An Interactive Shader Editor Made for Programmers and Artists

Master's Project Thesis

Joshua Bailey
MSc Computer Animation and Visual Effects
23rd August 2021

Abstract

This project documents the process of creating a Ray Marching tool, accessible to both programmers and artists, used to create 3D fragment shaders. The tool aims to provide a sandbox-like environment to shader development, be inclusive of a range of abilities and have the capability to be used as a scaffolding tool within a pedagogical environment. The resultant tool successfully provides users with a choice between a code or node editor, with the former relying on previous graphics programming experience and the later not. Further development of the tool would require an increased complexity and variety of nodes available within the node editor, to create more elaborate shader programs.

Contents

1 Introduction

'Creative Coding' can be described as a genre of programming with the goal of developing an artefact that is expressive, as opposed to strictly functional (University of the Arts London 2019). Resultant artefacts could be a product of a keen hobbyist entering competitions such as: The Shader Showdown (Revision 2021) and Digital Art Awards (Art Limited 2021) or, serve a greater purpose within advertising and entertainment industries (PBS 2013). Creative Coding can be used as a crutch for programmers and mathematicians alike, to help bridge the gap of applying a technical mind to an artistic endeavour. What differentiates algorithmically generated art from other digital types, is its proceduralism and the ability to automate processes quickly and easily, which would otherwise need to be adjusted by hand. The visualisation of artefacts and the hardware they are developed for can change from piece-to-piece, with some being programmed for the CPU, and others the GPU (specifically taking advantage of vertex and fragment shader pipelines).

Unfortunately, there is an obvious prerequisite to Creative Coding which relies on one's ability to, at the very least, have programming experience and a relative interest in the application of mathematics. This alone creates a barrier to entry that can be inherently too difficult for artists to overcome and thus discourage them from experimenting in the generation of algorithmic art.

The nature of this project is to design and implement an interactive shader editor that is equally accessible to both artists and programmers. Just as Scratch (2021) employs a code blocks editor as a scaffolding tool to "create stories, games and animations" (Scratch 2021), this project will rely on a node-based system to provide artists with a way of generating algorithmic art without any prior graphics programming knowledge or experience. The project will serve as an introduction to shader development, specifically built with the intention of utilising the fragment shader pipeline, in conjunction with Ray Marching (also known as Sphere Tracing) algorithms, to create 3D shaders using Signed Distance Functions (SDFs). The scope of the project will initially be limited to an OpenGL graphics context and the GLSL shader language. Future iterations of the project may then feature the ability to context switch between DirectX, Vulkan, Metal and their respective shader code languages.

2 Previous Work

2.1 Shadertoy

Shadertoy (2013) is an example of a simple web-based tool that allows users to create fragment shaders and visualise the output displayed to a quad. It serves as a platform for shader artists to network, teach, and learn, providing access to a huge repository of community driven content. The tool simply consists of a code editor and visualiser window (Figure 1), granting its users with a handful of shader inputs, textures, sounds, and videos that are accessible to use within their shaders. Despite the tools relatively limited functionality, co-developer Inigo Quilez, makes no hesitation to demonstrate the power of the tool and what can be visually achieved through the application of Ray Marching and mathematics alone (Figure 1).

Figure 1: "Snail" Fragment Shader by Inigo Quilez (Shadertoy 2015).

Alternatively, tools such as VertexShaderArt (2015) also exist on the web, providing similar functionality but instead using vertex shaders. Unfortunately, these tools are by far the least accessible to artists with limited graphics programming experience. Both require an extensive knowledge of the respective vertex and fragment shader pipelines and shader code syntax.

2.2 hg_sdf

hg_sdf (2015) is a library that has been developed with the purpose of helping to write "code that is reusable and that somebody else can actually understand" (Mercury 2021). The library provides a bundle of functions to create a wide variety of primitive geometry; space-folding operators to simulate infinity; and Boolean operators to combine, smooth and bevel objects. In as little as a few lines of code, SDFs like those seen in Figure 2 can be created. hg_sdf (2015) can help free up a user's time by allowing them to start developing shader art at a quicker pace, without first having to worry about implementing a large amount of commonly used functions.

Despite the library offering this solution, it does still require the user to implement a Ray Marching algorithm of their own accord, like Shadertoy (2015). Consequently, the library would be better suited acting as a crutch for user's that lack the confidence or graphics

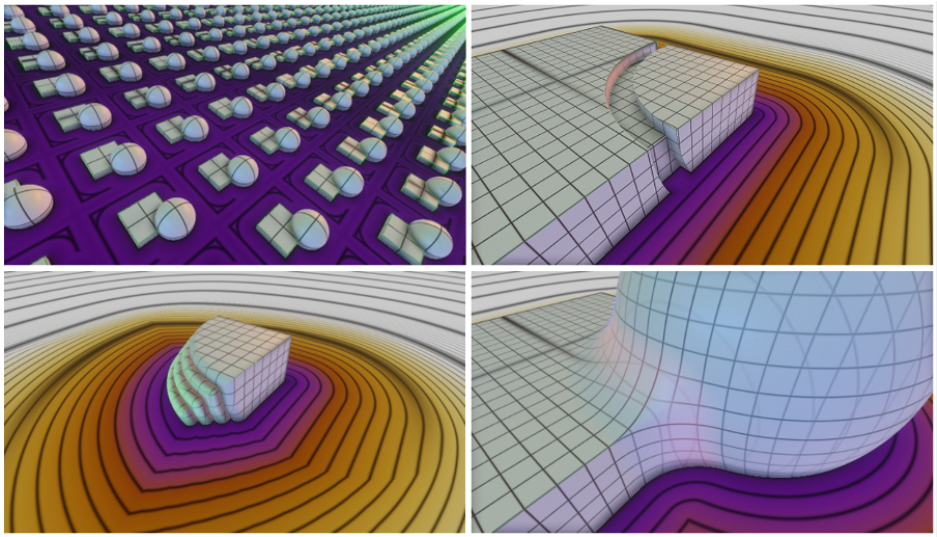

Figure 2: Examples of SDFs created using hg_sdf library functions (Mercury 2021).

programming experience to use a tool like Shadertoy (2015), in isolation. However, the library still may not be the perfect alternative for a non-programmer.

