrente ed osserva il lavoro che è stato fatto finora per rendere il processo di programmazione di shader un'attività più incentrata sul visivo in modo tale da essere preferibile ad artisti e designers. Viene inoltre analizzato il lavoro già fatto sulla generazione procedurale di shader tramite utilizzo di algoritmi genetici.
- Il Capitolo 4 mostra come lo strumento è stato implementato, le sfide affrontate e le scelte operate in termini di design al quale queste sfide hanno portato.
- Il Capitolo 5, invece, descrive lo strumento in sé e la sua interfaccia grafica, spiegando come un utente possa generarci shaders.
- Il Capitolo 6, infine, riassume la presentazione del lavoro fatto ed illustra ciò che può essere fatto per estenderlo.

Contents

List of Figures	viii
1 Introduction and Motivations	1
1.1 Motivations	1
1.2 Thesis	2
2 Background	5
2.1 Graphic Processing Units	5
2.2 Shaders	6
2.2.1 The HLSL Rendering Pipeline	6
2.3 Unity Engine	11
2.4 Unity's Shader Graph	13
2.4.1 Shader Graph's User Interface	14
2.4.2 Affected Shader Values	15
2.4.3 Shader Code Generation	18
2.5 Genetic Algorithms	19
3 State of the Art	23
3.1 Graphical Editors	23
3.1.1 Introduction	23
3.1.2 Online Editors	24
3.1.3 Visual Editors	26
3.2 ShaderBase in Processing	28
3.3 Genetic Algorithms applied to Shaders	29
3.3.1 Procedural Generation of Vertex Displacement Shaders	29
3.3.2 Evolving Pixel Shaders in a Video-Game	30

4	Implementation	33
4.1	Introduction	33
4.2	Interfacing with Shader Graph	34
4.2.1	Code Analysis	34
4.2.2	Interface Classes	36
4.3	Genetic Algorithm	37
4.3.1	Overview	37
4.3.2	Mutating Functions	40
4.3.3	Defining the Search Space	42
4.3.4	Limiting Graph Types	43
4.3.5	Guiding the Genetic Algorithm	44
4.3.6	Genetic Representation	47
5	Results	51
5.1	Tool Overview	51
5.1.1	Managing runs of the algorithm	53
6	Conclusion	57
6.1	Future Work	58
	Bibliography	65

List of Figures

2.1	Direct3D's Graphics Pipeline, in stages.	8
2.2	Example of grass being rendered through the use of a geometry shader.	10
2.3	The same 3D scene, rendered using different pipelines in Unity.	13
2.4	The Node Settings Tab, when a node is being selected.	15
2.5	The Graph Settings Tab.	16
2.6	Graph representation of a section of shader code written manually. Demonstrating visual bloating.	19
3.1	Example of GLSL Sandbox's UI, from [7]	25
3.2	Example of VertexShaderArt's UI, from [6]	26
3.3	Example of ShaderToy's UI, from [4]	26
3.4	Visualization of some results from [16]	30
3.5	Standard rendered view in [11]	32
3.6	Different views with procedurally generated shaders, from [11]	32
4.1	Example of a graph flowing into multiple output nodes (Base Color and Normal)	38
4.2	Left : outputs for a shader using the Lit preset. Right: outputs for a shader using the Unlit preset	44
4.3	Procedural Normals through gradient computation. A behaviour difficult to reproduce procedurally.	46
4.4	Examples of single point crossover done at the graph's output. In this case the red highlighted output nodes are taken from first parent, while the green highlighted ones from the second parent.	48
5.1	Some results from running the tool.	51

5.2	SGA can be launched by selecting the Tools tab into Unity. .	52
5.3	SGA's launch tab, with all the settings that can be changed by the user.	54
5.4	The preview will give the user the option to choose each spec- imen as parent of the next offspring	55
5.5	Preview stages available for the user.	56

In the following chapter there will be a description of the reasons why such a project may prove useful to developers and artists in the Video-Game industry as a bird's eye view of the work done and the underlying technology which made this work possible.

1.1 Motivations

The rate at which new Video-Game titles as well as 3D applications need to be produced has been growing faster every year. Along with the time constraints of production, the complexity as well as the detail put into the visual experience offered by new products has also been growing at unprecedented rates, due to the technological advances we have seen. The audience for new Video-Game titles has been at an all time high and has been exponentially growing. All these factors together created an environment where, each year, development teams need to meet higher expectations while keeping tighter deadlines. Development teams thus rapidly expanded in size in order to keep up with the demand. The most rapid expansion was seen in the art department, responsible for producing the 3D assets and visuals for the end product. In recent years new pieces of software as well as new optimizations for the underlying technology allowed to fine tune the visual aspect of virtual assets and produce high-fidelity visuals that can be rendered in real time. In order to fully leverage the underlying technology and create high-quality assets without loss of performance, Shader programs have been widely used as a support for the artistic development of 3D applications. The intro-

duction of specialized programming languages such as OpenGL Shading Language (GLSL) and High Level Shading Language (HLSL) allowed fine tuning of the inner workings of the rendering process for each element rendered into three dimensional and two dimensional scenes. All shading languages offered an approach similar to languages like C and C++, which introduced a degree of complexity that would not allow just any 3D artist to produce a shader that would fit their need. In order to fully leverage shader programs capabilities, highly vertical figures known as Shader Artists, responsible for producing shaders needed by artist, have been created. There have been attempts to bridge the gap between artists and production of shaders by making visual programming tools such as Unity's Shader Graph and Unreal Engine's Material Editor. These tools reduce the complexity by managing the most redundant operations and abstracting the programming process but still require the user to have knowledge of the underlying technology and rendering pipeline.

1.2 Thesis

The project presented in the following document aims to provide users with a tool which procedurally generates shaders and allows the user to produce fully functioning shaders without the need of a deep knowledge of the underlying technology, only requiring a brief understanding of how Shader Graph works. The tool has been written in C# and is implemented for the Unity3D Game Engine. The tool leverages Unity's Shader Graph package which allows users to produce shaders through the use of visual, node-based scripting. By taking advantage of the Shader Graph package, the Shader produced by the tool can also be changed further through visual scripting, where the user will be able to modify a node based representation of the produced shader. At the base of the tool, a Genetic Algorithm (GA) will generate new shaders and apply changes to the graph-based representation of the shaders that are selected by the user, modifying nodes in the graph as well as the value of the inputs. The Shader Graph produces shader programs which are supported by the Universal Render Pipeline (URP), one of the three rendering pipelines that have been implemented in the Unity Rendering

Engine[25]. The advanced features of the pipeline allow to have a vast array of visual effects which can all be leveraged by Shader Graph. As the Shader Graph package is internal to Unity, in order to leverage the graph-based representation, an interface has been built which allows the use of the package from external sources through C# programming. While the source code has been slightly modified in order to accommodate these changes, none of the changes disrupt Shader Graph's standalone operation. Ultimately, the contributions presented in this work are:

- A Genetic Algorithm (GA) specific for generating working shaders in Unity, which leverages Shader Graph's node-based model.
- A set of heuristics and templates that are used as references for generating production-ready shaders that fit the user's needs.
- A Unity tool which procedurally generates shaders as well as their graph-based representations, enabling users with little technical knowledge to produce shaders that fit their needs.

While the project being presented aims to be a new addition to the array of tools that aid artists in the production of high quality 3D assets, it is still crucial to present the amount of work that has been done that has made this project possible. It would not be possible to present this tool without the existence of Shader Graph as well as the Universal Render Pipeline implemented in the Unity Engine. The following chapter will build from the bottom up, starting from a brief description of GPUs and the Graphic Pipeline used by Unity. It will then move on to the Shader Graph package and the moving parts that make up the whole piece of software. The chapter will also offer an introduction to genetic algorithms, which have been used in this work as a way to produce variations for each type of shader offered.

2.1 Graphic Processing Units

The constant advances in the production of semi-conductors allowed, ever since the 1960's, to increase data storage capacity, complexity in Central Processing Units (CPUs) and overall increase the throughput of computers. While this growth has been beneficial to all fields related to Computer Science, the Video-Game Industry sought after specialized boards that would offer rendering capabilities while also cutting production costs. The first dedicated systems capable of rendering three-dimensional scenes were seen in dedicated game consoles in the early 90's. Together with the first 3D geometry processors, dedicated professional Application Programming Interface (API), which allowed professionals

to leverage the graphics accelerators capabilities, began surfacing on the market and allowed for a first level of abstraction when it came to programming. In the late 90's, both OpenGL and Microsoft's Direct3D offered lighting computation capabilities which eventually evolved into shader units that offered greater flexibility.

2.2 Shaders

Shaders were first implemented as a way to add flexibility, by allowing developers to program the rendering pipeline of the underlying GPU[2]. They allow the definition of custom programs which describe during rendering the transformation, clipping and lighting procedure applied to meshes: data collections that describe the shape and the characteristics of virtual 3D objects. The possibility of programming the GPU's rendering pipeline and being able to repurpose the GPU as more than a dedicated rendering system led the programmable pipeline model to supersede the fixed-function pipeline, which allowed for much more limited control over the rendering process but benefited from higher optimization and performance. The programmable rendering pipeline is split into stages, each one managing a different section of the rendering process. Both DirectX and OpenGL define their own custom pipeline but the differences between the two are minor, as both libraries are made to support GPU architectures and aim to achieve the same results. Due to its relevance to the work being presented, the following section will describe Direct3D 11's rendering pipeline specification, since it is the one used by Unity when defining custom shader programs[23].

2.2.1 The HLSL Rendering Pipeline

After the introduction of shaders and shading stages, the configuration of the graphics pipeline has seen little changes. As such, the DirectX specification of the rendering pipeline has been kept fairly the same in recent years.

All the stages that can be seen in Figure 2.5 can be programmed through the Direct3D API but HLSL allows access to only the stages which are represented in rounded boxes:

- Vertex Shader
- Geometry Shader
- Fragment Shader

There will be a brief introduction to each stage but more focus will be put on the highlighted stages, as those ones can be directly modified through Unity's Shader Graph and thus are of particular interest for this document[21].

- **Input-Assembler Stage**

The Input-Assembler (IA) Stage is responsible for gathering the primitive data coming from the memory buffer and assemble it in a format that is readable by the Vertex Shader. It will output primitive graphic types (such as triangle strips and line lists) and also compile adjacency data for each vertex, this piece of data will then be used by the Geometry Shader.

- **Vertex Shader**

The Vertex Shader is responsible for many of the initial operations that might be needed in common shader programs. It can be configured through HLSL and will perform all per-vertex operations. The vertex shader is always responsible for transforming vertex information, applying matrix operations in order to transform the vertex position from Local Space (reference system relative to the object itself, as defined inside the mesh data) to World Space (reference system global to all objects, which can be moved inside the scene). The vertex shader will always at least output all vertex information, but the output data can be extended to include color information and normal vector information[5][14].

