CSC8498

An Exploration of Building Render Pipelines With a Node-Based Editor

Stuart Lewis

School of Computing Science, Newcastle University, UK

Abstract

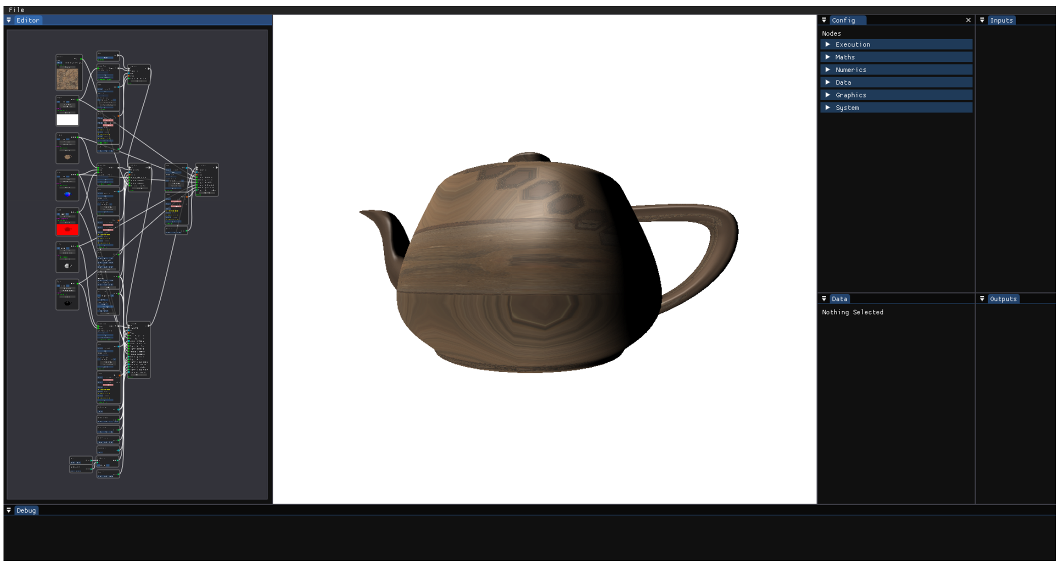
This dissertation presents a tool for building OpenGL rendering pipelines through the use of a node-based editor, similar to the likes of Unreal engine's blueprints and Unity engine's shader graph.

Visual scripting is a growing practice in game development, with engines like Unity[20], Unreal[8], and Godot[5] making prevalent use of it. The tool discussed in this dissertation attempts to take this paradigm of visual scripting, and use it to create a generalised interface for building full render pipelines. The goal with this is to make it easier to create and tweak test areas for different rendering techniques, both for use in learning new techniques and iterating on existing ones.

This dissertation is paired with a software artefact (called GLSandbox). The project is open source and available on GitHub. [a]

[a] Project repository: https://github.com/blacktack2/GLSandbox

Keywords: Node-based editor, graph-based editor, graphics, render pipeline



[1] Email: stuart.manfred.lewis@gmail.com

1 Introduction

1.1 Aims and Objectives

The overall aim of this project is to explore the process of building render pipelines using node-based editors. The objectives are as follows:

- Investigate existing visual scripting tools and methodologies.
- Create a tool which is able to build a simple render pipeline through a visual scripting interface.
- Analyse the performance of the tool by comparing a pipeline it generates with a programmed version of the same pipeline.
- Evaluate potential optimisations and improvements this tool, and others like it, could use.

1.2 GLSandbox

GLSandbox is the artefact being created for this project. The aim is to create a tool which uses a node-based editor (similar to those seen in Unity and Unreal) to build full render pipelines for prototyping, testing, and profiling of various rendering techniques. The minimum specification set for this project's artefact is as follows:

- Must have an application window with a graph-editor panel and a space to render the final pipeline.
- Must be able to build a render pipeline by linking relevant mesh, shader, and uniform nodes to a render-pass node.
- Must be able to handle any user specified shader or mesh.
- Must have enough functionality to allow the creation of a (simple) deferred pipeline.
- Must have a system for handling sub-graphs.
- Must have a method of comparing performance of a graph with an equivalently hard coded pipeline.
- Must include example scenes which can be used to showcase the use and performance of the tool.

A common problem in graphics development is that of creating a suitable test area for new shaders and pipeline structures. In many situations this involves hacking an existing system to work as a test bed (such as creating a dedicated scene in a game), or use a separate tool like shadertoy (which requires modifying the structure of the shader to work with it's system, and then changing it back to fit the actually desired system). The idea with GLSandbox is to provide a tool with which one can build a pipeline for an already existing shader or structure, and then still be able to plug it directly into it's target system, rather than modifying it to fit another system for testing. Instead of editing a shader to use the uniforms, environmental variables, and structure of an external tool, one should be able to quickly build a custom pipeline around the shader, with all the necessary variables and operations available to execute and test it.

Nodes provide a convenient built-in user-interface which can be dragged and organised however the user likes, and used to modify variables like shader uniforms. Node connections can also be quickly swapped around to easily test different structures, orders of operations, or re-order pipelines. This, along-side previews for any textures being used, can provide an easily made test bed for tweaking parameters, comparing methods, and viewing the underlying structure of a pipeline, without having to manually implement input points or display modes. Implementing some form of performance profiling would also be helpful in comparing different methods, such as for attempting to find a more efficient algorithm for a shader.

One problem with node-based editors is the sheer amount of visual noise resulting from large and complex graphs, hence the added requirement of sub-graphs to mitigate this. Other utilities for organising graphs could include: comment nodes, which can be used to wrap logical blocks together; tunnel nodes, which can reduce the mess created by many long distance connections; and collapsing nodes which allow logical blocks to be hidden (rather than relying solely on sub-graphs).

2 Background

2.1 Graphics Pipelines

Render pipelines (or graphics pipelines) are models which describe the operations a graphics system requires to render a 3D scene to a 2D screen[10] (usually making use of the Graphics Processing Unit (GPU)). As a greatly simplified overview, the process generally involves taking 3D mesh data, applying transformations to it (for correct scale and positioning), rasterizing the mesh data into pixels, then finally calculating the colours to display to these pixels.

GPUs have been around since the early 1980s[15], where they still relied heavily on the CPU and could only draw wire-frames. The first CPU independent GPU was released 1984, and was capable of drawing and colouring filled polygons. Modern GPUs are now capable of handling complex, real-time scenes and make use of advanced techniques to imitate real world aesthetics. With more powerful GPUs, more complex and advanced techniques become feasible such as NVIDIAs recent Deep Learning Super Sampling (DLSS) technology, which uses a deep learning AI to interpolate between frames[3]; allowing the expensive render pipeline to spend more than one frame generating (or simply halving the frame time).

A modern render pipeline usually consists of one or more sections (passes) and two main sub-sections which a developer will work with: the vertex shader, where mesh geometry can be manipulated on a per-vertex level; and the fragment shader where the colour of individual pixels is decided. There also exist additional shader types such as geometry and tessellation, as well as completely different structures like compute shaders (for arbitrary data manipulation), though this dissertation will mainly focus on the simplest. Other necessary steps, such as rasterization, are often handled in the background by graphics APIs like OpenGL, and will not the focus of this dissertation.