2.3 Learning Graphics Programming

When creating a tool with the intention of being inclusive of artists and being used as a learning resource, it is first important to identify how graphics programming has been successfully taught in the past. Talton and Fitzpatrick (2007) explored two of their existing pedagogical methods of teaching the OpenGL Shading Language (GLSL). The two methods they analysed were a top-down approach in which students used "popular graphics APIs (such as OpenGL) to incrementally build complex graphical applications" (p. 259), and a bottom-up approach in which students created simple applications to demonstrate each stage of the graphics pipeline. Upon reflection of the students, both methods were concluded to produce unsatisfactory results but saw a blended approach produce varying degrees of success. Some students specifically expressed their frustration with struggling to setup an appropriate graphics environment, prior to beginning to tackle their graphics programming assignment. This observation would highlight an additional barrier to entry for someone who just wants to create shaders but is not interested in having to set up a graphics environment. Having a tool that works straight out-of-the-box, without requiring any additional setup from the user, would be more appropriate. Another important factor that reflected the quality of learning for the students was their course length. Longer courses had the ability to fully explore specific topics and algorithms, in great depth, without rushing key concepts like the shorter alternatives. It is apparent that teaching the graphics pipeline is not a trivial task without the correct pedagogical scaffolding and sufficient time. It may be that a higher-level approach to learning would produce better results for beginners, especially under time constraints. An appropriate example would be Goetz et al.'s (2004) flexible XML-based Visual Shading Language (Figure 3).

2.4 Node-based Editors

Goetz et al.'s (2004) system took a higher level, more abstracted approach to shader development. The system consisted of two main features, "blocks delivered by the system with constant functionality" (p. 89) and the ability to develop new blocks that could be compiled and integrated into the pre-existing system. The former provided a user with a way to connect different blocks together to generate highly reliable shaders, as the generation of incorrect syntax was not possible. The latter would allow a user to extend the existing system in a way to specifically suit their own needs. This kind of system would allow a user with little knowledge of vertex and fragment shaders "to achieve good results" (p. 97).

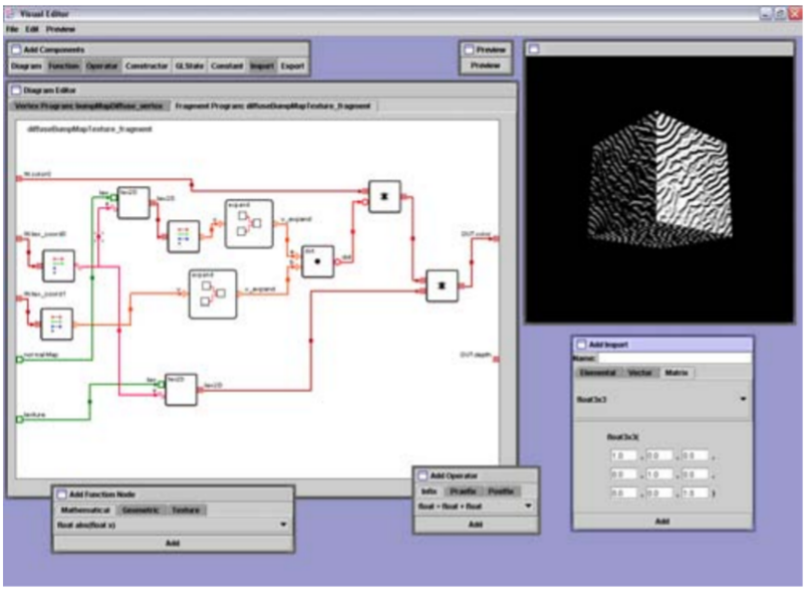

Figure 3: XML-based Visual Shading Language preview (Goetz et al. 2004, p. 89).

Jensen et al. (2007) also discuss the relative difficulty of shader programming and the benefits posed from using node-based tools instead. "Shaders are authored by connecting nodes of individual functionality with one or mode edges, building a tree structure that represents the shader" (p. 89). This structure can then be evaluated and compiled into shader code (Figure 4). These types of editors provide a user-friendly interface that avoids any direct interaction with shader code all together and has been successfully implemented into many popular engines such as Unity (Unity Technologies 2021) and Unreal Engine 4 (Epic Games 2021). Editors of this sort provide a very small barrier to entry and are inclusive of all abilities, hence their widespread success and use within game development and visual effects studios.

Jensen et al. (2007) concentrated their efforts into two distinct features to improve a user's experience with their node editor. Firstly, the ability to encapsulate several nodes into a single grouped node that could be reused within several other graphs. This created an efficient workflow and reduced the repeated need for certain combinations of nodes, much like a function or method within programming. Secondly, preview functionality of each node's effect on the final shader was very good in helping a user identify exactly how each node

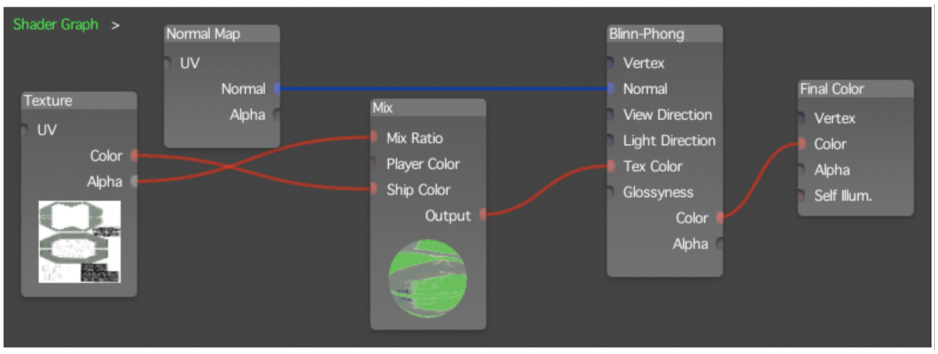

Figure 4: Example shader tree structure (Jensen et al. 2021, p. 89).

was influencing the result. It also served as a useful debugging tool to ensure nodes were behaving how they were intended to.

Creating a tool that is inclusive of all abilities will evidently require a combination of all previously discussed pieces of work. The requirements of a programmer are as straightforward as needing a code editor and, in some cases, access to an external library such as hg_sdf (2015). On the other hand, an artist will need to be accommodated with an intuitive node-based editor, in which more of the tool's emphasis will need to be placed on. A valuable extension of the preview functionality discussed by Jensen et al. (2007) would include the ability to preview the evaluated code of the node graph in real time. Upon extending the functionality of the node graph, the user should be able to view how the final evaluated code is being affected and identify the code contributed by each of the nodes. This will provide those artists who are determined to learn shader programming with a foundational understanding of how the nodes are exactly translated into shader code.