The use for this stage, though, goes beyond the simple mathematical transformations. One of the most common usages for the vertex shader is skinning and per-vertex lighting. In the case of per-vertex lighting, the lighting information will be computed per-vertex and then interpolated for all points in between multiple vertices. Apart from skinning for complex animations, the vertex shader

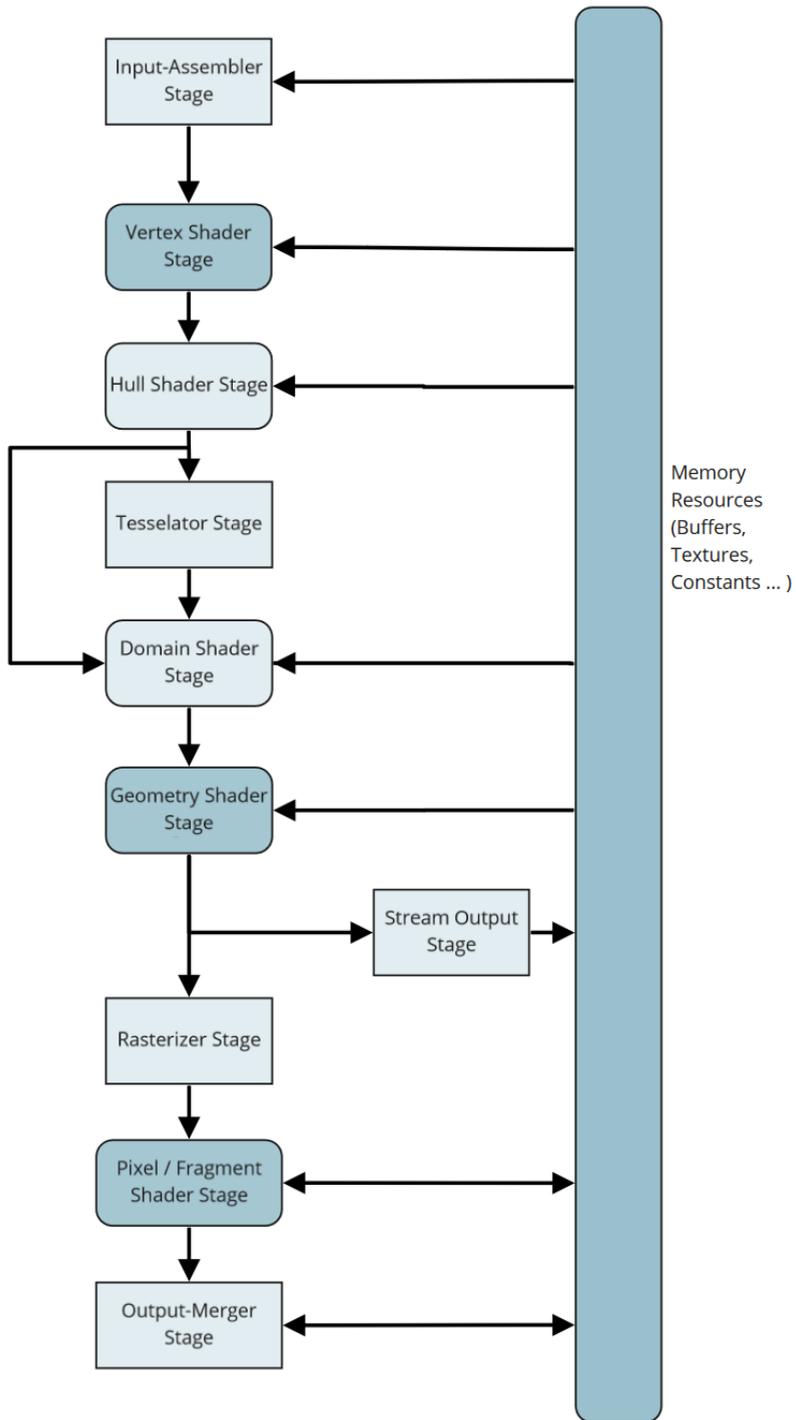


Figure 2.1: Direct3D's Graphics Pipeline, in stages.

can also be used to apply basic animations to a model. Through programmed computations, as a matter of fact, it is possible to replicate different visual effects through the sole use of the vertex shader, therefore easing up on the work of animators and reducing the memory needed for an asset. The vertex shader does not process adjacency information and therefore cannot render inter-vertex effects such as edge detection.

- **Hull Shader**

The Hull Shader is responsible for subdividing the low-order surfaces computed by the vertex shader. This subdivision operated at hardware-level allows for the restoration of high-level detail on low-order surfaces. A common practice in Video-Game development is applying pre-computed, high detail maps to low-order surfaces. The hull shader makes this possible by subdividing the surface and allowing high-detail back onto the surface in the tessellation stage.

- **Tessellator Stage**

The tessellator stage is a fixed function stage and uses the input coming from the Hull Shader in order to generate UV coordinates for each subdivided surface. UV coordinates are necessary for mapping 2D, high detail maps onto 3D surfaces.

- **Domain-Shader Stage**

The Domain-Shader takes in data from both the Tessellator Stage and the Hull Shader, in order to map each surface vertex onto the high-order surfaces produced by the Hull Shader. This allows the computation of the interpolated points between two vertices when applying maps to the high-order surface.

- **Geometry Shader**

The Geometry Shader is a fully configurable shader which is responsible for parsing information about full primitives (such as triangles and lines) as well as the primitives adjacent to the one being processed. The Geometry Shader is capable of generating new vertices based on the information in input and thus all generated vertices will be fed back into the vertex shader in order to be fully

processed by the pipeline. Due to the capability of generating new vertices, the geometry shader is often used for generating foliage or hair in 3D assets. Both visual effects are generated through the use of guiding information added in the mesh data, which generate all intermediate visuals through interpolation.

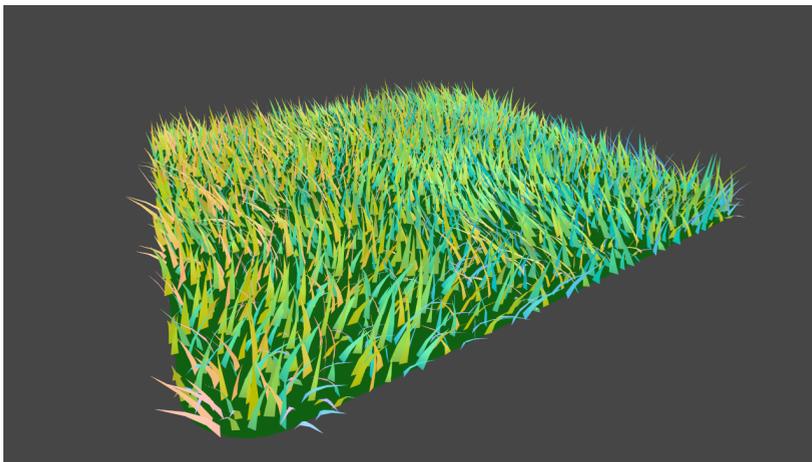


Figure 2.2: Example of grass being rendered through the use of a geometry shader.

- **Stream-Output Stage**

The Stream-Output Stage is responsible for writing new vertex data generated by the geometry shader into a memory buffer.

This allows information to loop back to the vertex shader in case of vertex generation. This also one of the stages with fixed function which cannot be programmed through HLSL.

- **Rasterizer Stage**

The Rasterizer Stage is responsible for mapping the information about vertices and primitives to the image that is shown on screen, composed by pixels. Each primitive is used to generate pixel information through interpolation of the per-vertex values.

This stage is also responsible for clipping the vertices that are out of the view frustum, unloading vertices that are not seen by the viewport.

- **Pixel/Fragment Shader**

The Fragment Shader is the last programmable shader in the pipeline and is also the most used when it comes to shader applications in rendering. The Shader takes, as input, all data about each pixel and runs a customizable program on a per-pixel basis. This shader is the most versatile as it has access to almost all information regarding the rendered scene. This allows developers to compute per-pixel lighting as well as post processing effects. The most common usage for the pixel shader is to compute per-pixel colors by sampling the textures of a 3D asset. Fragment shaders are also used often to generate so called procedural materials, material which have no need for textures and are generated entirely at shader level through the use of noise functions and careful shader programming.

In the case of Physically Based Rendering (PBR), values are computed at fragment shader level, due to the higher precision that the Fragment Shader can deliver compared to the vertex shader. The trade-off for using the Fragment Shader is performance, as running this shader becomes more expensive proportionally to the resolution of the viewport. This is also the reason why modern 3D applications show great performance changes depending on the rendering resolution chosen.

2.3 Unity Engine

In order to produce a 2D image out of information about a 3D environment, computers employ complex transformation computations as well as algorithms determining the final color of an object depending on light, position and other factors, the entire process is called rendering.

There is a vast array of choices when it comes to which rendering algorithm to use, as each different algorithm is specialized in optimizing a different aspect of rendering the 2D image of a three-dimensional scene. In order to allow interaction with a three-dimensional scene, there has been the need for algorithms specialized in computing the light transports and transforms of the 3D scene several times per second, in order to show each changes in the interactive environment without any hic-

cups, as if the user is watching a video.

In the last few decades, the increasing complexity matched by the increase in demand for graphic fidelity led to software house seeking to produce tools and software that would speed up the production of 3D games. Softwares solely responsible for rendering the scene are called rendering engines, as they often have as main focus the one of rendering a 3D scene. Engines that specialized in in real-time rendering also offered extra features such as a way for scripting behaviours associated to the scene and customizing part of the rendering process, as they often were used as a base for making Video Games.

Unity3D, has been one of the first game engines, produced in the early 2000's. It is now an open-source software which implements real-time rendering and allows users to produce interactive applications such as video-games and virtual experiences[26]. In order to offer these features, Unity offers the possibility of modifying a 3D scene by placing assets, apply post processing effects to the rendered image and program full fledged 3D application by allowing users to program behaviours that can change the 3D scene. This is allowed by leveraging C# as a scripting language, while keeping C++ as the language used for programming the core functionalities of the software, thus maintaining higher performance[24]. The recent advancements in rendering technology allowed for the production of very high-quality scenes, comparable to the quality of pre-rendered scenes used in cinema and architectural visualizations. Because of this, game engines have begun to be used in fields other than interactive applications such as cinema, architecture and design[26].

Thanks to the software being open source, most of Unity's internal workings can be changed, meaning the software can be tailored to the needs of the user, offering a great degree of versatility. Unity also offers a vast choice of expansions and extra modules that increase the capabilities of the software as well as add quality of life improvements. Through the introduction of the Scriptable Render Pipeline, users have direct control over the rendering workflow that Unity uses through C# scripting[19]. While the Render Pipeline is fully customizable, Unity offers two rendering pipeline presets which are meant to tailor different needs. The Universal Render Pipeline (URP) is meant for quick optimization of graphics across multiple platforms, while the High-Definition Render



Figure 2.3: The same 3D scene, rendered using different pipelines in Unity.

Pipeline (HDRP) is meant for high-quality graphics, as seen in products such as top quality video-games, previews of products in the automotive industry and architectural previews.

Both rendering pipelines allow further customization by allowing the implementation and usage of custom shaders. While Unity allows shaders to be written in both OpenGL Shading Language (GLSL) and High Level Shading Language (HLSL), the software is capable of translating all shaders written in HLSL into an optimized form written in GLSL [23].

This is done in order to ensure compatibility across multiple platforms, which is one of Unity Engine's main goals.

2.4 Unity's Shader Graph

Among the many extensions and extra modules, which enhance the original software and make it easier for users to change most part of Unity's workflow, the software offers a visual tool for producing shaders. During production, in order to achieve the result desired by the artist, there is often the need to produce dedicated shaders for different objects present in the scene. Shaders which can be used by Unity during rendering have to be written in HLSL, a C-like language used to describe shader behaviour. While HLSL is surely an higher-level language for producing shaders, when compared to GLSL, it still features a complex syntax which operates at a lower level when compared to C#, which is used for programming scripts that dictate the behaviour of the 3D application. The higher degree of complexity does not allow any regular user to approach the creation of dedicated shaders, which limits the capabilities of small development teams. On the other hand, bigger teams dedicate

multiple people, with vertical expertise, to the sole purpose of producing the custom shaders needed, increasing the cost of production due to the higher specialization. The need for expertise causes artists to be left out of the shader production process, limiting the way their creative vision can come to fruition.

In order to bridge the gap between creative vision and technical expertise, there have been multiple pieces of software that generate shaders through the use of Graphic User Interface (GUI) that show users a high-level representation of the shader produced and with which users can interact without the highly technical knowledge required for writing full fledged shaders. Most of these solutions are based on visual scripting, allowing users to build a program through the creation of a graph. Unity's solution for bridging the gap between artists and shaders is Shadergraph. Shadergraph is a visual approach at the production of shaders, offering an alternative to programming them using HLSL. A GUI offers the user a graph-based representation of the shader they are programming. Each node represents one or more lines of code that will make up the final produced program run by the engine. These nodes can have one or more inputs and outputs. Linking two nodes together will make it so the output of one will be used as input in the other and so on. This approach has shown to be successful as other software also implemented the same approach for visual scripting shaders (for example, Unreal Engine also offers Material Editor, a feature similar to Shader Graph, part of their latest software version).