2.2 Visual Scripting

Visual scripting is a method of programming which seeks to make development easier and more intuitive through a visual interface, as an alternative to plain text. Many visual scripting programs make the claim of being able to build their respective system, be it a game, shader, or procedural generation algorithm, without the need to write any code. Visual scripting is most well known from tools such as Unreal engine's blueprints, Unity engine's shader graphs (as well as many user created assets), and the Scratch visual scripting language.

Usage of visual scripting can be seen as far back as the 1970s with tools like Pygmalion[18] and GRAIL[4]; early experiments into the idea of visual computer interfaces. While it was initially a fairly niche tool, visual scripting has become more prevalent over time (particularly for applications focused on a visual output like games and websites). In more recent years, UI (User Interface) design tools (such as Wix and SquareSpace used for websites) have taken to using visual interfaces to provide a more intuitive method of designing websites or applications. The advantage of this type of visual script is clear, in that it provides immediate feedback on the final look of an application, showcases the underlying structure in a more visually intuitive manner, and makes tweaking details significantly easier.

Game development has seen use of visual scripting as early as 1990s with Game Maker[6], and Unreal saw the introduction of it's now renowned blueprint system in 2006, with Unreal engine 3. In general, modern game engines like Unity and Unreal tend to use graph based interfaces with the intent of building a complete programming language capable of any operations other languages could do. This type of interface tends to act as an analog for normal programming, allowing the same logic used in developing standard programs to be applied in a visual manner, rather than completely abstracting the system to only what is needed for it's visual components.

2.3 Modern Examples of Visual Scripting

2.3.1 Unreal Engine (Blueprints)

Unreal engine is a 3D game engine developed by Epic Games[8], beginning in the form of ZZT (1991)[25], a precursor to Unreal engine which used a custom Object-Oriented Programming (OOP) based scripting system. Unreal engine 1 was first showcased to the public in 1998 with the first-person shooter Unreal[2] and has since gone on to become one of the most prevalent game engines, used by industry and hobbyists alike. Unreal also has it's use outside of game development, with utility designed for movie graphics and 3D modelling. The blueprint system used today had it's start in Unreal engine 3 (2006), and is used widely as an alternative, or in conjunction with, standard scripting.

Blueprints are primarily used to establish the functionality of components within a scene. Every object with some form of logical action, including Heads-Up Display (HUD)s, particles, and the scene itself, can have it's own blueprint describing which actions to take under varying circumstances. Using an example from Unreal's official documentation[7], shown in figure 1, both a button and a door may have their own blueprints which make them play animations, sounds, and activate events in each other (e.g. pressing the button calls the open event on the door which plays an animation and moves it's hit-box).

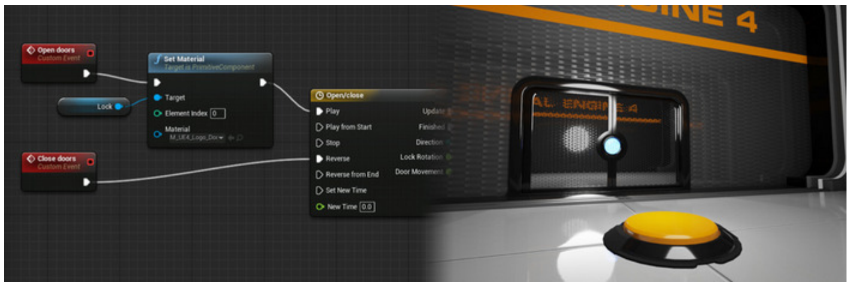


Fig. 1. Simple example blueprint for a door which can be opened and closed. Sourced from Unreal engine's official documentation[7].

Using blueprints can allow for a far more intuitive view of complex structures, particularly where conditional branches occur. In standard scripting languages you must follow logical paths in a program by scrolling between blocks of code and using potentially unreliable intellisense to jump between functions and classes. Conversely, the blueprint system allows you to do the same simply by following the lines between nodes, and even includes the ability to highlight execution paths as seen in figure 2 below.

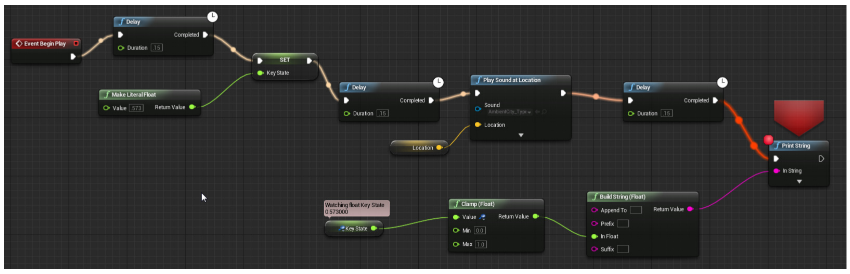


Fig. 2. Example blueprint showcasing the visual indicator used to highlight flow-control. Sourced from Unreal engine's official documentation[7].

2.3.2 Unity Engine (Shader Graphs)

Unity engine is a 3D game engine developed by Unity Technologies, first released in 2005[19] and with major versions being released yearly (version 2023.2 being the most up to date at time of writing, per Unity's user manual[23]). The goal of Unity has been to democratise the creation of video games, making it easier for non-experts to build their own game without having to deal with implementing complex systems like physics engines, render pipelines, and scene hierarchy. Unity boasts the ability for users to create games without any coding[22], with it's many built-in visual tools like shader graphs and bolt[24] (this analysis will be focusing on shader graph in particular). There also exist many user-made assets which use visual scripting such as MapMagic 2[17] (for procedural generation of terrain) and xNode[1] (for creating custom node-based editors).

Shader graph is a visual scripting alternative to shader languages like hlsl and glsl in which a node-based editor is used to build a shader, and is later automatically converted to optimised shader code. This approach to creating shaders is able to take advantage of it's highly visual nature to show previews of the output at each step (as seen in figure 3 below), rather than only being able to see the end result. Having these previews makes debugging much easier by allowing one to see exactly where a problem begins, and also helps understand the processes going into building the final result by showing the step-by-step process going into creating the output.

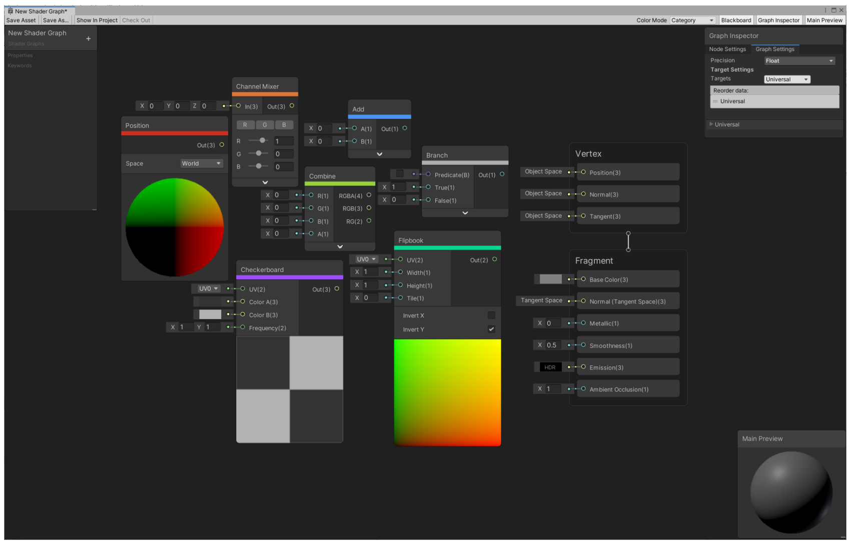


Fig. 3. Example shader built using Unity's shader graph tool. Sourced from Unity's official documentation[21].