3 Technical Background

Ray Marching will be the primary rendering technique driving the creation of 3D shaders within this tool. The technique shares many similarities with Ray Tracing, however, is characterised by Quilez and Jeremias (2014) to provide "a much more powerful way to describe 3D shapes and lighting" (p. 9), making it the most popular technique amongst the Shadertoy (2015) community. Firstly, before discussing the details of Ray Marching and how it renders objects within a scene, it is important to identify how these objects are represented.

3.1 Object Representation

Two ways that an object can be mathematically represented are parametrically and implicitly. The former is particularly good at generating polygonal surfaces to approximate a mathematical surface, whereas the latter works very well when used in conjunction with ray tracing (Clemson 2014) to identify the relationship between the surface and a point in space. SDFs are used to describe the latter relationship, by finding the shortest distance between a point in space and the surface of an object. The value returned from an SDF can be used to determine whether the point is inside, outside or on the surface of the object, depending on its sign being positive or negative (hence Signed Distance Function). Wong (2016) provides the following example, considering a sphere centred at the origin.

"Points inside the sphere will have a distance from the origin less than the radius, points on the sphere will have distance equal to the radius, and points outside the sphere will have distances greater than the radius." – Wong (2016)

A unit sphere can be implicitly described by the polynomial function:

$$x^2 + y^2 + z^2 - 1 = 0$$

Which represented as an SDF is:

$$f(x, y, z) = \sqrt{x^2 + y^2 + z^2 - 1}$$

Resulting in three total possibilities:

$f(x, y, z) = 0$	On the surface
$f(x, y, z) < 0$	Inside the surface
$f(x, y, z) > 0$	Outside the surface

3.2 Ray Marching

Now that it is understood how objects are represented within a scene, it is possible to begin explaining how Ray Marching is used to render those objects to the screen. There are three distinct stages to any Ray Tracing or Ray Marching algorithm (Figure 5):

Stage 1 - A *Camera* is defined within a scene and used as a point of origin to fire *View Rays* through an *Image* plane.

Stage 2 - Each *View Ray* is sampled by a function that calculates the point of intersection between itself and a *Scene Object*, returning the distance between the *Camera* (or ray origin) and intersection point.

Stage 3 - Lighting and material calculations can then be calculated at the point of intersection, returning a new value representing the colour of the pixel on the *Image* plane that the *View Ray* was fired through.

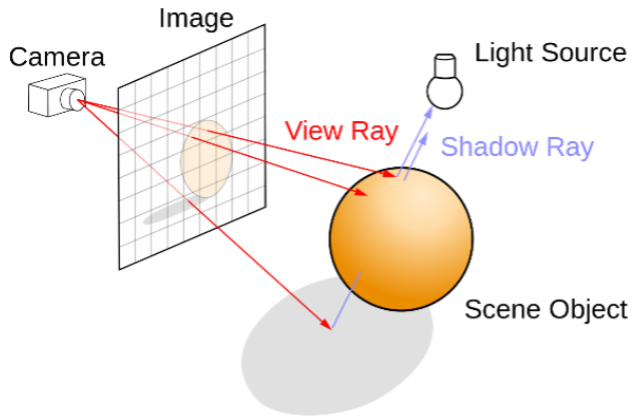

Figure 5: Ray Tracing / Ray Marching scene setup (Wikipedia 2021).

Where the two algorithms diverge is during *Stage 2*. Finding the point of intersection of basic primitive objects (cubes, spheres etc.) can be achieved with relative ease by using a ray-intersection equation for the specific shape. Explanations and implementations on how ray-intersection algorithms are calculated when creating a basic Ray Tracer are discussed by Scratchapixel 2.0 (2021). However, objects of far greater complexity require the following method, due to their potential irregularity and difficulty in finding their bounds (The Art of Code 2018).

In Figure 6, point p_0 represents a camera's position (or ray origin), the blue line represents the ray being fired, and the black line represents the scene object. When beginning to calculate the point of intersection, the distance between p_0 and the scene object must be calculated first (represented by the green circle centred around p_0). This is the currently known furthest distance the ray can be marched down, to point p_1, without intersecting into the scene object. This marching process is then repeated a fixed number of times until either the distance between the ray and scene object is smaller than a defined threshold (point p_4), or this threshold is not met within a fixed number of steps, denoting the ray missing the scene object entirely and heading towards infinity.

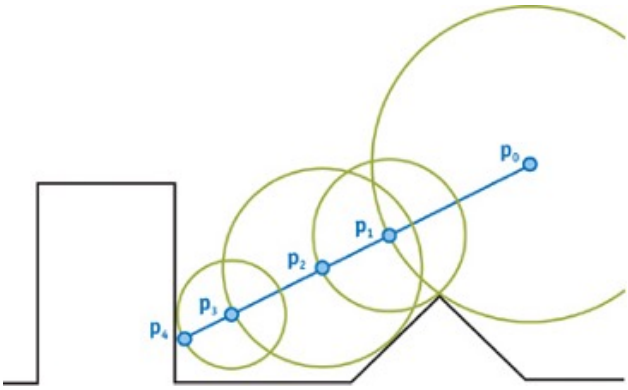

Figure 6: Ray Marching / Sphere Tracing (NVIDIA 2005).

3.3 Lighting

The result of this Ray Marching algorithm will return a distance field that can be passed into a lighting model function to generate a shaded image. A simple lighting model to implement, described by The Art of Code (2018) in Figure 7, is one that is only concerned with the angle of the light rays hitting the surface, to determine its brightness. Light rays pointing directly perpendicular to the surface will have a maximum brightness value of one, whereas light rays of an increasing angle with taper down to a minimum brightness value of zero, when the light rays are completely parallel to the surface. This is calculated as:

$$\vec{lightDirection} \cdot \vec{normalDirection}$$

With the following prerequisite:

$$lightDirection = \|(lightPosition - surfacePosition)\|$$

And, the value of the *normalDirection* being determined as a product of sampling three surrounding points in x, y, z directions and subtracting them from the point of intersection.