2.4.1 Shader Graph's User Interface

Shader Graph's User Interface (UI) allows the user not only to edit the visual representation of the shader, but to also change more technical parameters as well as add input values to the shader itself. The Graph Inspector panel shows a detailed list of settings for each node present in the graph representation. Through these settings it is possible to change the precision of the float values implemented by the final shader, thus optimizing performance.

A second tab in the Graph Inspector, the Graph Settings tab, shows settings that can modify the shader's mode of operation. In order to

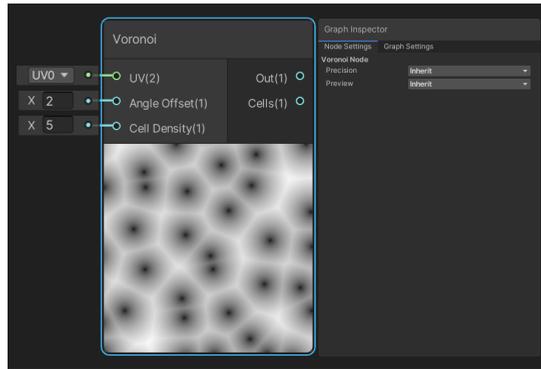


Figure 2.4: The Node Settings Tab, when a node is being selected.

offer the highest degree of flexibility, shaders can have multiple optional parameters such as an alpha value which defines transparency for the resulting rendered object and the way the shader is going to compute lighting, in order to offer the choice between Physically Based Rendering (PBR) and rendering based on the Phong Lighting Model. The Graph Settings tab allows to enable such values as well as define the default precision for all the nodes in the graph, it is also used for selecting the Render Pipeline targeted by the shader (URP or HDRP).

The Blackboard tab manages shader inputs. Shader inputs can be accessed from an external source and changed at run time and the user can add multiple input values to the shader program. Inputs allow changes to the render at run time without modifying more expensive parts of the render, such as textures and maps. Finally, the graph view shows the entire graph being built by the user. All nodes must output to the Master Stack, a collection of nodes representing fundamental values used during lighting computation.

2.4.2 Affected Shader Values

There are many parts that must come together when it comes to creating a fully functioning shader. Apart from the core of the shader's function, many more aspects need to be curated such as the number of passes (times the shader is run to draw different parts of the final image) as

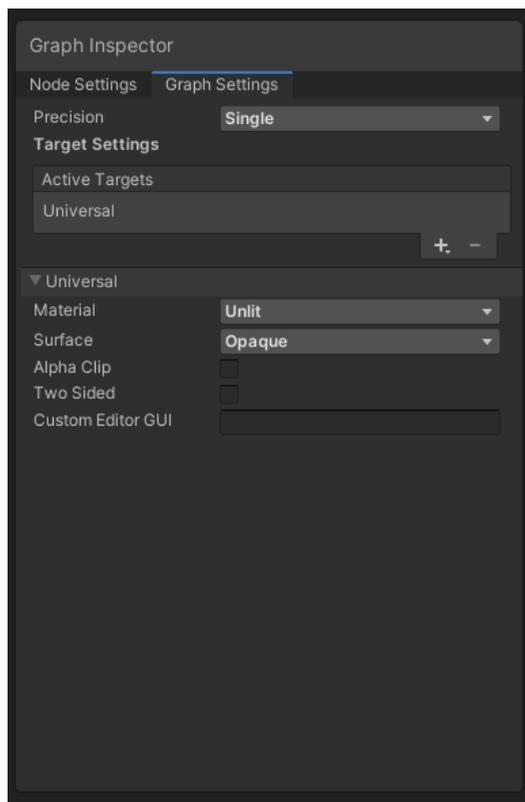


Figure 2.5: The Graph Settings Tab.

well as settings that range from Back-Face Culling to management of the Z-Buffer, which is used in order to select what is to be rendered.

All shaders generated by Unity's Shader Graph are composed by sections of pre programmed shader code. These sections are used in order to allow the highest possible degree of flexibility for generated shader without encumbering the users with technical parameters that contribute little to the end product. On the other hand, though, the presence of pre-programmed sections which cannot be removed (especially extra passes) can have a performance overhead when those sections are not being used by the final program.

In order to slightly reduce the performance overhead, the extra settings in the Graph Inspector tab as well as the possibility of choosing between a lit shader and an unlit shader allow the final code to ease up on the

most expensive part of any shader: lighting. Thus, Shader Graph's interactive UI allows users to change only those sections of the shader that are responsible for the following values.

- **Vertex Position** : changing this parameter allows to change the 3D coordinates of any vertex present in the mesh. This is the parameter often used to bake animation inside the shader itself and it is used by animation algorithms to move the mesh in the desired way.
- **Normal (at Vertex Stage)** : the normal vector is mainly responsible for computing lighting data such as reflection and shadows at run time. The standard normal vector defined by the mesh file can be modified on a per-vertex basis during vertex stage. During rendering, normal vectors are then interpolated for each point in the midst of multiple vertices.
- **Tangent** : the tangent vector is directly tied to the normal vector and thus it is usually changed along with it in order to maintain consistency when defining vectors in tangent space during the Fragment Stage.
- **Base Color (Albedo)** : The base color parameter dictates either the final color of the fragment (in the case of an Unlit Shader) or the color before operating lighting computations. Thus, it is the parameter which is changed in most of the use cases, as it is often used coupled with texture maps which gets sampled on a per-pixel level.
- **Normal (at Fragment Stage)** : while the normal vector defined at vertex stage is computed on a per-vertex basis, it is possible to recompute the normal vector at fragment-stage, using interpolated per-pixel values. This is often the case when shadow detail is reintroduced into the final image through the use of normal map sampling[3].
- **Smoothness and Metallic** : these two values are parameters used when applying PBR to a mesh. This is often the case for assets that need to emulate real-life objects and which have to present

a high degree of detail. The two values are input values for a more complex shading algorithm compared to the simpler Phong shading model.

- Ambient Occlusion : Computing shadows for small nooks in the mesh is computationally expensive as it requires computing many bounces of the lighting reaching the surface. As such ambient occlusion is a value which can darken the final color by a preferred amount. Similarly to normal mapping, occlusion maps are used in order to introduce an extra level of detail.
- Emission : The emission color is useful for all PBR shaded meshes which are supposed to show parts of the mesh emitting light. As such this parameter also often takes maps as input in order to show lit up sections of the rendered mesh.
- Alpha and Alpha Clip Threshold : These two values are used in all shaders which can render transparent and translucent objects. They influence the final color by taking into account what's behind the rendered mesh. Since rendering transparent objects is often much more expensive than fully opaque ones, Shader Graph allows toggling on and off transparency in order to save resources.

All of these values contribute to the final computation of the pixel color, by being used in Unity's rendering process, using a different algorithm depending on the type of shader selected.

Ultimately, Shader Graph only allows to program the Vertex Shader and the Fragment Shader stages of the entire rendering pipeline, but the Shader Graph package also allows the definition of custom nodes, which can also access the capabilities of the Geometry Shader stage.

2.4.3 Shader Code Generation

In order to generate a fully functioning shader from the graph representation, the Shader Graph package parses the graph and generates code snippets based on the nodes present and their topology. For each node, all the inputs are mapped to variables in the shader code and fed into pre-built functions defined for each node representation. The same process is applied to outputs which are also mapped to internal variables.

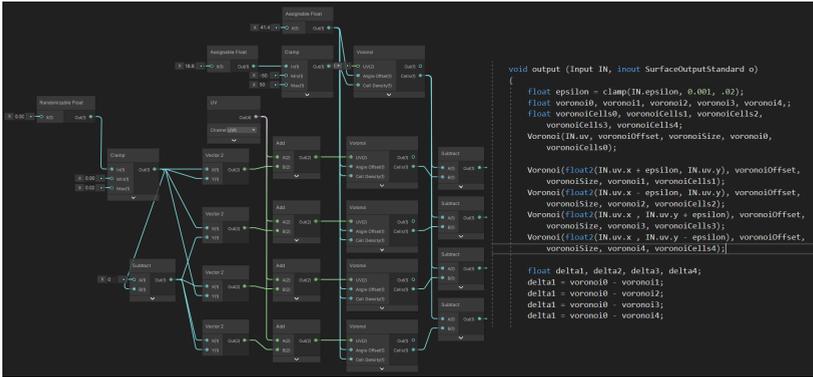


Figure 2.6: Graph representation of a section of shader code written manually. Demonstrating visual bloating.

Ultimately, the final shader will declare as many variables as there are inputs and outputs while reusing the same declared function in case the same type of node is reused in the graph.

While there are optimization at compilation time, this generation process creates fully functioning shader with a slight performance overhead due to abundance of extra variable declarations and the presence of definition of functions which only wrap HLSL functions. Moreover, all function calls and operations in the shader's code are represented by single separate nodes, which can bring the resulting graph to bloat quite rapidly compared to a manually written code operating in the same way, as shown in Figure 2.6.

2.5 Genetic Algorithms

The term Genetic Algorithm is used to refer to a whole class of algorithms which emulate the process of natural selection in order to optimize solutions to a given problem. The way the algorithms emulate natural processes is through the reproduction of processes such as mutation, fitness based selection and crossover of genes between two specimen, applied to multiple specimina in a given population[18].

In order to operate these processes, all generated specimen are characterized by a chromosome, made by multiple genes. Genes are pieces of data which represent one of the many properties characterizing a given

specimen. Chromosomes are usually represented through an array of bit to ease on the evolution processes but other heterogeneous forms of chromosomes can also be implemented, where each gene has its own data format (In the work presented, the graph model as well as values of the inputs given to it will be both used in the chromosome representation). Evolution needs both the genetic representations as well as the definition of a fitness function in order to work properly.

The fitness function dictates the likelihood of reproduction of each specimen in the population, thus establishing the chances of it passing down its genes to the next generation compared to the others specimen in the population. This fitness function can be defined in a variety of ways which has to be closely tailored to the problem at hand, in order to maximize the effectiveness of the algorithm.

Essentially, after a new population is generated, a number of specimen will emerge that will present more favourable features compared to the rest. These specimen are then used as a base for producing a new population. The new generation will be produced by taking the fittest genes and operating two different processes:

- Crossover, in which chromosomes will be selected in couples from the parent population and a new chromosome will inherit part of one parent's genes and the rest of them from the other. This can be done in multiple ways such as single or multiple point crossover, in which the ordered list of genes is split at one or more points, or uniform crossover, in which the inherited values will be chosen at random between one of the two parents.
- Mutation, where the new chromosome will have the chance of mutating one or more genes in the representation, by changing the representing value. This is done in order to reintroduce variety into the population and allow better representations to be passed down.

Through the use of the described processes, each generation of specimen should ideally show characteristics which solve the given problem in a more efficient way than the previous one. The main limitation of Genetic Algorithms is given by the fitness function itself. Given that it is the

main decision driver for choosing the best solutions, the solution is going to be only as good as the fitness function itself. Furthermore, genetic algorithms can be subject to exponential increase in the size of the search space, when the number of elements being influenced is too high (which is a challenge that has been tackled during the production of this work).

Before going into the project presented in this document, it is important to look at the work that has already been done. The material produced around the subject has been useful as a reference for the work presented and it might also be of inspiration for future improvements on what has been done. This chapter will present work that also aimed at simplifying the work for artists using different frameworks, as well as a framework specialized in simplifying a shader, which may prove useful as a future addition to the product of this work.

3.1 Graphical Editors

3.1.1 Introduction

Shaders were first introduced as a mean to allow user control over the shading process[2]. The idea of directly managing the rendering process on a per-object basis quickly gained popularity and, before long, shading languages became an industry standard, with both major graphics APIs having their own syntax.

Having a language specific to just handling the rendering process meant that developers had to learn a new language just to handle a small part of the entire development process. This solution has been less than ideal, since the usage of a programming language would lock out many artists and less technical users out of the autoring of the shading process.

Thus, there have been steps made towards bridging the gap between less technical users and process of shader customization.

At first, collection of pre-programmed shader programs called shader libraries were built, such as NVIDIA's Shader Library[13] and Geeks3D's Shader Library[8] (now both unused).