2.3.3 *FrameGraph (Frostbite)*

FrameGraph is a render pipeline generation tool created as part of the Frostbite game engine[9], designed to automatically optimise resource management, and execution order of pipelines. During a 2017 GDC talk, EA developer Yuriy O'Donnell showcases the FrameGraph tool, and discusses it's underlying principles and methods[16].

A notable feature of FrameGraph is it's ability to generate pdf graphs of the pipelines it builds, an example of which can be seen in figure 4. The graphs highlight FrameGraph's ability to optimise parallel operations and memory barriers in complex render pipelines, as well as allowing for more efficient visualisation and debugging of these pipelines.

Pipelines are generated by populating what amounts to a list of configurations and dependencies used in a later stage to allocate resources to the GPU and determine the optimal execution order. Having knowledge of the entire pipeline allows for individual passes to be sorted and parallelized based on their requirements and dependencies. Graphs provide an intuitive manner of visualising this by highlighting the dependencies of each pass and showcasing blocks of parallel execution with horizontal alignment of the respective nodes.

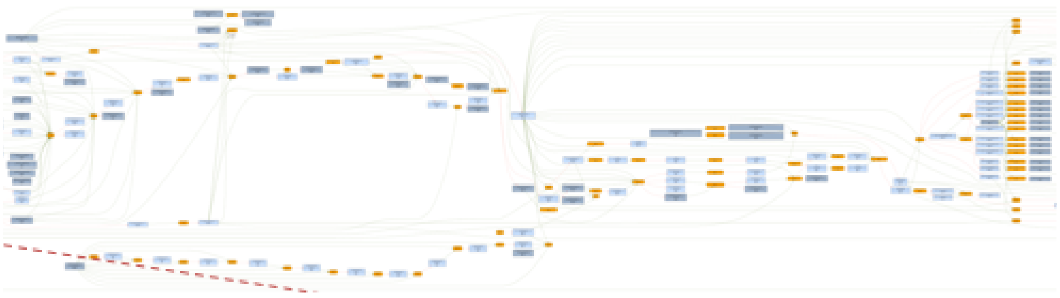


Fig. 4. Example snippet of a graph generated by the FrameGraph tool for Battlefield 4. Image sourced from slide 17 of Yuriy O'Donnell's GDC talk[16].

2.3.4 Scratch (MIT)

Scratch is a coding community and visual scripting language designed to teach young children how to program[12]. The first stable version of scratch (1.0) was released to the public in 2007[13], with the latest version at time of writing (3.29.1) being released 2022. As a programming language, Scratch is designed to be simple and intuitive, in order to help children learn and practice the core concepts around programming.

The interface for Scratch consists of a display panel, a code panel, and a command panel as can be seen in figure 5 below. Operations are denoted by jigsaw-like nodes which connect to each other vertically, or can be placed inside each other to create operations which often read like sentences (particularly with conditional statements and mathematical operations). Figure 6 shows a close-up example of this for drawing a frame of Conway's Game of Life.

Visual scripting in this case provides an excellent tool for learning programming by reinforcing important concepts visually. While Scratch does not conform to the more common designs for visual programming, it instead provides a simpler and easier to digest approach; especially so for people with little to no programming experience.

[1] Project source: https://scratch.mit.edu/projects/345738692/

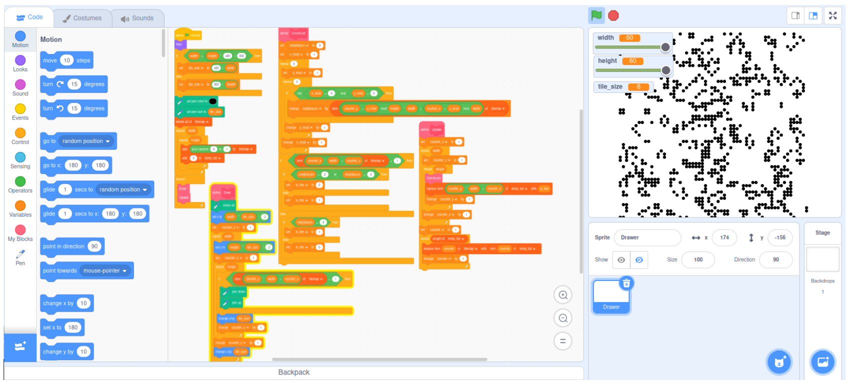
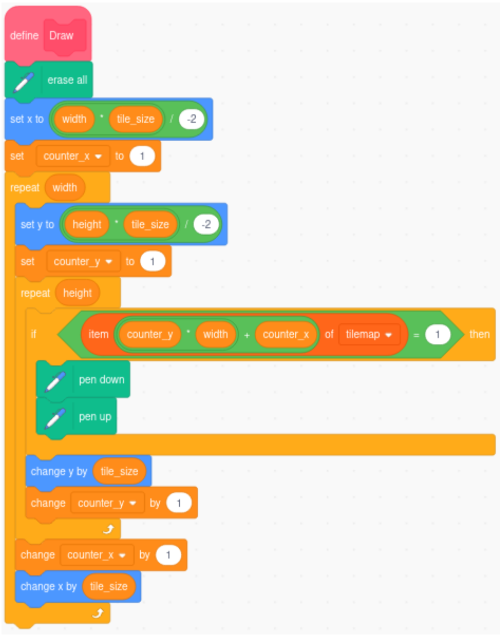


Fig. 5. Example of Conway's Game of Life I built in Scratch [2] .



```
define Draw
erase all
set x to width * tile_size / -2
set counter_x ▾ to 1
repeat width
    set y to height * tile_size / -2
    set counter_y ▾ to 1
    repeat height
        if item counter_y * width + counter_x of tilemap ▾ = 1 then
            pen down
            pen up
        change y by tile_size
        change counter_y ▾ by 1
    change counter_x ▾ by 1
    change x by tile_size
```

Fig. 6. Close-up snippet of the code shown in figure 5.

3 Implementation

3.1 Tools and Libraries

This project is written in C++, built using CMake, and makes use of the following libraries:

SDL2	- For a cross-platform application window back-end.
Dear ImGui	- For displaying and handling UI widgets.
imgui-node-editor	- For displaying the graph interface.
OpenGL	- For interfacing with the GPU
GLM	- For handling mathematical functions and data structures.
nativefiledialog	- For creating platform native file dialogues.
stb	- For reading images from files.

In order to work between multiple of my devices I have written this project to support both Windows and Linux operating systems (tested on Windows 10, and Linux Mint 20). While use cases for this type of project may vary, for the purposes of this development and analysis there will be no major hardware requirement. Most modern devices should be able to handle the tool.

3.2 Graph Interface

3.2.1 Structure

This project's editor is structured as a graph, containing nodes which connect to each other via ports. Each node has it's own attributes and usually represents only one thing (be it a mesh, shader, render-pass, etc.). The ports of a given node have a fixed type, determined by a variadic template and using STL variants to pass values (though some nodes use dynamic ports to change their output type based on conditions). I chose to use variants instead of STL anys as variants are more type safe, provide a simpler and faster method of identifying type, and make the code a lot more clear/self-documenting.