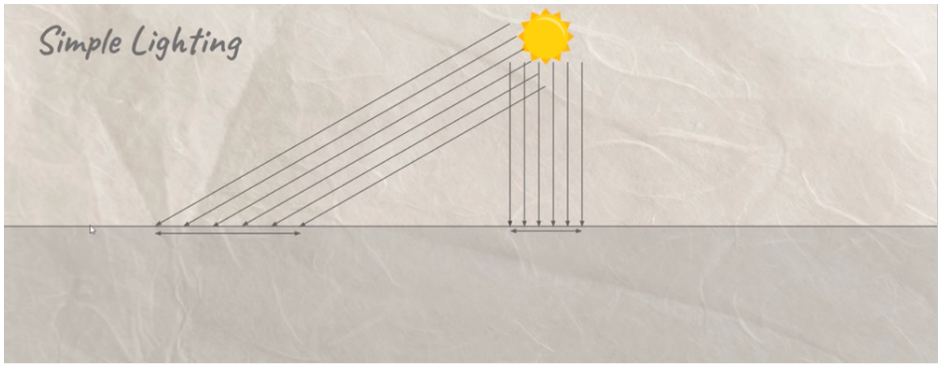

Figure 7: Simple Lighting Model (The Art of Code 2018).

4 Implementation

OpenGL was the selected graphics API due to its cross-platform compatibility and well-established history and support from Khronos Group (2021). As opposed to programming in raw OpenGL, the NCCA Graphics Library (Macey 2021) was used, due to its prebuilt methods that wrap-up most of OpenGL's functionality for ease of use. The Qt API (Qt Project 2021) was also selected to encapsulate the entire application and handle UI functionality.

4.1 Project Objectives

The three primary objectives of this project, later to be used as success criteria to evaluate the project's overall completion, are as follows:

4.1.1 Objective 1 – Shader Development

Firstly, this tool should aim to provide users with a sandbox-like environment where they can quickly and easily develop fully fledged shader programs, or prototype specific shader algorithms/features, without the hassle of having to create an OpenGL context from scratch. This tool should also consider seamless importation and exportation of content to itself and third-party applications.

4.1.2 Objective 2 - Accessibility and Inclusivity

Secondly, this tool should aim to be accessible to both programmers and artists, inclusive of a spectrum of abilities. Users should be provided with an abstracted tool for them to create shader programs without the need to write code. Alternatively, users who wish to write their own code should also be accommodated appropriately.

4.1.3 Objective 3 - Teaching and Learning

Thirdly, this tool should aim to provide the ability to be used within a pedagogical environment, to be used as a scaffolding tool for those who wish to learn GLSL and to create shader programs.

4.1.4 Considerations

Finally, it is worth noting that this tool will very much be the product of the "make it work, make it fast, make it pretty" philosophy. Despite the user interface and experience (UI/UX) of tools being just as important as its functionality and efficiency, it will not be considered a primary focus of the project at this time. Likewise, the project's realistic expected outcome is not to be a fully-fledged deployable tool but serve as a proof of concept that will later evolve.

4.2 Core Functionality

The core functionality of this tool primarily takes advantage of the fragment shader pipeline and alternates between two separate shader programs (Figure 8). Shader A begins as the currently compiled shader program projected to a quad, whilst shader B begins as the shader program the user is currently creating or editing. If shader B is successfully compiled by the user, the shaders swap over thus now projecting shader B onto the quad and the user now editing shader A. This process will then repeat. Otherwise, upon unsuccessful compilation, the shader programs do not swap, and the user is fed back an appropriate compilation error. The tool is divided into two distinct sections that provide entirely unique features depending on which one is selected for use. The user can appropriately decide between one of the two following sections, based upon their needs and ability:

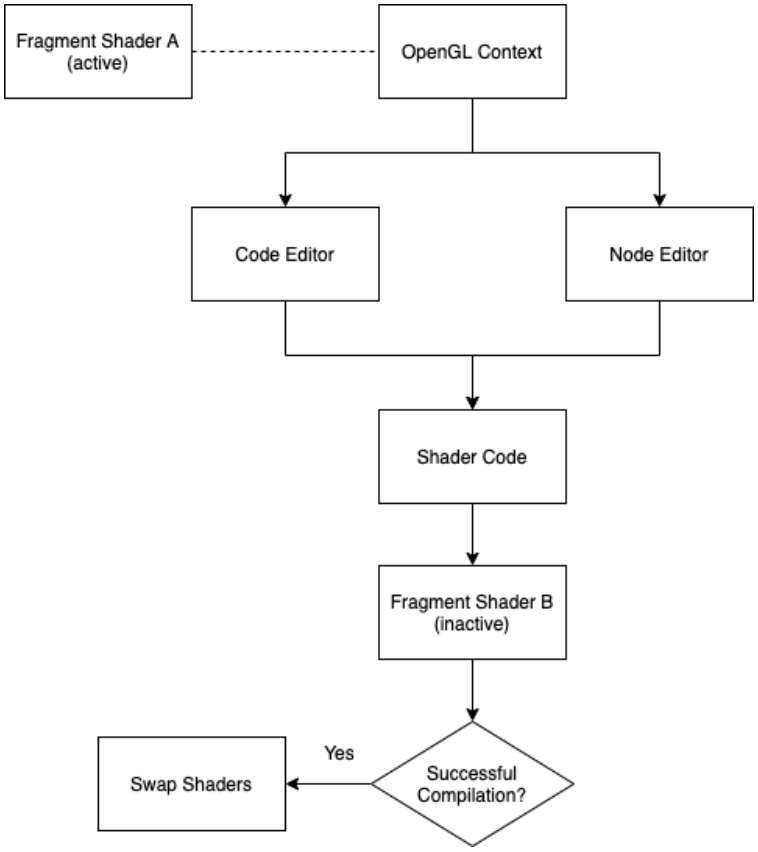

Figure 8: Basic program overview (personal collection).

4.2.1 Code Editor

The Code Editor (Figure 9) is an all-access tool designed specifically for users that wish to have direct access to the fragment shader and all its functionality. Users have no restrictions on what they can create and implement other than being provided with a predefined list of shader uniforms to access and use. It is the ultimate sandbox tool for any keen shader artist looking to develop 2D or 3D shaders using the fragment shader pipeline. The Code Editor

also offers basic built-in syntax highlighting of keywords including the complete contents of the OpenGL Shading Language (GLSL) v4.10 specification. GLSL v4.10 was specifically chosen due to its recent deprecation on Macintosh platforms (PC Gamer 2018), making it the most recent cross-platform compatible version. Due to the intended purpose of the tool being used to implement Ray Marching algorithms to generate 3D shaders, users also have on hand access to the hg_sdf (2015) library, providing them with an exhaustive list of commonly used functions to speed up their development process. Due to the expectation of users implementing their own algorithms, they will not be provided with an example Ray Marching algorithm upon start-up. An example Ray Marching algorithm, and scene, can however be accessed from the examples folder provided with the tool. Other smaller features include the ability to pause compilation of the current shader to save processing power for users with limited hardware. Font size adjustment was also implemented in consideration of the tool being used to demonstrate and teach graphics programming to a large audience.