In recent years, visualization tools which not only offer instant feedback but also allow less technical users to edit shader code have progressively gained popularity. Among these we can recognize online shader editors and Graph-Based shader representations as being the major drivers of such change.

3.1.2 Online Editors

The first work done on making shader curation more accessible to less technical users has been focused on visualizing the resulting shader right away, in order to give users visual feedback of the program they were writing and changing. The most relevant work implemented an online version of a OpenGL Shading Language (GLSL) editor leveraging WebGL, a Javascript API for the Open Graphics Library (OpenGL), as a mean to render the shader result in real time.

- **GLSL Sandbox** (Figure 3.1) has been the first project that lets users have instant response on the written shaders. It is an online editor offers a wide selection of shader programs made by the community. Each shader can be inspected as well as used as a template for a new shader made by the user, who can also create a shader from scratch.

Each change to the shader code is instantly reflected onto the editor, which shows the resulting shader as a background of the editor. It is important to note that GLSL Sandbox allows the editing of the code responsible for just the Fragment Shader stage, it therefore does not allow manipulation of neither the vertex shader stage nor the geometry shader stage[9].

- **Vertex Shader Art (VSA)** (Figure 3.2) has been heavily inspired by the work done on GLSL Sandbox and thus it reflects much of the same characteristics UI wise. VSA is also an online editor, which instead allows users to only edit the Vertex Shader stage of the shader used by the renderer built into the online editor. The main difference between the previous described work lies

in the limitations imposed on the users, as the Vertex Shader is run on a per-vertex basis and thus the values in output will be of lower resolution compared to the fragment shader. On the other hand, the vertex shader gives the users control over new values such as the vertex global position in the final rendered scene as well as normal and tangent vector directions for each vertex in input. The online editor lets users control the amount of vertices on input as well as the primitives associated to each vertex. (Line Strips, Points, Triangles etc...)[27]

- **ShaderToy** (Figure 3.3) is an online editor which also hands users control over the Fragment Shader in the GLSL rendering pipeline. It has built over the initial concept implemented by VSA and allows users to manage multiple input buffers as well as receive Human Interface Device (HID) input[12].

Through the use of multiple dedicated buffers, users can thus add textures and maps which can be leveraged by the program being written.

ShaderToy acquired a great amount of popularity among users and community evolved around building complex scenes through the use of Ray Marching.

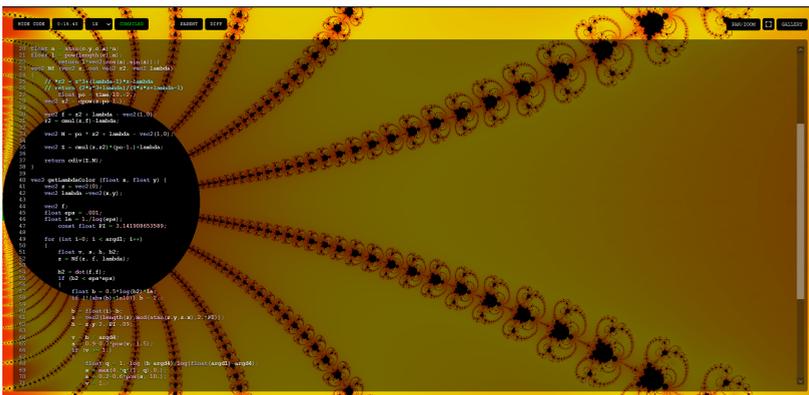


Figure 3.1: Example of GLSL Sandbox's UI, from [7]

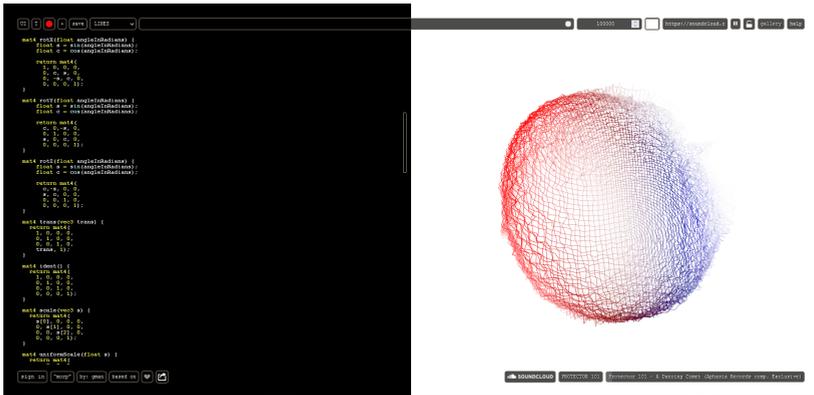


Figure 3.2: Example of VertexShaderArt’s UI, from [6]

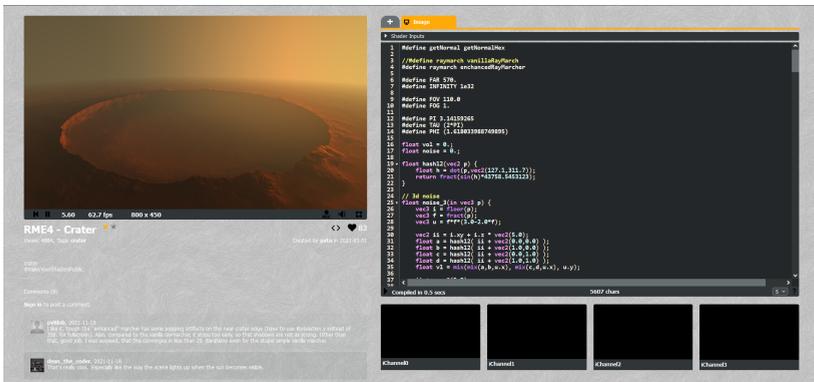


Figure 3.3: Example of ShaderToy’s UI, from [4]

3.1.3 Visual Editors

As stated previously, Unity’s Shader Graph isn’t the only visual editor for shaders that has been released to the public. Before the introduction of such extension into Unity, there were already some third-party made visual editors made with the purpose of giving control of shader programming to users in the most simple way possible.

Furthermore, one of Unity’s competitors, Unreal Engine, has also released a visual editor for shaders to the public in recent years.

- **Shader Weaver** is a third party Unity Extension which imple-

menting a visual editor for shaders restricted to 2D rendering[22]. It is part of the tools offered on Unity's Asset Store and gives users a visual scripting approach to the creation of 2D sprites.

Shader Weaver relies heavily on the usage of prerendered images (as is the case of most 2D shaders) but gives the user a wide array of choice when it comes to modifications and post-processing effects applied to the input images. Through Shader Weaver, the user can also overlay the input images in customizable order and animate the final 2D Sprite through image manipulation offered by the tool.

Ultimately, Shader Weaver is a tool specific for 2D shaders which can be much more comparable to a visual image editor rather than a full fledged shader editor.

- **Shader Forge** (F. Holmer, 2014)[20] is an extension available for Unity, developed up until 2018. It implemented a visual editor for shaders very close to the current version of Unity's Shader Graph. Shader Forge offered a real time preview of the produced shader and implemented helper-nodes which would add complex shader features such as screen-space refraction and the possibility for users to define a per-light custom function, enabling more technical users to implement their own custom lighting processes. The tool was mainly focused on simplifying the shader creation process as well as streamlining more complex shader functions such as edge detection, vertex animation and transparency effects.

Once the user completed the creation of the shader, the tool would generate a .hlsl file which can be parsed by unity as a custom shader inserted by the user.

- **Amplify Shader Editor (ASE)** is another visual editor for shader that has been produced for Unity. Differently from Shader Forge, Amplify Shader Editor is still being developed and supports the new Render Pipelines implemented into Unity, which give greater control over shader functionalities[1].

ASE also uses a graph-based representation of shaders and it offers many more features when it comes to shader editing. For starters, the users is able to both create new shaders as well as start off from

premade template which cover the most common use cases, further simplifying the creation process for less technical users. While the tool offered advanced features for the old Unity Renderer such as control over stencil buffers, tessellation process and translucency, the new version is based off the Unity's Shader Graph package, thus making the tool much more similar to the Unity sourced tool.

3.2 ShaderBase in Processing

The Processing project is a piece of software used in interactive media, teaching and data visualization. The main objective of Processing is making Computer Graphics (CG) more accessible to individuals such as artists and designers, with the final aim to "increase computer literacy within design and visual arts, and visual literacy within technology and engineering." [17]. As Processing's main goal is to streamline the process of creating visual experiences through code, it offers an array of preset rendering paths with different shaders built into them. All shader values can be accessed through Processing's API called PShader [15].

Beyond simply managing shader values, PShader also allows the usage of custom shaders written by the user, which still have to be written in GLSL. Thus, users with little knowledge of shading languages won't be able to change the rendering process through shaders, which can be a waste in both resources and customizability.

ShaderBase (Gómez, Charalambos and Colubri, 2016)[10], simplifies the process of custom shader creation by creating a repository of available shaders which can be included through the ShaderBase interface. Through an underlying Git repository, ShaderBase allows users to upload, download and edit shaders which can be directly brought into their project through the Processing's API.

This enables all users to both rapidly implement new shaders that can fit their needs as well as learn GLSL rapidly through the visualization of different examples, made by the Processing community itself.

Ultimately, though, ShaderBase streamlines the process of technically creating shader but does not directly solve the problem of having to learn a new language and managing the rendering process, with all the complexity attached to it.

3.3 Genetic Algorithms applied to Shaders

The idea of applying some version of a Genetic Algorithm (GA) to programming, and specifically to shader programming, isn't novel in itself. As a matter of fact, there has been prior work which applied evolutionary computation to the generation of shader code. Most relevant work limited the search space of the GA to only one aspect of the final shader program. In most cases, programs were often represented through a tree structure and were finally implemented in GLSL code, thus not tackling the problem of code literacy when producing the shader program.

3.3.1 Procedural Generation of Vertex Displacement Shaders

The main issue posed by shader program code is their lack of flexibility. While the managing of input values is straight forward and easy for any user to manage, given that memory needs to be optimized and that branching evaluation is an expensive operation during the rendering process, making a shader program that covers multiple use cases, even if those cases share similarities, might not be as straight forward as it seems.

In order to tackle this problem, Quiroz and Dascalu's work (2016)[16] applies Genetic Algorithms to the code of a vertex shader, trying to explore the space of different types of vertex displacement that can be applied to the rendered mesh. The work consisted in developing a web application, able to render any input mesh. The aforementioned application would then render the input mesh using a variety of shaders generated through evolutionary algorithms. The user is then able to choose the specimina of the population that best fit their needs, the chosen individuals are then used as parents for a new generation that iterates on the selected characteristics.

Each shader program only influences the final vertex position of the mesh and is represented through a tree structure. The tree structure can be modified by the GA through the addition of operational nodes and single value nodes. Operational nodes represent the most common operations used when applying vertex displacement: addition, subtraction, multiplication and division as well as single value operations such

as sine and cosine (Figure 3.4); single values nodes are instead the leaves of the trees and represent numerical values.

The results from the generated operations, ultimately, is uniformly applied to all components of the three dimensional vector representing each vertex's position, displacing the vertex on input by the same amount in all three directions. The Genetic Algorithm, in this case, is used to explore only the different results obtained by using different kinds of vertex displacement shaders.

While the produced work can certainly be considered useful in some specific cases, it covers only one category of shader programs (vertex displacement shaders) and many of the iterations produced destructive modification of the mesh. Furthermore, the web application is agnostic to the mesh being processes and thus applies the resulting modification to the entire mesh, not keeping into account specific characteristics.