Ports can also be identified as dynamic, which is mainly significant for deserialization (see section 3.3). Dynamic ports represent those which can be added or removed at runtime, such as the texture input ports for the framebuffer node, which are added and removed at the user's request; or the uniform ports on a render-pass node, which are updated to match the uniforms of the linked shader (see figure 7). This is useful for situations in which the number of ports a node will have is unknown, or where an output port should only be available or have a different type under certain conditions.

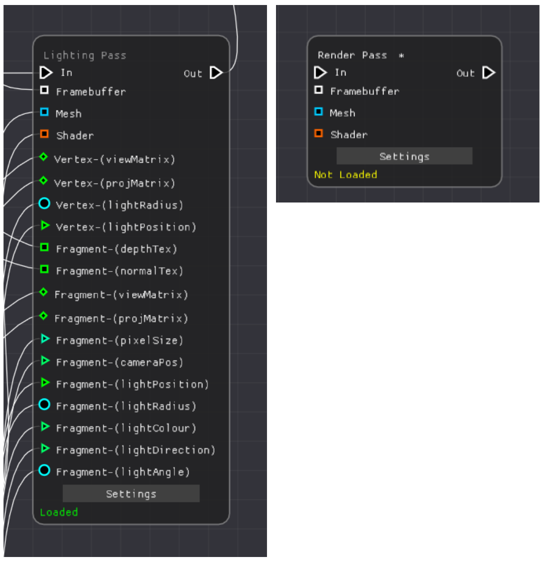


Fig. 7. Comparison of two render-pass nodes, with the left having generated uniform ports for it's connected shader.

The majority of nodes in the current implementation represent the creation of a value, with flow control and operation nodes being the notable exceptions. Data is passed between ports by value, though for complex or API dependant structures they are passed by pointer. For example, the declaration of a port which accepts only shaders would look like: `Port<Shader*> myPort` (passing the pointer itself by value). Callbacks are used to send the data on demand, as well as to handle update events (including on-link, on-unlink, and on value update).

Ports which can handle multiple types, such as the input port for the arithmetic node, will accept any input type that is a subset of it's own. For example, a port declared as `Port<int, float, glm::vec3>` could accept a connection with a `Port<int>` and a `Port<int, glm::vec3>`, but cannot accept a connection to a `Port<int, long>` as `long` is not an accepted type of the initial port (regardless of whether the actual value being stored is valid). Though the functionality exists, as of current implementation there are no nodes which use an output port containing multiple types (although there are nodes which use dynamic ports to change the type of the output).

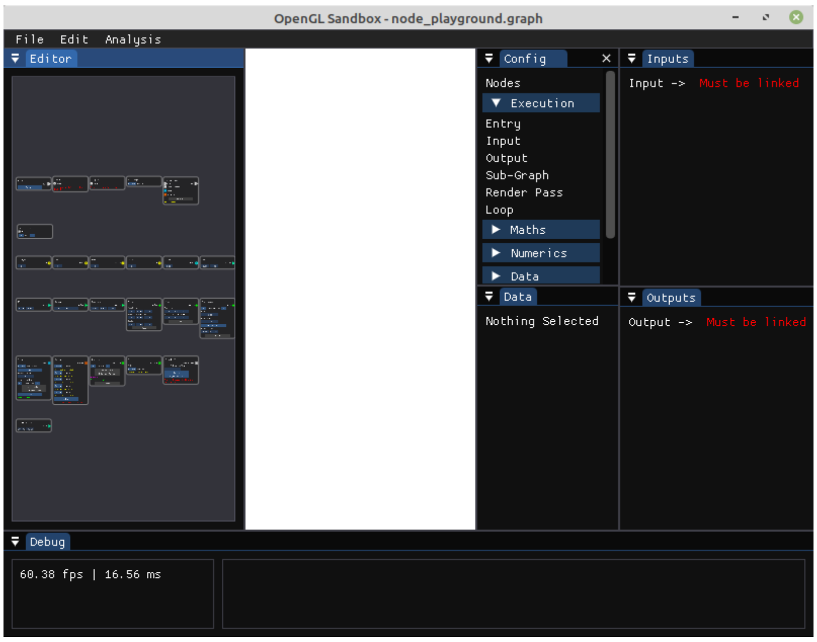
Shown below is some simplified code for defining a node which can perform a type cast from int to float:

```
class CastNode : Node {
public:
  CastNode() : Node("Cast") {
    addPort(mIntIn);
    addPort(mIntOut);

    mIntIn.addOnUpdateEvent([this]() { mFloatOut.valueUpdated(); });
  }

  [[nodiscard]] unsigned int getTypeID() const final {
    return (unsigned int)NodeType::Cast;
  }
protected:
  typedef std::vector<std::pair<std::string, std::string>> serial_data_t;
  [[nodiscard]] serial_data_t generateSerializedData() const final { return {}; }
  void deserializeData(const std::string& dataID, std::ifstream& stream) final {}

  void drawContents() final {}
private:
  Port<int> mIntIn = Port<int>(*this, IPort::Direction::In, "In", "In");
  Port<int> mFloatOut = Port<float>(*this, IPort::Direction::Out, "Out", "Out",
    [this]() { return mIntIn.isLinked() ?
                  (float)mIntIn.getLinkedValue<int> : 0.0f; });
};
```

As shown above, a node class consists of three main parts: port definitions, serialization, and contents. The port definitions involve specifying which ports exist (stored as private members) and are active (determined using the addPort and removePort protected functions). Each port must be given a unique name (per-node) which is used in serialization, and any port can be given a callback which defines how a value should be retrieved when requested. Every node must also define a drawContents function, which is usually filled with buttons and inputs for the node, to allow manipulation of any values (the above example has no need for value manipulation so this is left empty). It is also worth noting that every node must have a unique (per-type) ID defined in the getTypeID, used to identify which class to instantiate during de-serialization. I store this value in the NodeType enum within the NodeClassifications.h file, though the only requirement is for there to be no duplicates.

3.2.2 Interface

Aside from the graph editor itself, there are five supporting interface panels. The background of the application is where the pipeline actually renders to, though future implementations may see this moved into it's own panel. Figure 8 below shows the structure of the application. Panels can be moved an docked as desired, so actual layouts may differ by user preference.



60.38 fps | 16.56 ms

Fig. 8. Example image of the full application.

Editor Panel:

The graph-based editor can be found in the editor panel. Individual nodes are represented as blocks with coloured ports on the sides (the colour and symbol of the port providing a visual indicator for which type they represent). Below the ports are relevant input widgets and messages for configuring the operation handled by the node. Texture nodes also contain a preview for the image they represent.

Debug Panel:

The debug panel contains performance information (fps counter), as well as any error messages which crop up (mainly just caught OpenGL errors). As of current implementation, the error message panel should remain empty since all potential errors are expected to be caught by the nodes before they become an issue.

Config Panel:

Nodes are created from the config panel, which sorts them into size categories:

- *Execution* - For flow-control and input/output related nodes.
- *Maths* - For nodes related to mathematical operations (currently only contains arithmetic node).
- *Numerics* - For numeric data nodes (integer, float, vector, and matrix types).
- *Data* - For numeric nodes with special conditions applied to their values (such as colour which fixes it's values between zero and one).
- *Graphics* - For graphics related data structures (including shaders, meshes, and textures).
- *System* - For data node which depend on dynamic properties such as current time or window size (currently only contains screen size node).