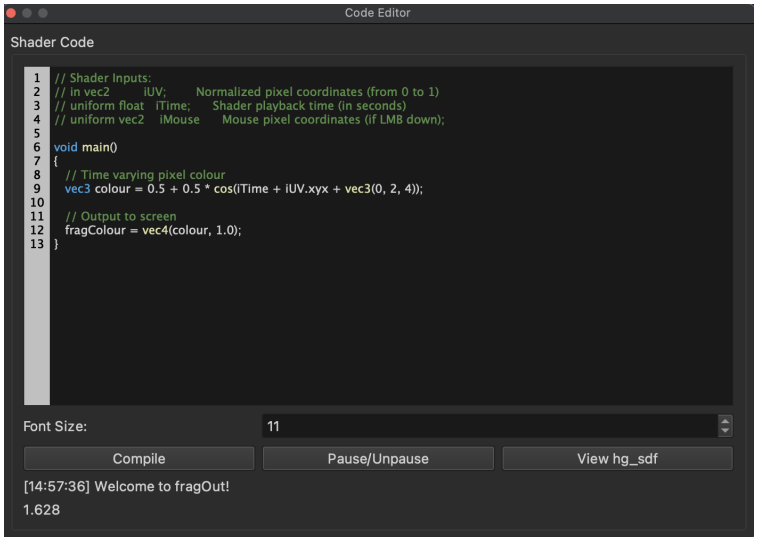

Figure 9: Code Editor (personal collection).

4.2.2 Node Editor

The Node Editor (Figure 10) is an abstracted higher-level tool when compared to its code editor counterpart, built upon a node-based library authored by *paceholder* (2021). The Node Editor is designed specifically for users with no shader programming experience or for those users who would much prefer to create shader programs using a node-based system. The Node Editor provides the benefit of a user being able to quickly piece together, a working 3D shader that would otherwise take more than 100 lines of code implement. It too also provides a guarantee of compiling successfully, assuming the user has appropriately created a node graph producing no validation errors. As it currently stands, the node editor can only take advantage of a pre-implemented Ray Marching algorithm to compile 3D shaders, and therefore lacks much of the fidelity that the code editor has to offer.

Nevertheless, the node editor does in fact offer a significant amount of complexity and creative potential through utilising the following available nodes:

Ray Marching - The Ray Marching node is the final node of every graph, encapsulating the pre-defined Ray Marching algorithm and lighting calculations of the SDFs that are received as inputs.

SDFs - These nodes encapsulate the respective signed distance function of one of the following shapes: Box, Capsule, Cylinder, Infinite Plane, Sphere and Torus.

Materials - Material nodes are used to pass RGB colour data to an SDF to influence a shapes colour.

Operators - The Boolean operator node offer access to: Intersection, Union, and Difference functions, that can be used to create complex geometric shapes by combining SDFs (Figure 11). The technique is known as Constructive Solid Geometry (Pixar 2021).

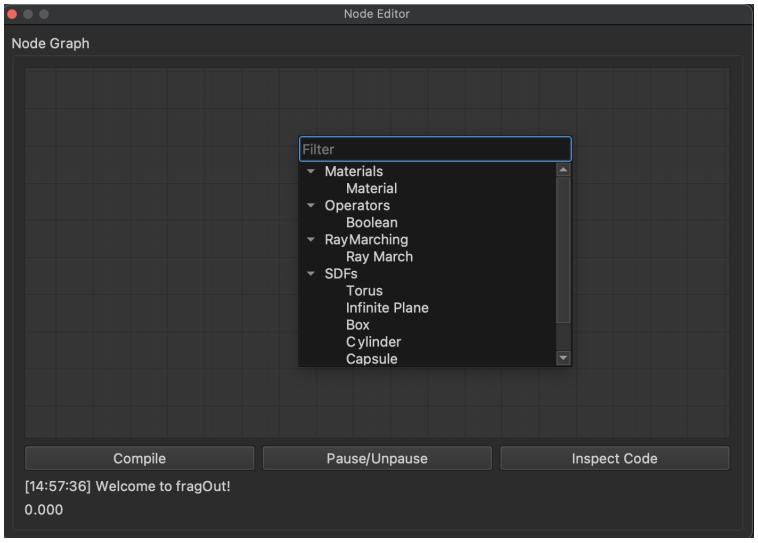

Figure 10: Node Editor (personal collection).

Each node contains some level of customisation by offering several unique editable parameters such as the ability to edit an object's position or size. Each node also provides the functionality of inspecting its code for users interested in understanding what each node is doing under the hood.

During construction of a node graph, each node's data is passed to its connected node(s), as an input, and relevantly stored to be used at the current (or later) stage. Once a valid node graph has been constructed, the Ray Marching node will contain the final evaluated code as a product of itself and all previously connected nodes in the graph. Upon clicking 'Compile', this shader code will then be passed to the relevant shader program and attempt compilation. Successful compilation will then result in this shader code being passed to the fragment shader and displayed on screen.

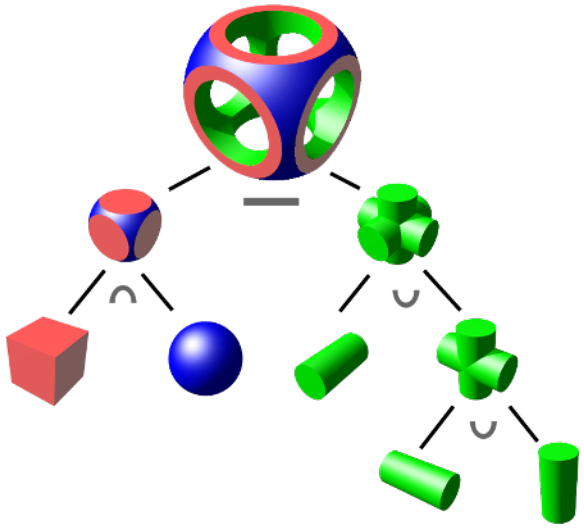

Figure 11: Example of Constructive Solid Geometry (Wikipedia 2021).