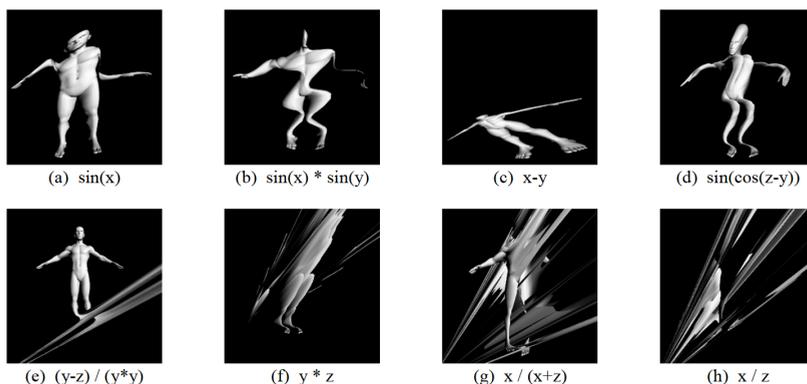


Figure 3.4: Visualization of some results from [16]

3.3.2 Evolving Pixel Shaders in a Video-Game

While vertex shaders can manipulate the final shape and some lighting characteristics of a rendered object, pixel shaders are responsible for the final color of the object, shading included, which usually makes up the most part of the appearance of the rendered mesh. It is thus useful to see what can be done to change fragment shaders programmatically as they are of great relevance to artists and designers due to the great impact it can have on the final look of the rendered scene.

In the work done by Howlett, Colton and Browne[11], a similar genetic algorithm approach with user interaction has been applied to fragment shaders, in order to render the landscapes generated in the game Subversion with different shades of color which would highlight some areas rather than others. In this game, the user is presented with an aerial view of a procedurally generated city which, by default was rendered fully black with white edges (Figure 3.6).

The shader program generated by each iteration changed the way the virtual city is rendered, by changing the shades of color of different parts of the city itself. The Genetic Algorithm implemented into the project also used a tree representation of the final fragment shader. As the game the work has been done on is a web application, the use of the WebGL API dictated the use of GLSL as the target language in which the shader produced by the algorithm would be implemented in.

In such work, the issue of letting the user change the shader code hasn't been tackled since the end user of the product would be a player of the web game, thus limiting the interaction with the shader program to a set of dedicated sliders through a UI which would guide the genetic algorithm to the preferred solution. As the project has been entirely implemented into the Video-Game mentioned previously, there has been further optimization of the genetic algorithm, through the implementation of a fitness function designed by the developers. The fitness function used parameters given in input by the user, such as hue, saturation and brightness of the color palette in the final representation of the game. This would affect different parts of the city in a different way, since a carefully chosen heuristic could guide the final shader to explore the search space in a way optimized for the final use case.

Ultimately, in this project the search space defined for the genetic algorithm is quite restricted compared to the full possibilities that can be leveraged from shader programming, but the results obtained show how iteration on fragment shader can yield results which have great impact on the final look of a render.

3. STATE OF THE ART



Figure 3.5: Standard rendered view in [11]

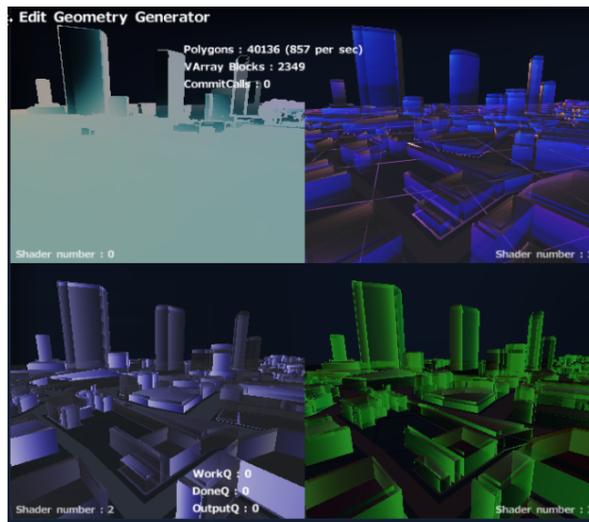


Figure 3.6: Different views with procedurally generated shaders, from [11]

This chapter will go into detail about the work that has been done and the brought contribution. Firstly, there will be a description of the work done in order to interface the Shader Graph code with external programs, followed by an overview and description of the Genetic Algorithm's design as well as the design choices for the genetic representation and challenges of defining the scope of the search space for the genetic algorithm.

4.1 Introduction

Looking back at the work that has already been done, it is clear that solving the problem of technical literacy can be beneficial to incentivizing usage of custom and dedicated shader among less technical users. The main issue to tackle right now is given by the fact that node-based representation still requires the user to have technical knowledge on the render pipeline and shaders' workflow. Furthermore, graph representations tend to scale up in size and complexity quite rapidly, making the process of iterating on the produced shader a time consuming task, as mentioned in Section 2.4 .

The objective of this work is thus offering users a way to quickly iterate on the shader being built, as well as adding one more layer of abstraction to shader creation, allowing less technical users to approach shader creation more easily. At the same time, it can be beneficial to offer users the graph representation of the generated shader program to im-

prove understanding of shaders, thus leading to the choice of leveraging Unity's Shader Graph to generate shaders.

4.2 Interfacing with Shader Graph

In its current state, Unity's Shader Graph does not offer a public API, allowing only internal functions to access the data used for building the graph. This brought up the need to slightly modify Shader Graph's code base, in order to add a way to manipulate the graph through classes outside the package's content.

4.2.1 Code Analysis

As the main objective of this work is building a Unity tool which leveraged Shader Graph's model and capabilities, the scope of the code analysis will be the code section responsible for representing shaders through the model of a graph, which is relevant for the work presented.

The software implements a textbook Model-View-Controller (MVC) design pattern, where the view updates the controller when the user interacts with the UI, the controller is then responsible for changing the underlying data model which will, in turn, also produce changes in the view, allowing the user to receive visual feedback of their actions.

- **Model**

The model in the MVC pattern is implemented through a collection of C# classes, all handled by the `GraphData` class. The main class also doubles as the model of the graph representation itself. Due to this, it is characterized by all the methods used to manipulate the underlying model such as the `AddNode`, `RemoveNode` and `Connect` methods, whose function is self explanatory.

The class also contains methods for managing all other aspects of the Shader Graph (for example: the `activeTargets` field enumerates the render pipelines the shader supports) as well as validation functions which check the graph's correctness and the compatibility between connected nodes.

The graph representation keeps track of the nodes in the graph and the connections through collections of two model classes: `AbstractMaterialNode`

and an `Edge` class. While the `Edge` class is taken from the Unity's programming API and does not contain any extra information about the shader, the `AbstractMaterialNode` class is instead a custom class defined for representing sections of shader code as well as input and output data from the represented section. The abstract class is extended by a wide array of classes which represent the different nodes that the user can add to the graph and ultimately offers all information regarding the node in the graph as well as the needed information for converting the produced graph into working shader code.

The node representation class contains data about the slot that make up the inputs and outputs of the node through the `MaterialSlot` class, which is also an abstract class that represents all possible inputs and outputs characterizing a single node. The slot class is responsible for handling compatibility checks as well as keeping track of the input values to the node when the slots are not connected into the graph.

When converting the graph into shader code, all abstract node class implementations inheriting the interface `IGeneratesBodyCode` will have their method `GenerateNodeCode` called, which will return the section of code for which the node is responsible.

Once the body of the code is generated, in the case there are nodes which represent a more complex function or a wrapped HLSL function, an optimization process will ensure such function is defined only once, even if there are multiple instances of the same type of node.

- **Controller** In the code, the controller role is taken up by a collection of classes, each fulfilling a different part of the role, in order to manage the `.shadergraph` files containing the information about the graph representation and applying the changes to the model when the user does so through the UI.

As Shader Graph also allows saving the created graphs for later modifications, all relevant information is encoded into a json. In order to do so, the `GraphObject` class is responsible for ensuring a correct serialization of the entire model as well as handling any

issue that may arise. It is thus the section of the controller responsible for reacting to when the user chooses to save the currently edited graph.

The remaining part of the controller is responsible for updating the model when there are new connections added from the user, when there is a request to generate a new node and when the shader properties are changed. All of this information, together with the visual topology of the shader, is maintained by the `GraphEditorView` class which reacts to most of the new events generated by the view.

- **View** While the classes making up the view of the Shader Graph are many and each one covers different roles, for the purpose of presenting the work, the focus will be set onto the class responsible for representing the graph itself. All user interactions with the graph itself are handled by the `MaterialGraphEditWindow` class, which is also responsible for handling the Editor Window into Unity itself. For each action the user can take, listeners are setup so that an event is triggered when the user interacts with the view, as it is done for any standard view.

Ultimately, in order to be able to interact with the Shader Graph model, the written code will be interacting with the `GraphObject` class and `GraphData` class so that it is possible to modify each specimen in the population and save the resulting graph separately.

4.2.2 Interface Classes

In order to interface the tool with Shader Graph's code, there has been the implementation of two main classes which handle graph generation, modification, validation and file management : `GraphInterface` and `GraphManager`.

- **GraphInterface** is the class responsible for operating file management and creation when it comes to the `.shadergraph` files. Since the process of creating a new file is separate from the management of the model itself, this required interfacing with Shader Graph internal classes at a different stage. It is possible to do so through

this class, which creates a new **GraphManager** instance when loading a new `.shadergraph` file through the method `InitGraph`. It is also possible to create new graph representation, specifying graph type, path and name of the file itself.

- **GraphManager** is the class that will be mainly used throughout the entire implementation of the work, as it is essentially the substitute to the Controller in the MVC design pattern. When instanced, the class maintains the state of an internal **GraphData** instance that represents the graph model. Through this class it is possible to directly modify the graph as well as operate more complex operations such as transferring sections of a graph from one file to another and graph traversal to find all the nodes to which a given node is connected directly or indirectly.

In order to accommodate for new classes which can interact with Shader Graph's code, some additions have been made to the original source code which do not hinder the piece of software's intended operation.

4.3 Genetic Algorithm

4.3.1 Overview

The genetic algorithm implemented into the tool reuses the Shader Graph model coupled with a collection of value ranges as the genetic representation of the shader being produced. The chromosome representation will thus be made up of multiple sub-trees, each one flowing into one of the output nodes of the Shader Graph model.

As often the values which dictate the final render depend from the same computations, the same graph might be used as input for multiple output nodes, with some changes depending on the intended output(as seen in Figure 4.1). This allows to implement different behaviours depending on the aspect of the shader being modified.

All leaf nodes will have their input slots not connected to anything and will instead be characterized by an associated value. In the case of slots requiring color values, for example, the user will be able to select a pre-determined color as input to the node. These free input slots are used in

4. IMPLEMENTATION

the genetic representation to command the graph's input values. To each slot a range of values is associated and used as a gene in the representation, thus allowing specimen to pass it down to the new generation as well as mutation of such inputs. The generated values keep into account the topology of the graph and the intended use of the input and allow a higher degree of variety in the programs generated. The associated ranges are chosen per specimen and dictate the range in which the value taken by the free input slot will be chosen from, uniformly at random.

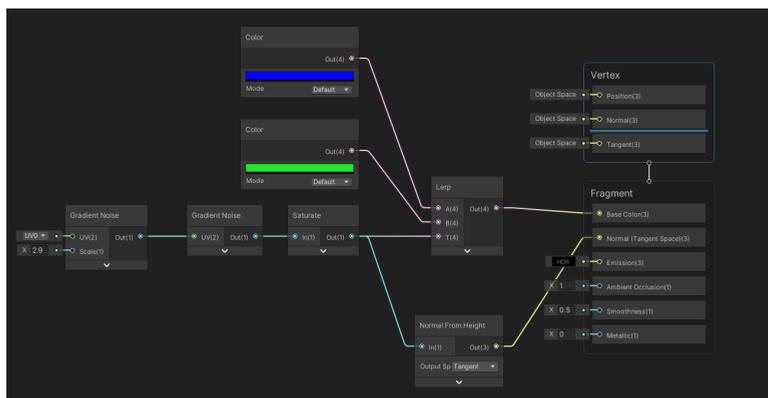


Figure 4.1: Example of a graph flowing into multiple output nodes (Base Color and Normal)

The graphs are generated through the `GraphInterface` class and the user is responsible for setting the number of individuals generated for each generation, the number of successive mutations applied to each individual at each generation and the mutation strength, which can be set to either Low, Medium or High, dictating which of the sub-trees will be modified, thus changing the visual impact that the new mutations will have. The process of creating new graphs is described further in Algorithm 1.