Clicking on the button for a given node in the config panel will create a new node of that type in the currently opened graph.

Data Panel:

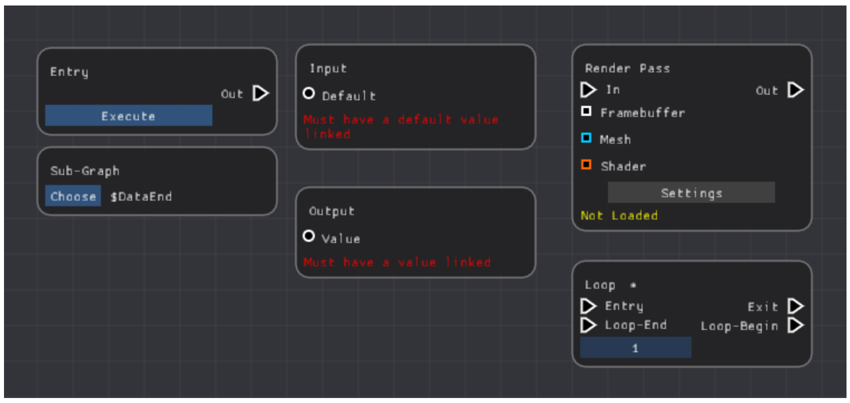
The data panel contains node-specific information which cannot reasonably fit on the node itself, such as the vertex data of a mesh node. This was initially created as a workaround to the limitations of the node editor library being used, though may be expanded in future to contain all content data of a node (except for names, ports, and certain messages).

Input/Output Panels:

Finally, the input and output panels contain lists of every input/output node within the graph, as well as what values they hold (if applicable) and whether they are valid or not. The input and output nodes mainly exist for use in sub-graphs, though the panels can be used to set and read the values to test that the graph works as expected.

3.2.3 Flow-Control

All pipelines begin from a single entry node, which has one output for pipeline execution. While it is possible to connect an entry node to many other nodes, only the first will be considered when building a pipeline (the same happening for any other flow control nodes). Future implementations may see the ability to link to many flow control points to allow multi-threading, or simply to denote operations which don't depend on a specific order of operations. In order to make a rendering call, one or more render-pass nodes must be within a chain of links starting at a given entry node. Figure 9 below showcases all nodes directly, or indirectly, related to flow-control.



Must have a default value linked

Must have a value linked

Fig. 9. All nodes capable of handling flow-control.

Each entry node has a button which, when pressed, will attempt to generate a pipeline from the connected render-passes. On request, the node will first validate the pipeline to ensure it can actually execute without error. If validation fails, the entry node will display an error message, and any invalid render-passes will show a list of problems that need to be fixed before execution is allowed. This will occur if, for example, a render-pass has no shader or mesh attached, or if one of it's attached components is invalid.

Render-pass nodes contain one output and four main input ports. The output, and one of the inputs, is for flow-control; representing an entry and exit point of execution respectively. The other three input ports represent the necessary data structures for the render-pass. Both a mesh and shader are required for the pass to be considered valid, and they represent the operation being executed (be it drawing a model to the scene, or applying post processing effects, etc.). Framebuffers are an optional parameter which, if present, will determine which textures the shader will write to. If no framebuffer is provided the backbuffer will be drawn to (which results in the actual background of the application being updated). Render-pass nodes also contain a list of inputs for settings such as buffer clearing and alpha-blending.

When a valid shader is connected to a render-pass, several input ports will be generated which represent each uniform present in the shader. Ports are generated by parsing the shader code with regular expressions to find all uniforms and then deduce their types, names, and default values. Currently all uniform types are supported, except for uniform arrays, structs, and more complex buffer objects like SSBOs (functionality for which may be implemented in future).

It should be noted that despite OpenGL treating identical uniforms in separate shader passes as the same variable, the render-pass node will generate separate ports for each pass (similar to how other graphics APIs will treat uniforms in separate passes as independent). This will likely be changed in future implementations to more closely match OpenGLs approach and remove the ambiguity currently present (though the functionality may be preserved in some form to allow potential expansion into using other APIs).

3.2.4 Sub-Graphs

Sub-graphs nodes can be used to encapsulate the functionality of a graph into a single node by letting the user pick a saved graph to load into the node. When loading in a sub-graph, it will look for every input and output node, then generate a corresponding input/output port of the correct type, as shown in figure 10 below. Currently this supports all types including flow-control, which allows for functionality like individual render-passes to be encapsulated within sub-graphs, while abstracting away data which will remain constant.

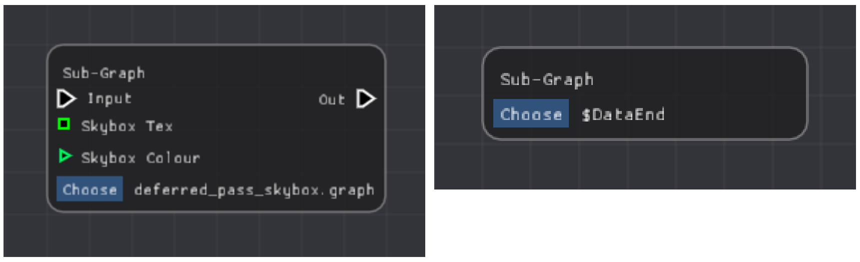


Fig. 10. Comparison of two sub-graph nodes, with the left having loaded a graph used to carry out the skybox pass of a deferred pipeline.

The main purpose of this node is to simplify more complex graphs, as well as allow for the same operation to be used in multiple places without having to redefine it each time (much like a function in standard scripting). One of the example graphs in the project showcases this by comparing a simple deferred pipeline using a single graph, with one using sub-graphs, highlighted below in figure 11.

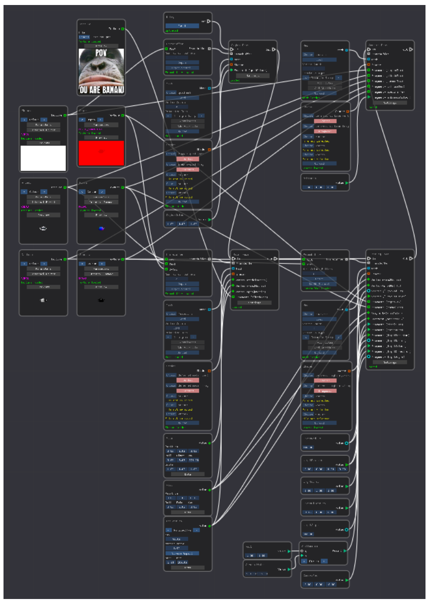
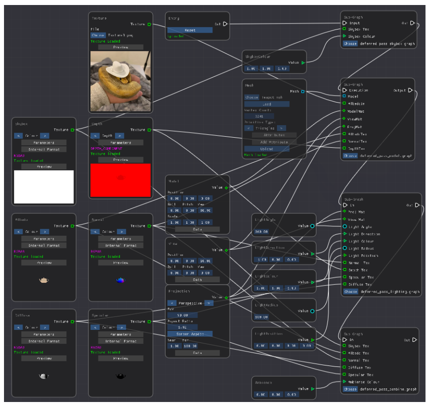



Fig. 11. Two graphs running the same deferred render pipeline, but with sub-graphs used on the right to reduce the visual complexity.