4.3 Saving / Loading

Both editors handle the saving and loading of files. The code editor enables users to simply save out their GLSL shaders to be used by third party applications, as well as loading in external GLSL files. The node editor offers the same saving and loading functionality however, the data is stored to a unique *.flow* file type, provided on behalf of the node-based library by *paceholder* (2021). The file format is saved in a JSON style, storing node information such as position, connections, and parameter data. Not all node and shader code data is saved to the *.flow* file, only that of which cannot be generated from reimportation of the node graph.

4.4 Examples

The following example scenes have been created using the Node Editor to demonstrate the functionality of specific nodes:

4.4.1 RGB Spheres

RGB Spheres is a simple scene (Figure 12) showcasing multiple SDF spheres that have been rendered using their respective red, green, and blue materials. The node graph seen in Appendix A.1 shows how this scene was created.

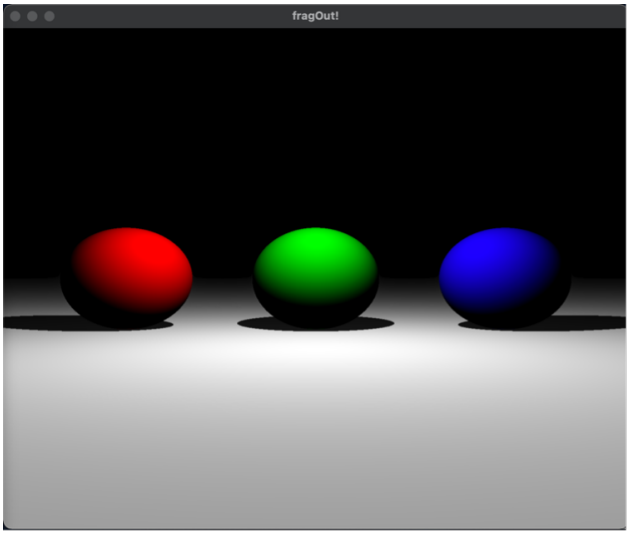

Figure 12: RGB Spheres scene (personal collection).

4.4.2 Constructive Solid Geometry (CSG)

Constructive Solid Geometry (CSG) is a scene (Figure 13) inspired by Figure 11. It combines all three Boolean Operator functions (Intersection, Union, and Difference) to create a curved cubic object, with holes punched through each of its six faces. The node graph seen in Appendix A.2 shows how this scene was created.

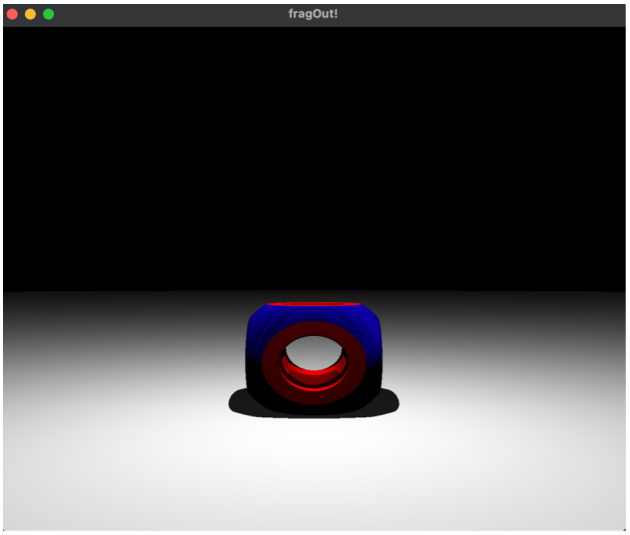

Figure 13: Constructive Solid Geometry scene (personal collection).

To generate the two previous examples via the Code Editor, a user would be expected to write about 180 lines of code from scratch. This code can be inspected upon successful compilation by clicking the "Inspect Code" button in the Node Editor. Even though the

Node Editor is specifically designed for artists, it can always also be used by programmers as a quick prototyping tool, to get a basic scene up and running.

4.5 Node Validation

Feedback to the user is provided by the Node Editor in the form of validation messages displayed on the nodes. There are two types (Figure 14); validation errors, which are highlighted in red and will result in a compilation failure; and validation warnings, which are highlighted in yellow to remind the user of something they may have forgotten to do.

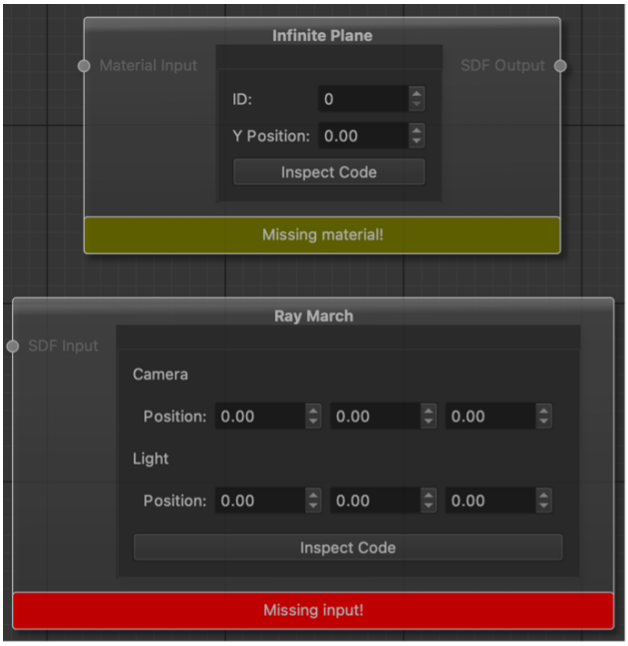

Figure 14: Node validation messages (personal collection).

4.6 Problems Encountered

A problem that was experienced exclusively with the Node Editor, when evaluating the node graph into shader code, was the initialisation of unique variable names to store data. Originally, when an SDF node was created, it would be initialised with a variable name of the shape, for example 'sphere'. However, if two SDF sphere nodes were created then the initialisation of a variable named 'sphere' would happen twice and thus cause compilation errors. One solution to this problem would be to simply generate a random ID for each node and append it to the variable name, so no two variables were initialised with the same name. Unfortunately, despite this solution working, it would end up bloating the code with strings of random numbers that a user inspecting the code may find inconvenient. It would also be very difficult for a user to identify what SDF the variable name represents within the node graph, when inspecting the final compiled code. An alternative solution that was implemented allowed the user to assign each SDF node with an ID, displayed as a parameter on the node (Figure 14). This ID could then be used and appended to the end

of a variable name as a unique identifier, meaning that upon code inspection the user could quickly identify the corresponding node and variable by matching their name and ID. A very similar process is also carried out for Boolean Operator nodes, when accepting two Boolean Operators as inputs, as opposed to SDF nodes.