The generated population will then be displayed to the user, who will be responsible for choosing the generated shaders that best fit their needs. The selected shaders are then used as parents for a new population. In order to generate new specimens, a single point crossover is operated over the output nodes of each one of the two parents, thus inheriting all sub-trees from the first parent up until one output node

and the remaining sub-trees from the other specimen selected as parent.

Algorithm 1: Graph Population Generation Function

```

Data: int size  $\geq$  0, int successiveMutations  $\geq$  1 ,
          MutationStrength strength;
Result: List of generated and mutated graphs
List<GraphManager> generatedGraphs;
for i = 0 to size do
    /* Choose the preset to use */
    if RandomBoolean() then
        | GraphType type = GraphType.Lit;
    else
        | GraphType type = GraphType.Unlit;
    /* Create new graph file and model instance */
    GraphManager graph =
        GraphInterface.GenerateNewGraph();
    /* Generate mutations as many times as chosen by
       the user */
    graph.MutateGraph(successiveMutations, strength);
    /* Generate input values for the marked input nodes
       */
    graph.GenerateInputValues();
    generatedGraphs.Add(graph);
  
```

Algorithm 2: Crossover Function

```

Data: GraphManager parent1, parent2, child;
Result: Graph Generated From the Crossover
int splitPoint = Random.Integer(0, length(child.outputNodes));
for i = 0 to length(child.outputNodes) do
    if i  $\leq$  splitPoint then
        | child.inheritGraph(parent1, child.outputNodes[i]);
    else
        | child.inheritGraph(parent2, child.outputNodes[i]);
  
```

After children individuals have been generated from the parents, the number of successive mutations set by the user is used to determine the

number of times the child graphs will be mutated by the algorithm. The mutation selects from a pool of mutating functions, removing the ones unwanted by the user. The user can set which functions to include into the mutation function through toggles present in the UI.

Algorithm 3: Mutation Procedure

```
Data: bool allowGraphExpansion, allowPresetChange,  
         GraphManager graph  
Result: Resulting Graph after the mutation  
/* list of all possible mutating functions */  
List<Function> mutatingFunctions;  
if allowGraphExpansion then  
    └ mutatingFunctions.Add(expandGraph);  
if allowPresetChange then  
    └ mutatingFunctions.Add(changePreset);  
...  
Evaluate compatibility condition for each mutating function  
...  
int index = Random.Integer(0, length(mutatingFunctions));  
Function chosenMutation = mutatingFunctions[index];  
chosenMutation.Invoke(graph);
```

The process is then repeated, until the user is satisfied with one of the results from the Genetic Algorithm.

4.3.2 Mutating Functions

Given that single node additions to the graph did not bring to satisfactory results, the different approach that has been chosen is the one of applying pre determined mutating functions, which modify the graph in a controlled way and ensure the result is still a working shader.

With this method it is thus possible to quickly generate shader program variations which also differ in terms of code lines instead of just a difference in pure input values.

Apart from the change of preset and the expansion of the graph model, there are three other mutating functions implemented into the Genetic Algorithm.

- **Swap Noise Maps**

Many shaders (as well as the references used), implement the use of noise functions in order to generate variations in the rendered mesh without facing the issue of repetition, which is usually a problem with pre-rendered textures that cannot be repeated easily without letting the user notice the redundancy. Shader Graph currently implements three of the most used noise functions: Voronoi, Simplex and Gradient Noise.

When a graph representing the shader program contains one of these noise functions, the algorithm can thus choose to swap the present noise functions with one of the other two, heavily changing the final image rendered. At times, the shader program will contain different calls to the same noise function, all semantically bound to each other. An example of this is the process of computing procedural normal vectors, as seen in Figure 4.3. The mutating function keeps this caveat into account, by checking the graph topology and swapping all semantically bound noise nodes as seen in Algorithm 4. This check can be done by examining the topology of the graph, thus determining noise function sharing some specific parameters as semantically linked.

- **Temporization of Inputs**

As most inputs are static and do not change over time, an interesting mutation on the graph is making some of the input values of the shader change over time, either making them scale with time or assigning periodic values through the use of the sine and cosine time functions. This usually leads to colors or the size of the feature which varies with time.

- **Changing Value Ranges**

Given that, for each specimen in the population, all graph inputs which can be modified are assigned to different value ranges, another mutation that has been implemented is changing these ranges, thus modifying the maximum and minimum range of values that the inputs can take. In order to avoid generating ranges too big which would distort the final rendered image beyond use-

fulness, these ranges are defined inside a range itself, which bounds the maximum possible values regardless of mutations.

Algorithm 4: Swap Noise Function Procedure

Data: GraphManager *mutatedGraph*
Result: Resulting Graph after the mutation
List<Node> *noiseNodes* = *mutatedGraph*.getNoiseNodes();
int *index* = Random.Integer(0, length(*noiseNodes*));
Node *chosenNode* = *noiseNodes*[*index*];
int *noiseType* = Random.Integer(0,2);
swapNoiseNode(*mutatedGraph*,*chosenNode*,*noiseType*);
for *noiseNode* in *noiseNodes* **do**
 if *noiseNode* is semantically linked to *chosenNode* and
 shares class type **then**
 swapNoiseNode(*mutatedGraph*,*noiseNode*,*noiseType*);

4.3.3 Defining the Search Space

One of the main issues to tackle when generating shaders programmatically is handling the vast variety of use cases and types of shaders that can be of use in the production process. Furthermore, in order to be able to generate more complex shaders (compared to the simpler ones seen in Sections 3.3.1 and 3.3.2) compatibility of input and output values as well as code correctness needs to be accounted for. Dealing with this amount of complexity led to the choice of leveraging Shader Graph's own internal shader validation process, which leveraged the same graph model and node definition to check compatibility functions and variables called inside the shader. This ensured that the graphs and shaders being built were first and foremost built correctly.

Given this choice for ensuring shader correctness, the natural consequence has been to also use the model implemented into the Shader Graph package as the equivalent of the tree representation used for shaders in the work that has been presented previously. These design choices brought the benefit of being able to use the internal graph validator during crossover operations and the capability of generating more

complex behaviours in shaders such as noise map generation, which are offered as prebuilt nodes by Shader Graph itself.

Given that the issue of producing functioning shader was being dealt with by Shader Graph's validation process and some complex behaviours were already built into its node definitions, a first attempt at generating procedural shaders was done through the simple randomized generation of a graph which could then be expanded in successive generations. This implementation allowed the user to produce a number of individuals, each one generated randomly where new added nodes were chosen uniformly among all the usable nodes that were offered by Shader Graph. While this approach produced functioning shader which, at times, produced some interesting results, the majority of the shader did not have any plausible practical use or did not change the final image in a discernible way. It was thus required to scope down the search space of the genetic algorithm, as well as guide it through the use of an heuristic.

4.3.4 Limiting Graph Types

By itself, the Shader Graph Package allows the user to generate either a completely blank graph with no preset output, which leaves everything to the user, or one of two preset shader types, which target, respectively, the generation of an Unlit Shader and the one of a Lit Shader.

The main difference between the two resides in the fact that the Lit Shader version defines more output values in the Fragment Shader stage, which determine parameters used in PBR to produce the final image. Having to define every part of the shader from scratch can extend the search space beyond manageable size, it was therefore chosen to base all graphs generated by the GA off these two presets in order to have a manageable search space.

During mutation, individuals of the population also have the possibility of changing from one preset to the other, in case this change can be of use to the final user. Ultimately, the family of generated shader will vary depending on the preset:

- The **Unlit** preset, given that it produces shaders which do not react to light, will ultimately generate more abstract shaders which

will be of use in data visualization, graphical design and 2D applications.

- The **Lit** preset will instead be able to generate a wider range of shaders which can be of use in other fields such as architectural visualizations, 3D applications and Video-Game graphics.

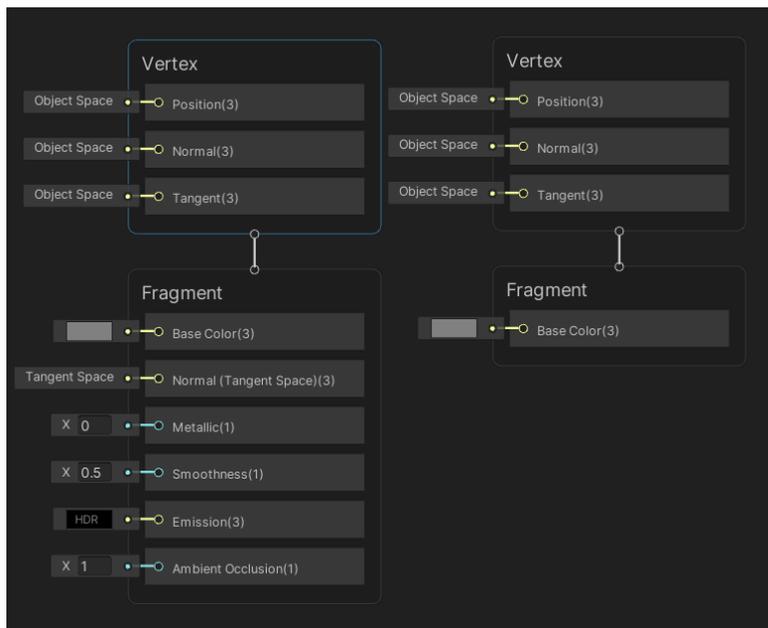


Figure 4.2: Left : outputs for a shader using the Lit preset.
Right: outputs for a shader using the Unlit preset

4.3.5 Guiding the Genetic Algorithm

As stated before, randomly choosing new additions for the generated graphs didn't lead to the production of useful shaders.

There was therefore the need to implement a set of criteria which would guide the Genetic Algorithm when applying mutations to each individual in the population, as well as a way to identify good crossover points when producing the offspring from the selected specimen.

For this purpose, two things were implemented which proved useful in

creating meaningful output.

- **Scaffolding**

A set of prebuilt graphs are taken as reference from the Genetic Algorithm when expanding the graphs in the population. Through the use of such references, the GA can take sections to then insert into the population's specimen and thus implement more complex behaviour which would be arduous to replicate through the application of multiple single-node addition. An example of such behaviour can be the computation of the gradient from a two dimensional map in order to generate a normal vector at runtime (as seen in Figure 4.3, where the mesh rendered is a plane but per-pixel computed normals allow water-like reflections). These reference graphs also serve as a way to guide the Genetic Algorithm towards producing useful shader programs that can fit one of many use cases.

Currently more than 20 reference have been built, but the set of graphs which the algorithms takes into consideration can be quickly expanded through the Shader Graph tool itself, by adding new `.shadergraph` files to a specific repository inside the tool's directory. In order to be usable by the tool, specific nodes have to be added to the reference graph in order to inform the algorithm about the sections of shader program which can be cut up without losing functionality.

Ultimately, either when generating a new run of the genetic algorithm or when expanding the graph of a new specimen from an existing parent, the `ExpandGraph` function will be called where the scaffolded graphs are picked as reference.

- **Marker Nodes**

The vast variety of shader programs as well as the necessity of keeping some values constant in order to generate a working shader, dictated the need to define a way to discern between nodes which produce meaningful changes when the inputs are modified and nodes whose inputs should not be changed, thus avoiding degrad-

ing the shader program's function. In order to do so, a set of special nodes have been implemented which can be parsed by both Shader Graph and the Genetic Algorithm, which can be used as targets of both value changes and graph expansions.

Adding these marker nodes allowed the algorithm to generate meaningfully variations of the code sections being taken from the reference graphs, while leaving the critical code sections untouched, thus producing a more meaningful output. Apart from these marker nodes, the only other nodes which the algorithm deliberately modified or attached graph sections to are the final output nodes.