The current implementation of sub-graphs leaves plenty of room for improvement, in both optimisations and better user interactions. In particular, when loading multiple of the same sub-graph the program will actually load the exact same graph twice, rather than loading once and referencing it twice (resulting in unnecessary memory overhead). It would also be useful to allow for multiple graphs to be opened at one time, in particular to allow easy tweaking of a graph encapsulated in a sub-graph node while also editing the parent graph.

Input and output nodes can also be improved by having separate nodes for every input/output type, thus removing the need for a compulsory default value to decide it's type. This would allow for some inputs to become mandatory while still allowing the option of a default, as well as making them more intuitive to work with (removing the dynamic ports and ambiguously typed input ports).

3.3 Serialization

3.3.1 Graph

Graphs are serialized in a custom human-readable format, designed with some degree of forward compatibility in mind. Earlier in the project I used a more efficient, closer to binary, format which assumed it would know the size of given components and simply read and write that number of bytes, and would assume that particular properties will always exist. The issue with this format however, arises when attempting to modify it in any way, usually by implementing a new feature (such as adding a new node type, or changing an existing one). Changing how the format works in any way would immediately invalidate any existing graph files, forcing me to recreate each graph just to test they work.

One potential solution would be to create a slightly different algorithm for each version of serialization, and use metadata to decide which version to read a graph as. This method however, would very quickly become convoluted and result in a lot of redundant and obsolete code. Instead, my solution was to create a new format for my graphs which uses a series of prefixes, markers, and human-readable names to denote data. The format roughly follows the following schema (subject to minor changes in future implementations):

```
# [n] - Number, (n) - Text, comments (#) are ignored
[unique-graph-id]
$NodeBegin
^NodeType (node-type-name)
-NodeID [node-unique-id]
-NodeName (node-display-name)
-NodePosition [pos-x] [pos-y]
-InPort (port-unique-name) [unique-id-of-linked-port] *
-OutPort (port-unique-name) [unique-id-of-linked-port] *
-Dynamic-InPort (port-unique-name) [unique-id-of-linked-port] *
-Dynamic-OutPort (port-unique-name) [unique-id-of-linked-port] *
$DataBegin
... **
$DataEnd
$NodeEnd
```

In the example above the * marks are not included, and simply indicate which lines will repeat for every port a given node has. At the ** mark is per-implementation data (for example, a vector node will store it's numeric data here). The NodeBegin and NodeEnd marks (and their contents) are repeated for every node within the graph (changing the exact values for each node). For brevity I will not include a full example in this text, however if you are interested more examples can be found in the GitHub repository.

The actual process of serialization and de-serialization can be organised into four sections as follows:

Graph Level:

The graph begins serialization by writing it's unique ID (used to prevent infinite loops with sub-graphs). Then, for each node it will write the corresponding *BeginNode* and *EndNode* markers, wrapped around node-specific data. The graph itself will write the type node's specific type and will then request for the node to serialize it's own data.

On de-serialization, the graph will begin by seeking the pairs of *BeginNode* and *EndNode* markers which wrap the data of each node. The *NodeType* attribute will then be used to deduce which specific type of node needs to be instantiated and added to the graph. From here the data between the two markers will be sent to the node for further de-serialization

Node Level:

Every node will serialise their ID, graph-space position, and a list of all port specific information (including which the ID of any linked ports, or -1 if there are none, and whether they are dynamic). Each port uses a unique name (unique per node, not overall) instead of an automatically generated ID for both readability and to avoid issues with handling numeric IDs in the node-editor library I use. As such, the actual numeric ID of a port (as well as any other object with an automatic ID) is generated at runtime and may not be consistent with past executions. Finally, the node will make use of the pure virtual generateSerializedData() function (wrapped between the *BeginData* and *EndData* markers) to write per-implementation data.

De-serialization begins by finding each attribute and reading them directly into their respective variables. Links between ports must be handled separately, so they are appended to their corresponding arrays and handled on the Port Linking Level (described below). The data within the *BeginData* and *EndData* markers is then parsed on the Implementation Level using the pure virtual deserializeData() function.

Implementation Level:

Each implementation of the abstract Node class must define two functions related to serialization; generateSerializedData() and deserializeData() (with the optional onDeserialize() function to allow events to be called after de-serialization is finished).

The generateSerializedData() function expects a return value containing a list of all data to be written to the output file. This is done by populating and returning a std::vector<std::pair<std::string, std::string>>, where the strings represent the unique (per-node) name of the data point, and it's serialized value respectively. Data from this array is iterated over in the above stage and serialized.

The deserializeData() function takes in a string (representing the unique data point name from serialization) and a stream pointing at the location of the data itself. This function is then expected to use this data to determine which variable it should write to (usually via a sequence of if statement checks), as well as what method should be used to de-serialize the data.

Shown below is a reduced example of the code used to serialize and de-serialize the TextureNode:

```
typedef std::vector<std::pair<std::string, std::string>> serial_data_t;
serial_data_t TextureNode::generateSerializedData() const {
  serial_data_t data{};
  data.emplace_back("Type",
    SerializationUtils::serializeData((int)mTextureType));
  data.emplace_back("ScreenLock",
    SerializationUtils::serializeData(mIsScreenLocked));
  data.emplace_back("Bounds",
    SerializationUtils::serializeData(mTexBounds));
  data.emplace_back("Min",
    SerializationUtils::serializeData((int)mMinFilter));
  data.emplace_back("Mag",
    SerializationUtils::serializeData((int)mMagFilter));
  data.emplace_back("Wrap",
    SerializationUtils::serializeData((int)mEdgeWrap));
  return data;
}

void TextureNode::deserializeData(const std::string& dataID,
                                  std::ifstream& stream) {
  if (dataID == "Type")
    SerializationUtils::deserializeData(stream, (int&)mTextureType);
  else if (dataID == "ScreenLock")
    SerializationUtils::deserializeData(stream, mIsScreenLocked);
  else if (dataID == "Bounds")
    SerializationUtils::deserializeData(stream, mTexBounds);
  else if (dataID == "Min")
    SerializationUtils::deserializeData(stream, (int&)mMinFilter);
  else if (dataID == "Mag")
    SerializationUtils::deserializeData(stream, (int&)mMagFilter);
  else if (dataID == "Wrap")
    SerializationUtils::deserializeData(stream, (int&)mEdgeWrap);
}
```

What this method lacks in efficiency, it makes up for greatly with ease of implementation and readibility. For most uses, graphs should not become complex enough to make saving and loading a significant bottleneck in performance. As such there is plenty of leeway to allow for the method shown above where values can be added and removed on demand, without causing invalidation of existing graphs. The main development issue to keep an eye out for however, is name clashing (where two values are given the same data ID) which will cause one of the values to be surpressed or read incorrectly.

It is also worth noting that I use the custom made static utility class `SerializationUtils` to handle serialization and de-serialization of different data types through overloaded functions. This allows for data of the same type to be consistently serialized in the exact same way and makes ambiguous cases easier to handle (such as handling whether to use forward '/' or back '\' slashes in filenames depending on Operating System). For enum values I simply cast to a corresponding numeric value (usually `int` for serialization, and `int&` for de-serialization).

Port Linking Level:

While ports can be serialized at the node Level, de-serialization must occur after all nodes have been generated. This is to prevent a port from attempting to link with a node that has not yet been made.