Another problem encountered was identifying an appropriate area to place each nodes editable parameters. Node Editors used in popular game engines and DCC (Digital Content Creation) tools have significantly different user interfaces. Some display a node's editable parameters in a separate window, some within the node itself, or using a blend of both. For this project, the decision to display the editable parameters within the nodes themselves was made. The primary reason for this decision was due to each node only containing a small number of parameters, meaning the nodes wouldn't become too big and consume much space within the node graph. A positive of having all node parameters on display at once, as opposed to selecting a node to view its parameters in a separate window, is that a user can very quickly compare the parameters of many nodes at once. When tweaking the values of many nodes on the fly, having to constantly select between many different nodes, may become frustrating for the user. Unfortunately, the downside of having a nodes parameters embedded within the node itself is that when a node graph becomes particularly large, the user may constantly find themselves to be zooming in and out of the graph to inspect and change the parameters. Alternatively, if the parameters were displayed in a separate window, this wouldn't be the case. Ultimately, whatever user interface is adopted is going to result in an inconvenience for the user at some point down the line.

5 Conclusion

As a reflection of the project's objectives, overall, the outcome has been a major success. Firstly, the tool provides users with a sandbox-like environment to develop fully fledged shader programs quickly and easily, or prototype specific algorithms/features. This can be fully achieved via the Code Editor, and to a lesser degree via the Node Editor. Both editors also handle the seamless importation and exportation of their respective *.glsl* and *.flow* type file formats. Secondly, this tool is accessible to use for both artists and programmers and is inclusive of many abilities in-between, such that users can create shaders without writing a single line of code or by coding them from scratch. This tool also offers support to those currently learning graphics programming by having access to the hg_sdf (2015) library. Furthermore, users are also provided with a way to inspect the code of nodes, to gain a deeper understanding of their functionality. These features would also be suitable to be used within an academic environment, as a scaffolding tool, for those learning GLSL and/or about shading programming in general. Students and hobbyists can utilise the out-of-the-box functionality and dive right into shader programming with no barriers to entry.

To improve the tool, further developments and progress to the Node Editor need to be made. As it currently stands, the Node Editor provides users with a simple framework to create basic 3D shaders, however, it requires much more functionality to be used as an appropriate tool for creating complex shader art. Some additional features would include:

Space Folding Operators - Space folding operators would provide a user with the functionality of simulating infinite space or an infinite number of objects within a scene. This type of operator would be useful for creating 3D fractals and Mandelbulbs.

Smoothing Operators - Smoothing operators would provide a user with the functionality of seamlessly blending shapes together (Figure 2). This type of operator would provide a user with a more intricate toolset when modelling objects with multiple SDFs and Boolean operators.

Group Nodes - Group nodes would be particularly useful at encapsulating large parts of the node graph and collapsing them into a single node, not only to save space but to also make a user's workflow more efficient. Users would not be required to have to repeat the creation of certain parts of their network.

Additional Lighting Models - Implementing lighting models of greater complexity, such as a Phong Reflection Model (Phong 1975), would improve the overall quality of SDF rendering. If the tool were to offer ambient, specular, and diffuse parameters to an object's material, it would increase a scenes realism and thus allow the simulated rendering of many different types of objects, such as metals and plastics.

Further development of the project could then be tailored towards integrating access to shader uniforms, to allow the creation of animated shaders within the Node Editor. Moreover, taking advantage of other graphics APIs such as DirectX, Vulkan and Metal, as similarly featured during a previous Qt World Summit (YouTube 2019), would open the tool

up to a larger audience. However, to make a firm decision as to what direction the project must next take, would require receiving first-hand feedback from a control group of users testing the tool. This would remove all speculation as to what a user 'may' want and would directly identify what they 'do' want.

References

art limited, 2021. *Digital Art Awards 2021 Competition* [online]. Available from: https://www.artlimited.net/competition/digital-art-awards-2021/en/79 [Accessed 20 July 2021].

Clemson University, 2014. *Computer Graphics Course Notes – Implicit and Parametric Surfaces* [online]. Available from: https://people.cs.clemson.edu/~dhouse/courses/405/notes/implicit-parametric.pdf [Accessed 20 July 2021].

Epic Games, 2021. *Unreal Engine* [online]. Available from: https://www.unrealengine.com/en-US/ [Accessed 20 July 2021].

Goetz, F., Borau, R. and Domik, G., 2004. An XML-based visual shading language for vertex and fragment shaders. *Proceedings of the Ninth International Conference on 3D Web Technology*. 87-97.

hg_sdf, 2021. *hg_sdf – A glsl slibrary for building signed distance functions* [online]. Available from: https://mercury.sexy/hg_sdf/ [Accessed 20 July 2021].

Jensen, P. D. E., Francis, N., Larsen, B. D. and Christensen, N. J., 2007. Interactive shader development. *Proceedings of the 2007 ACM SIGGRAPH Symposium on Video Games*. 89-95.

Jeremias, P. and Quilez, I., 2014. Shadertoy: learn to create everything in a fragment shader. *SIGGRAPH Asia 2014 Courses*.

Khronos Group, 2021. *Khronos Group* [online]. Available from: https://www.khronos.org/ [Accessed 27 July 2021].

Macey, J., 2021. *NCCA Graphics Library (NGL)* [online]. GitHub. Available from: https://github.com/ncca/ngl [Accessed 13 June 2021].

NVIDIA Corp., 2005. *GPU Gems 2* [diagram]. NVIDIA Corp.

paceholder, 2018. *nodeeditor* [online]. GitHub. Available from: https://github.com/paceholder/nodeeditor [Accessed 13 June 2021].

PBS, 2013. *The Art of Creative Coding* [video, online]. PBS. Available from: https://www.pbs.org/video/-book-art-creative-coding/ [Accessed 20 July 2021].

PC Gamer, 2018. *Developers fear for Mac gaming as Apple deprecates OpenGL support* [online]. Available from: https://www.pcgamer.com/developers-fear-for-mac-gaming-as-apple-deprecates-opengl-support/ [Accessed 2 August 2021].