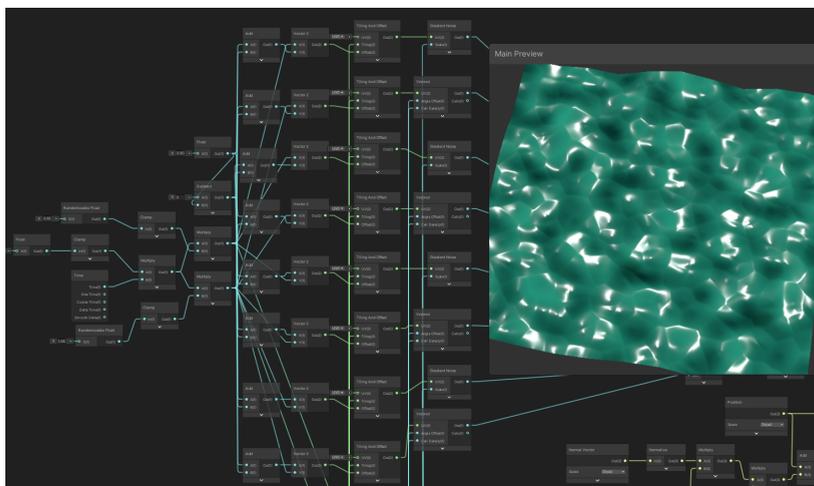


Figure 4.3: Procedural Normals through gradient computation. A behaviour difficult to reproduce procedurally.

Therefore, when calling the `ExpandGraph` function, the algorithm will parse the graph, finding all marker nodes inside the specimen's graph model as well as all output nodes. Starting from the collection of node which can be linked to new section of graph, the algorithm will filter all reference graphs which are compatible with the nodes present in the model. The function will then choose uniformly among the reference graphs and select the appropriate section from the reference which can be applied to the model being modified. It will then copy each node present in this section onto the model and finally, link the output of the section with the marker node.

Algorithm 5: Procedure for choosing a reference for graph expansion

```

Data: GraphManager graph
Result: Resulting Graph after the mutation
/* list of all marker nodes and output nodes in the
   graph                                                    */
List<Node> nodes = graph.getMarkerNodes();
nodes.Add(graph.getOutputNodes());
Node expansionNode = Random.Choose(nodes);
/* filter out incompatible reference graphs                */
List<GraphManager> referenceGraphs =
  FilterReferenceGraphs(expansionNode);
/* choose uniformly at random                              */
GraphManager chosenReferenceGraph =
  Random.Choose(referenceGraphs);
/* transfer nodes from reference to graph                  */
Node referenceOutputNode =
  chosenReferenceGraph.GetOutputFor(expansionNode);
graph.AddNodes(expansionNode,
chosenReferenceGraph, referenceOutputNode);

```

At the end of the expansion process the function will also verify if there are any dead nodes or dead sections in the produced graph (sections not connected to an output node indirectly or directly, thus not being useful to the computation) and such sections will be pruned off the model. Marker nodes are kept inside the model, thus allowing future mutations to also substitute section of the graph instead of just adding to it.

4.3.6 Genetic Representation

In the work that has been seen in Section 3.3.2, the chosen genetic representation of the final shader was the tree model with constant values being the leaves of the tree thus having single values as input. This representation, though, does not capture one of the most important aspects of shaders which makes up much of their flexibility and that is the possibility of setting external values which can greatly influence the final

image rendered through such shaders. It was considered useful then to iterate on the simple tree model and expand the genetic representation in a way that is also able to capture such variety through the different generations of the algorithm, thus allowing users to reiterate rapidly on the shader input values. This need, coupled with the necessity of being able to evaluate the range in which input values need to be generated led to the choice of slightly modifying the genes' representation, in order to make it more expressive.

Thus, beyond the graph model implemented into Shader Graph which doubles as a representation for some genes of the shader, a collection of input ranges is implemented which take into account the graph's topology. For each leaf node marked as modifiable, a range of values is generated by first evaluating the topology, thus generating a maximal range in which the node can take significant values. From this bigger

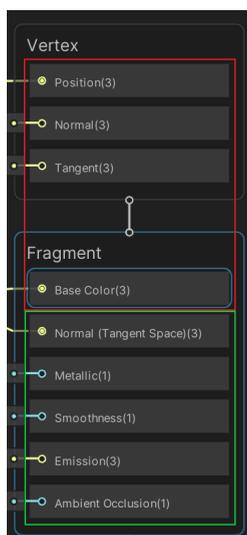


Figure 4.4: Examples of single point crossover done at the graph's output. In this case the red highlighted output nodes are taken from first parent, while the green highlighted ones from the second parent.

range a random, smaller range of values is generated and it dictates the values the associated input will be able to take. This was done in order to generate a good degree of variations for each given graph, while keep-

ing all variations inside a manageable range without distorting the final shader too much. Therefore, when a new specimen inherits genes from its parents, the steps for generating it are the following:

- **Graph Topology Crossover**

When inheriting from its parents, the genetic algorithm operates a single point crossover over the output nodes present in the graph model, as shown in Figure 4.4. This generates a new graph topology with nodes partly from one parent and the rest from the other. In case there is only one selected specimen from the user, the graph topology is kept the same.

- **Input Range Inheritance**

For each input node in the resulting graph, the range of values that the input node can take is inherited by the corresponding parent. Thus inheriting the range from the respective parent node.

If mutations applied to the specimen cause new modifiable input nodes to be generated, a new range for the input node is created and stored into the genetic representation.

The work presented thus far led to the full implementation of a Unity tool which allows fast iteration on shader creation and gives users with little knowledge of shading languages and rendering pipelines a way to have control over the shading process. This chapter will describe into detail how the resulting tool works, how a user can interact with it through the UI and what the user can do to expand the pool of reference graphs used by the tool itself to generate shader variations.

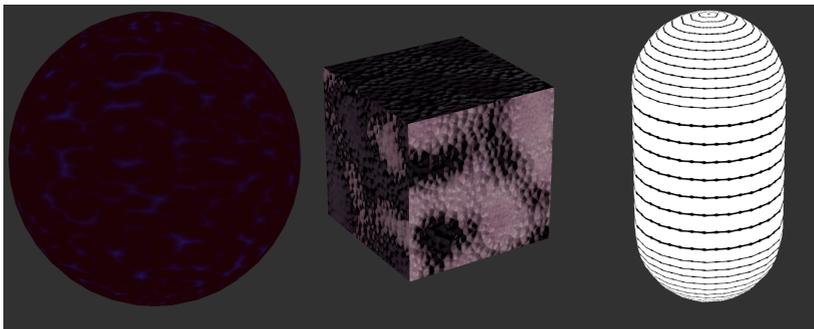


Figure 5.1: Some results from running the tool.

5.1 Tool Overview

The tool, which is called SGA, short for Shader Graph Autogen, can be imported into any Unity project through the package manager. Once installed, the user will be able to launch the tool through a dedicated

tab in the upper side of the Unity window, as shown in Figure 5.2.

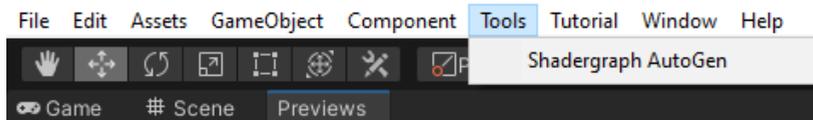


Figure 5.2: SGA can be launched by selecting the Tools tab into Unity.

Launching the tool will open up a Unity tab inside the editor, which will allow the user to tune different settings. The user will be able to first and foremost set the path where new generated graphs will be saved to, in order to better organize their work. The same tab will also display the number of reference graphs being loaded into the tool, as a useful piece of information. All the parameters used to tune the Genetic Algorithm, thus controlling the way new shaders are generated, will be accessible through the tab under the "Multiple Graph Generation and Randomization" section.

The user will be able to change the following values:

- **Generated Graphs Number**

Through this value, the user will dictate the number of graphs generated in each new generation. This will therefore influence the amount of variety present in a single iteration of the algorithm.

- **Successive Mutations**

With each new generation, the offspring from the selected specimens will be mutated a number of sequential times. This is done in order to change the amount of variability when creating new graphs. Through this setting, then, the user is able to set how many times the offspring will be mutated after inheriting genes from the parents.

- **Mutation Strength**

As stated earlier, the strength of the mutation can be set to either Low, Medium or High. This ultimately determines the impact each mutation will have on the visual result for each mutation, therefore High mutation will modify parts of the graph flowing

into the most crucial output nodes such as the Base Color (or Albedo) of the shader and the computed normal vectors, which sharply affect reaction to light by the rendered mesh. The user is able to change this value with each generation, allowing them to progressively narrow the variability when finding a shader fit to their needs.

- **Graph Expansion toggle**

In the case the user desires to explore only the possible variations of a given shader, without impacting the implemented code, the possibility of expanding the graph when mutating the offspring can be disabled. When enabled, a slider also gives control over the frequency with which such mutation can happen.

- **Type change toggle**

Depending on the needs of the user, changing the preset used to generate the graphs may be more or less desirable. Thus a toggle has been implemented which allows the user to choose for each new iteration of the GA if new offspring can mutate from one preset to the other and, if so, the probability with which this mutation can happen.

Finally, there are buttons which allow the user to launch a new run of the GA, generate new offspring from the selected graphs and preview the last generated population.

5.1.1 Managing runs of the algorithm

Once the user launches a new run of the Genetic Algorithm through the button shown in Figure 5.3, the program will proceed to generate the first population using the values set through the tool's window. All of the generated shader will then be presented to the user through a preview window showing one specimen at a time, as seen in Figure 5.4. The user will then be able to rotate the rendered mesh as well as change the mesh used for rendering. The tool offers, by itself, all standard mesh models such as sphere, box and capsule, but the user can select any custom mesh through the Custom Mesh option that appears when right-clicking on the preview itself. This is especially useful when previewing

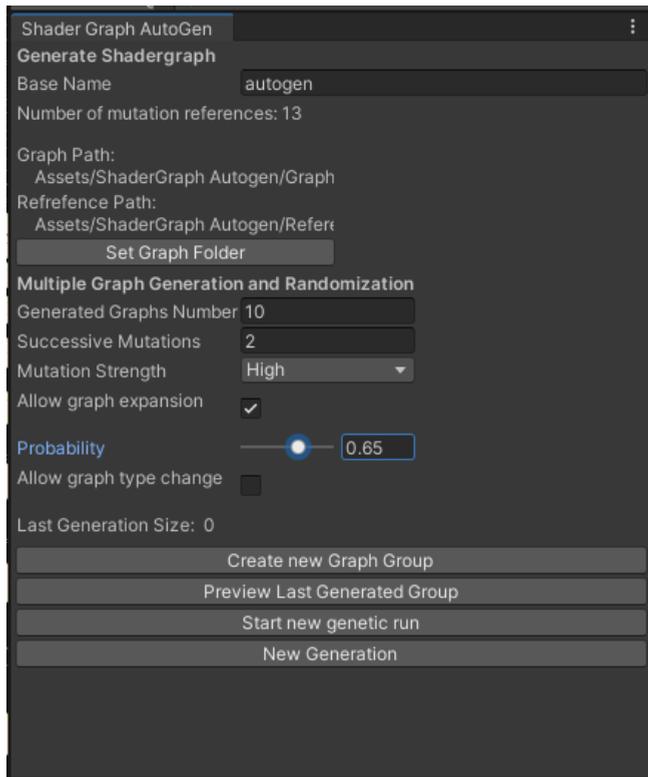


Figure 5.3: SGA's launch tab, with all the settings that can be changed by the user.

shaders that employ vertex displacement, as the underlying mesh can greatly vary the resulting modification.

In order to show the resulting shader under different conditions, apart from the blank background preview, the tool offers 3 more preview stages which can be selected through the preview window, using the buttons "Next Stage" and "Previous Stage". The stages currently implemented are the classic Cornell Box, which highlights different colored reflections, a dark room with moving lights which will better show how the procedural normal vectors react to light and finally a room with a checkerboard pavement. All these stages can be seen, respectively, in Figure 5.5.