Link data is separated into four data structures; one for each combination of input/output ports and dynamic/static ports. Links with dynamic ports must be de-serialized after completely static links as they may be dependant on data generated after linking. For example, the render-pass node requires a link to a shader node in order to determine which uniform ports it requires. Therefore the render-pass node must first link to a shader node, which can then give the required information to generate the uniform ports, which can now have links applied to them.

One flaw with the above method is when a dynamic port depends on another dynamic port to first be generated, as the order of generation may result in the dependency port not being generated before the dependant port. To get around this I wrap the operations in a simple `while` loop to keep attempting to link dynamic ports until no more links can occur.

3.3.2 Mesh

Meshes in this project use a custom format which allows for complete user control over vertex attributes. This takes the form of allowing users to add and remove attributes, decide what type each attribute should be, and even edit individual values of each attribute array (including the index buffer).

While this project does support reading from Wavefront *.obj* format files (which is a commonly used format for storing mesh data), the additional control over attributes makes the *.obj* format insufficient for storing these custom made meshes without significant changes. As such I have created my own format using the *.msh* extension which roughly follows the following schema:

```
# [n] - Number, (n) - Text, {n} character (i or f), comments (#) are ignored
# GLSandbox MSH File: (path-to-file)
[num-vertices] [num-indices] [type]
(attr-name) [num-components] {type} [binding]
[value-n]#...
#...
```

And shown below is an example of the *quad.msh*[3] file used in the project:

```
# GLSandbox MSH File: Meshes/quad.msh
4 0 1
position 3 f 0
1 1 −1
−1 1 −1
1 −1 −1
−1 −1 −1
uv 2 f 1
1 1
0 1
1 0
0 0
```

This format of storing meshes allows for any number of attributes to be defined, rather than just the conventional ones (position, uv, and normal). Bindings can also be set manually to allow compatibility with a range of shaders.

Because the *.obj* format does not include tangent data, I have included functionality in the mesh node to allow tangents to be generated if a mesh is loaded from an *.obj* file. This mesh can then also be written to a *.msh* file, removing the need to generate tangents every load. I included this functionality because tangent data is required for some shaders, particularly in relation to deferred rendering, and can be deduced from normal and index buffers (which are usually present in *.obj* files).

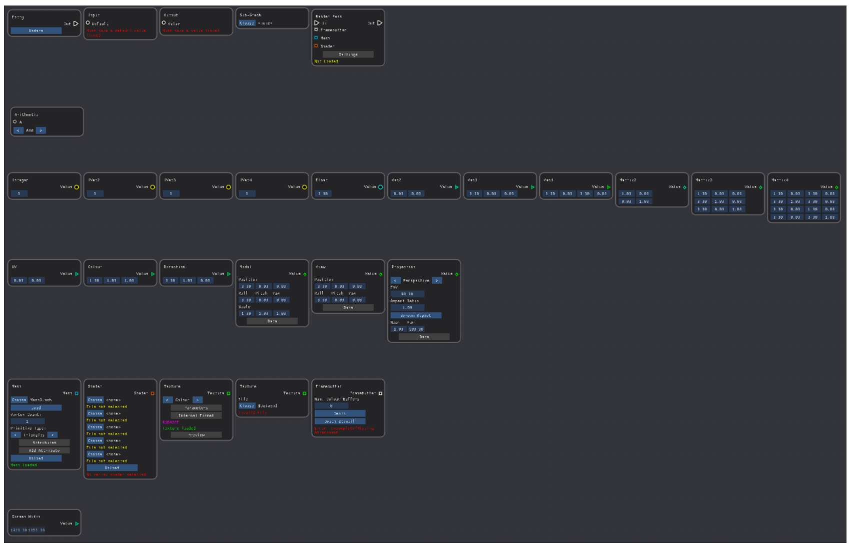
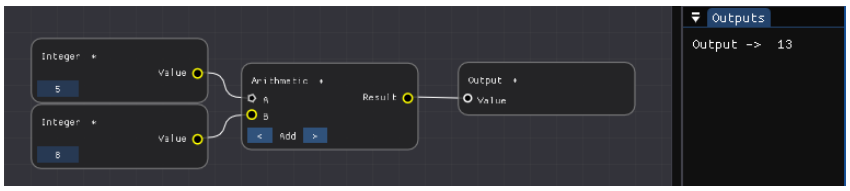


Fig. 12. Example graph containing all nodes present in the current version of the project, as of the time of writing.



Output -> 13

Fig. 13. Example graph showcasing a simple addition operation, with the output shown on the right.

[3] The *quad.mesh* file, and others, can be found in the GitHub repository.

3.4 Pipeline Generation

Generation of the render pipeline occurs in two steps:

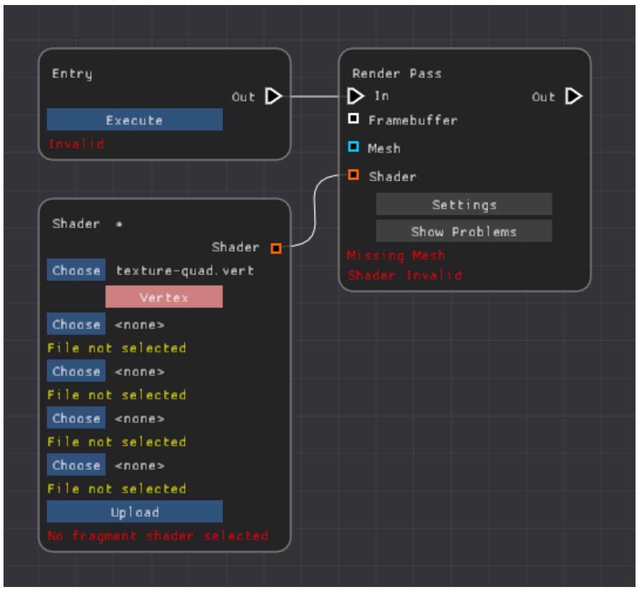
Pipeline Validation:

Before a render pipeline can be generated, it must first be validated to ensure it can be made in the first place. This involves recursively iterating through the graph, starting at the entry node and passing to each linked flow-control node, and validating the state of every connected node.

Each node has it's own method of ensuring validation, with some being more complex while others require no validation at all. Validation checks occur roughly as follows (with omitted nodes having no required checks):

- *Render-pass node* - Must have a valid mesh and shader node connected, and if a framebuffer is connected it must also be valid.
- *Sub-graph node* - Validity of this node is implicit by checking the contents of the loaded graph in the same way.
- *Mesh node* - Must have a valid mesh uploaded to the GPU.
- *Shader node* - Must have a valid shader uploaded to the GPU.
- *Texture node* - Must have a valid combination of internal-format parameters, and have a valid texture uploaded.
- *Texture node (from file)* - Must have successfully read a texture from a file and uploaded it.
- *Framebuffer node* - Must have one or more buffers, and all buffers must be linked to a valid shader.

In the event one or more nodes are flagged as invalid, generation of the pipeline will be cancelled and the user will be prompted to fix any issues. Render-pass nodes which have been identified as invalid will display a list of problems which prevented generation, shown in figure 14 below. Most nodes with an invalid state will have their own validity checks occur during editing; for example with the shader node updating it's error message as the user changes it's parameters.