Phong, B. T., 1975. Illumination for computer generated pictures. *Communications of the ACM*. 18 (6), 311-317.

Pixar, 2021. *RenderMan - Constructive Solid Geometry (CSG)* [online]. Available from: `https://renderman.pixar.com/resources/RenderMan_20/constructiveSolidGeometry.html` [Accessed 21 August 2021].

Qt, 2020. *Provide a higher level of code with Qt 3D Node Editor and Shader Generator* [video, online]. YouTube. Available from: `https://youtu.be/drUpXH9Z-RE` [Accessed 6 July 2021].

Qt Project, 2021. *Qt API* [online]. Available from: `https://doc.qt.io/` [Accessed 19 August 2021].

Revision, 2021. *Special Events – Shader Showdown – Live Coding Competition* [online]. Available from: `https://2021.revision-party.net/events/special-events` [Accessed 20 July 2021].

Scratch, 2021. *Scratch* [online]. Available from: `https://scratch.mit.edu/` [Accessed 20 July 2021].

Scratchapixel 2.0, 2021. *A Minimal Ray-Tracer: Rendering Simple Shapes (Sphere, Cube, Disk, Plane, etc.)* [online]. Available from: `https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes` [Accessed 19 August 2021].

Shadertoy, 2015. *"Snail"* [online]. Available from: `https://www.shadertoy.com/view/ld3Gz2` [Accessed 20 July 2021].

Shadertoy, 2021. *Shadertoy* [online]. Available from: `https://www.shadertoy.com/` [Accessed 20 July 2021].

Talton, J. O. and Fitzpatric, D., 2007. Teaching graphics with the openGL shading language. *SIGCE Bull.* 259-263.

The Art of Code, 2018. *Ray Marching for Dummies!* [video, online]. YouTube. Available from: `https://youtu.be/PGtv-dBi2wE` [Accessed 23 July 2021].

Unity, 2021. *Unity* [online]. Available from: `https://unity.com/` [Accessed 20 July 2021].

University of the Arts London, 2019. *How to Start Creative Coding* [online]. Available from: `https://www.arts.ac.uk/study-at-ual/short-courses/stories/how-to-start-creative-coding` [Accessed 21 August 2021].

VertexShaderArt, 2021. *VertexShaderArt* [online]. Available from: `https://www.vertexsh`

aderart.com/ [Accessed 20 July 2021].

Wikipedia, 2021. *Constructive Solid Geometry* [diagram]. Available from: https://en.wikipedia.org/wiki/Constructive_solid_geometry [Accessed 12 August 2021].

Wikipedia, 2021. *Ray Tracing (Graphics)* [diagram]. Available from: https://en.wikipedia.org/wiki/Ray_tracing_(graphics) [Accessed 24 July 2021].

A Appendix

A.1 RGB Spheres - Node Graph

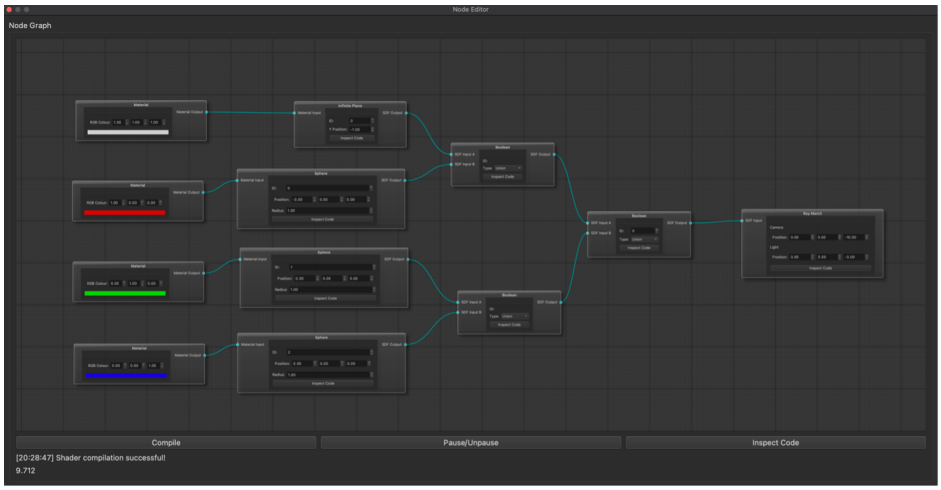

Figure 15: RGB Spheres node graph (personal collection).

A.2 Constructive Solid Geometry (CSG) - Node Graph

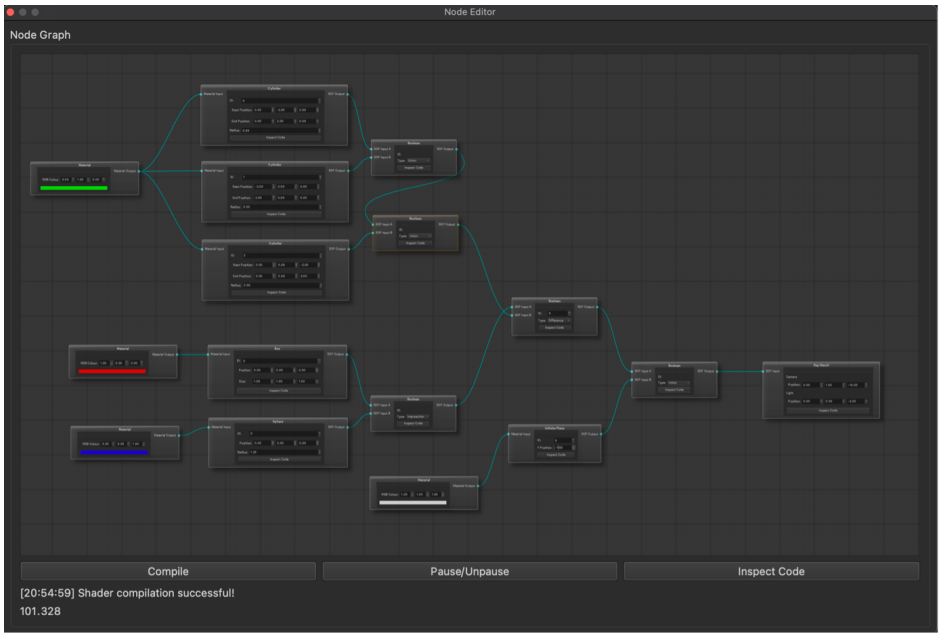

Figure 16: Constructive Solid Geometry node graph (personal collection).