Through the preview window, finally, the user can select which generated shaders are preferred and, if needed, generate a new offspring using the selected ones as parents for the generation. The new offspring

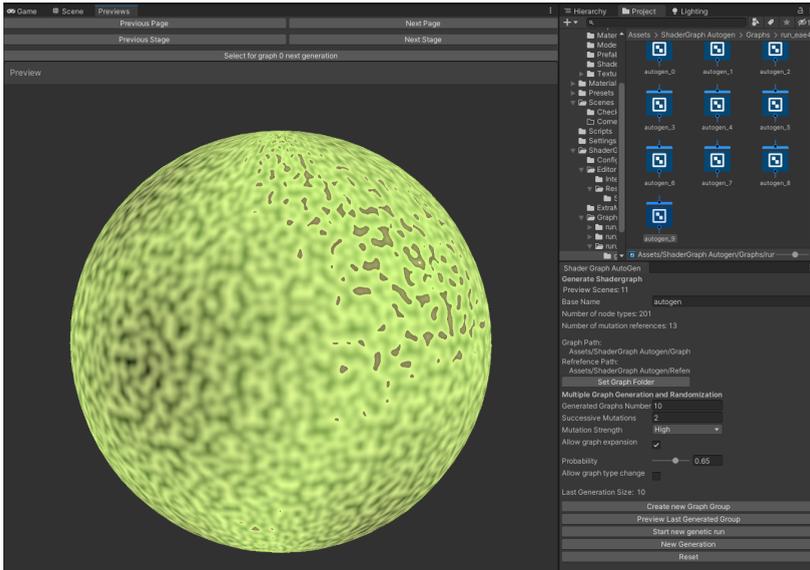


Figure 5.4: The preview will give the user the option to choose each specimen as parent of the next offspring

will then be previewed through the same window, replacing the previous generation. At any time, the user will be able to inspect the graph representing the generated shader, as each new specimen will be saved into a dedicated directory. The unity project inspector will be automatically pointed to that directory after each generation.

Running the tool allowed to generate interesting variations on different types of shaders, such as variations on shaders emulating a body of water and animated shaders for visual effects, examples of the results can be seen in Figure 5.1.

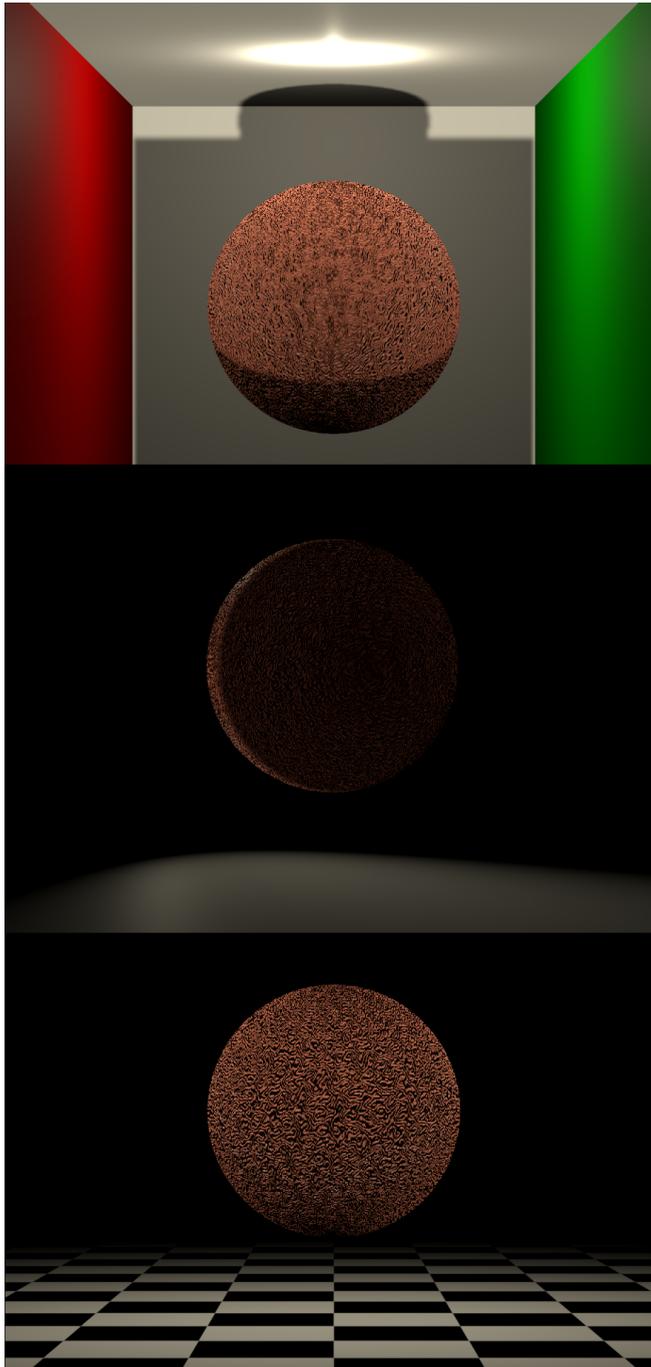


Figure 5.5: Preview stages available for the user.

We have presented Shader Graph AutoGen (SGA), a tool which allows Unity developers and artists to approach the world of shader creation and shader editing without the encumbrance of learning a dedicated language and without the need to learn the entire Shader Graph work pipeline. SGA also allows users with already some experience in shaders to quickly iterate on shaders and test different variations of a graph. There has been an in depth description of the challenges faced when using Unity's Shader Graph as a framework for producing this project and the reasoning behind the design choices that led to the presented version of the Genetic Algorithm.

Currently, the tool still has to go through a phase of user-testing, where its usefulness and real-world application will be experimented. Moreover, the algorithm currently chooses among a preset pool of mutation functions, which ensure the production of a working shader but tend to be limited in the range of variety they can offer. The pool of reference graphs, currently, is limited to 20 graphs which represent the most common use cases for a dedicated shader and that would save developing time, especially for smaller development teams.

In the end, the foundation has been laid for a project which might produce some interesting results and can see actual real world use, especially in the case of smaller development teams that need to maximize efficiency in the face of having higher economic restrictions and thus not being able to dedicate entire parts of the team to just shader programming. Therefore, while the graph-based representation are certainly a step forward in terms of increasing shader programming usage, we think

that automating the process of shader generation can be truly beneficial to an industry where time constraints are often the bottlenecks for quality in Video-Game and 3D applications production.

6.1 Future Work

Given that this work laid the foundations for a project which can become bigger, it would be first and foremost interesting to test the tool by expanding the pool of reference graphs significantly. In this way it will be possible to see what the algorithm manages to generate and, if needed, it might also be useful to implement an heuristic which filters out some reference graphs in order to better fit the user's needs. Another interesting addition to this work will also be an expansion of the mutation functions, as currently there are few which ensure a correct shader program creation. Broadening the range of modifications that can be applied to shader programs could bring forward novel and interesting variations of the shader programs.

Acronyms

API Application Programming Interface. 5, 6, 23, 24, 31, 34

ASE Amplify Shader Editor. 27

CG Computer Graphics. 28

GA Genetic Algorithm. i, ii, 2, 3, 20, 29–31, 43–46, 52, 53, 57

GLSL OpenGL Shading Language. i, iv, viii, 2, 13, 24, 25, 28, 29, 31, 65

GPU Graphics Processing Unit. 6

GUI Graphic User Interface. 14

HDRP High-Definition Render Pipeline. 12, 13, 15

HID Human Interface Device. 25

HLSL High Level Shading Language. i, iv, vi, 2, 6, 7, 10, 13, 14, 19, 35

IA Input-Assembler. 7

MVC Model-View-Controller. 34, 37

OpenGL Open Graphics Library. 6, 24

PBR Physically Based Rendering. 11, 15, 17, 18, 43

SGA Shader Graph AutoGen. i, iv, ix, 51, 52, 54, 57

UI User Interface. ii, 14, 17, 24, 31

URP Universal Render Pipeline. 2, 5, 12, 15

VSA Vertex Shader Art. 24, 25, 65

Bibliography

- [1] *Amplify Shader*. URL: <http://amplify.pt/unity/amplify-shader-editor/>.
- [2] A.A. Apodaca and M.W. Mantle. «RenderMan: pursuing the future of graphics». In: *IEEE Computer Graphics and Applications* 10.4 (1990), pp. 44–49. DOI: 10.1109/38.56298.
- [3] James F. Blinn. «Simulation of Wrinkled Surfaces». In: *Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '78. New York, NY, USA: Association for Computing Machinery, 1978, pp. 286–292. ISBN: 9781450379083. DOI: 10.1145/800248.507101. URL: <https://doi.org/10.1145/800248.507101>.
- [4] *Crater Shader from ShaderToy*. URL: <https://www.shadertoy.com/view/MlSBdt>.
- [5] *Direct3D*. URL: <https://docs.microsoft.com/en-us/windows/win32/direct3d>.
- [6] *Displacement Shader from VSA*. URL: <https://www.vertexshaderart.com/art/RnwjSt42YXLcGjsgT>.
- [7] *Fractal Shader from GLSL Sandbox*. URL: <https://glslsandbox.com/e#77117.0>.
- [8] *Geeks3D's Shader Library Website*. URL: <https://www.geeks3d.com/shader-library/>.
- [9] *GLSL Sandbox*. URL: <http://glslsandbox.com/>.
- [10] Andrés Felipe Gómez, Jean Pierre Charalambos, and Andrés Colubri. «ShaderBase: A Processing Tool for Shaders in Computational Arts and Design». In: *VISIGRAPP*. 2016.

- [11] Andrew Howlett, Simon Colton, and Cameron Browne. «Evolving pixel shaders for the prototype video game Subversion». English. In: *Proceedings of the 3rd International Symposium on AI and Games - A Symposium at the AISB 2010 Convention*. Proceedings of the 3rd International Symposium on AI and Games - A Symposium at the AISB 2010 Convention. AISB Symposium on AI and Games 2010 ; Conference date: 29-03-2010 Through 01-04-2010. Dec. 2010, pp. 41–46. ISBN: 1902956907.
- [12] Pol Jeremias and Inigo Quilez. «Shadertoy: live coding for reactive shaders». In: *International Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 2013, Anaheim, CA, USA, July 21-25, 2013, Computer Animation Festival*. ACM, 2013, 103:1. DOI: 10.1145/2503541.2503644. URL: <https://doi.org/10.1145/2503541.2503644>.
- [13] *NVIDIA Shader Library Website*. URL: https://developer.download.nvidia.com/shaderlibrary/webpages/shader_library.html.
- [14] Saket Kumar Pathak. «3D graphics hardware and role of shaders». In: *Confluence 2013: The Next Generation Information Technology Summit (4th International Conference)*. 2013, pp. 351–356. DOI: 10.1049/cp.2013.2340.
- [15] *PShader*. URL: <https://processing.org/reference/PShader.html>.
- [16] Juan C. Quiroz and Sergiu M. Dascalu. «Design and implementation of a procedural content generation web application for vertex shaders». English. In: *25th International Conference on Software Engineering and Data Engineering, SEDE 2016*. Ed. by Frederick C. Harris Jr., Yan Shi, and Sergiu Dascalu. 25th International Conference on Software Engineering and Data Engineering, SEDE 2016 ; Conference date: 26-09-2016 Through 28-09-2016. The International Society for Computers and Their Applications - ISCA, Jan. 2016, pp. 97–102. ISBN: 9781510828971.
- [17] Casey Reas and Ben Fry. *Processing: A programming handbook for visual designers and artists*. en. 2nd ed. London, England: MIT Press, 2014.
- [18] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (4th Edition)*. Pearson, 2020. ISBN: 9780134610993. URL: <http://aima.cs.berkeley.edu/>.

- [19] *Scriptable Render Pipeline*. URL: <https://docs.unity3d.com/2018.4/Documentation/Manual/ScriptableRenderPipeline.html>.
- [20] *Shader Forge*. URL: <https://acegikmo.com/shaderforge/>.
- [21] *Shader Graph*. URL: <https://docs.unity3d.com/Packages/com.unity.shadergraph@10.7/manual/index.html>.
- [22] *Shader Weaver*. URL: <https://www.shaderweaver.com/>.
- [23] *Shaders in Unity*. URL: <https://docs.unity3d.com/Manual/SL-ShadingLanguage.html>.
- [24] *Unity*. URL: <https://docs.unity3d.com/Manual/index.html>.
- [25] *Unity Render Pipelines*. URL: <https://docs.unity3d.com/Manual/render-pipelines-overview.html>.
- [26] *Unity3D*. URL: <https://unity.com/>.
- [27] *Vertex Shader Art, Online*. URL: <https://www.vertexshaderart.com/>.