Invalid

texture-quad.vert

<none>
File not selected
<none>
File not selected
<none>
File not selected
<none>
File not selected

No fragment shader selected

Missing Mesh
Shader Invalid

Fig. 14. Example graph showcasing an invalid pipeline (render-pass missing mesh and connected to invalid shader).

Pipeline Generation:

Once the graph is known to be in a valid state, a pipeline can be generated by recursively iterating through the graph and filling in an array of callbacks which get executed every frame. Each render-pass node will build it's own list of callbacks to concatenate to the full pipeline, with different options for executables and their parameters based on the inputs to the render-pass.

This method of pipeline generation allows for all of the conditional checks to be applied only once, rather than every frame. As such, the final pipeline will end up slightly faster since it needs fewer redundant checks, though in return it results in more indirection in the form of several functions the program must enter and exit each pass.

The pseudocode below roughly outlines the process of building the pipeline for a single pass:

```
function generateRenderPassPipeline(pipeline)
  settings<- # Wrapper for general render-pass settings

  shader<- # Retrieve the shader for this pass
  mesh<- # Retrieve the mesh for this pass
  framebuffer<- # Retrieve the framebuffer for this pass, or null

  if framebuffer != null
    pipeline.append(lambda(){framebuffer.bind()})
  endif

  pipeline.append(lambda(){shader.bind()})

  pipeline.append(lambda(){
    if settings.viewport == null
      resetViewport()
    else
      setViewport(settings.viewport)
    endif

    # Other settings omitted for brevity

    if settings.doClear
      setClearColour(settings.clearColour)
      clearBuffers(settings.bufferMask)
    endif
  })

  textures<- # Retrieve a list of all sampler (texture) input uniforms
  if textures != null
    pipeline.append(lambda(){
      for (i = 0; i < length(textures); i++)
        textures[i].bind(i) # Bind tex and set it's attachment to i
      endfor
    })
  endif

  pipeline.append(lambda(){
    mesh.bind()
    mesh.draw()
  })

  if framebuffer != null
    pipeline.append(lambda(){framebuffer.bind()})
  endif

  return lambda(){
    for callback in pipeline
      callback()
    endfor
  }
endfunction
```

4 Evaluation

4.1 Utility

I believe this project acts as an adequate proof of concept for using a node-based editor to create an easy to build, tweak, and profile render pipeline generation tool. Visual scripting provides an intuitive method of visualising logical operations, and does so even better for visual applications like Unity's shader graph and this project.

Though not in a complete state, this project is already at a point I would consider useful for it's intended purpose of building and testing, at least simple, render pipelines. Future implementations could see improvements to many areas, especially in the realm of performance profiling; though in it's current state the tool is capable of handling many of the simpler pipeline operations, making it useful for testing and gaining a better understanding of the structure and utility of basic shaders and render pipelines.

In terms of visual appeal and ease of use, the application serves it's purpose; though I believe there is room for improvement in making it both look nicer and more intuitive for users to pick up. This can take the form of better tooltips and widgets, but from a more basic standpoint this application tends to have issues with readability and intuitiveness in it's overall layout and use. Preferably an application should be designed under the principle that a user can pick it up without the need of a tutorial and figure it out without issue. To this end, many future changes to this project will involve overhauling the UI (see section 6.1.1) and making improvements to the usability and readability of this application's components.

4.2 Further Use

Currently, this project contains five main examples: *profile.graph*, which shows a render pipeline directly comparable to the analysis pipeline; *simple_teapot.graph*, which draws a teapot mesh over a coloured background; *lit_teapot.graph*, which uses blinn-phong lighting in a simple forward rendering pipeline; *deferred_teapot.graph*, which uses blinn-phong in a simple deferred rendering pipeline; and finally *deferred_sub_graph.graph*, which uses sub-graph nodes to simplify the exact same pipeline used by *deferred_teapot.graph*. I used deferred rendering of a teapot as a bench-mark for what this tool must be capable of handling (at minimum), as this requires implementation of a majority of the most commonly used components of render pipelines.

This tools ability to handle deferred pipelines allows for it to show it's potential usage and provide some minimal utility for toying around with simple systems. However, for more complex render pipelines (particularly those seen in modern industry-standard tools), there are more features that would be required. For example, to implement forward+ rendering the tool would need to be capable of handling both compute shaders and SSBOs (which is not present at the time of writing). As such this tool is still some distance away from reaching the standards required for proper use.

4.3 Profiling

One of the criteria for this project was to analyse the performance overhead of using a visual scripting tool, over using a standard (more hard-coded) method. To test this, I created a simple render pipeline which loops over a single render-pass 200 times each frame (the value 200 was selected by increasing the number of iterations until V-Sync was overridden by a significant margin). The shader being run uses a fairly expensive way of generating a random colour (which can be increase to take longer via uniform). I created this pipeline both as an executable graph, and as a hard-coded option in the analysis menu. The hard coded version uses a simple loop to iterate over the same render-pass, removing any overhead from having multiple indirections via callbacks.

The results of this test were that both methods had an almost identical performance of 20.2fps (49.5ms). Any differences between the two are too small to reliably identify without far more rigorous tests, through per-render-pass profiling and averages over a much longer time-span (currently the fps is calculated as a rolling average over 100 frames).

It is possible that a more complex render pipeline would have a more significant performance difference, as adding in more parts to each render-pass will create more indirections and require more callbacks. However I believe these results at least show that the use of callbacks on it's own has fairly negligible impact compared to pure iteration.

To improve on this test further I would want to first implement better profiling methods (which I elaborate on further in section 6.2.2), as well as more test cases (including closer to real world examples). It would also be good to work on further reducing indirection to the point where CPU caching could be optimised, especially in the analysis of pipelines, as this would allow a better comparison of approaches.

4.4 Comparison With Existing Tools

While this tool is nowhere near the standards of Unity and Unreal engines, I do believe it at least showcases a potential use case of equivalent utility to their visual scripting tools. If developed further, this tool could provide a useful utility for software developers to iterate and test render pipelines and shaders.

Unity makes good use of it's visual scripting tool, shader graph, by allowing the users to preview the results of their shader in steps through per-node images. I was able to implement a similar feature in the texture node previews which allows users to see the state of individual textures as they end up by the end of execution, similar to the commonly used debugging technique of modifying a shader to output the contents of one of the in-use textures.

In terms of visual style, Unity and Unreal have a few notable features in their visual scripting tools which would work quite well for this project. In particular, the range of styles they have for nodes is much greater, with Unity's shader graph showcasing much simpler default-value nodes that lack any significant form of descriptive traits in return for taking up less space. This makes them useful for simpler values like numbers, colours, and booleans as they don't need to take up a lot of space and the input area itself can often give enough information about the intended value type. As of current implementation, the GLSandbox tool only has one style for nodes regardless of type. Implementing more styles would allow for greater flexibility in organisation of graphs and make certain features like comment nodes easier to add in as well.

Many applications of programming involve visual outputs, such as graphical shaders or logical motions and decisions of an actor in a scene. Quite often, these programs can be better visualised using visual scripting to provide a more structured display of the logic and flow of an operation, such as how Unity's shader graph uses preview images to help visualise the steps taken to produce a final image. I believe the application of building, testing, and comparing render pipelines is a natural continuation of this idea, which can take full advantage of the visual aspect of a graph-editor to display a normally complex system of render pipelines in an intuitive manner.

5 Conclusion

Overall, I believe this project can showcase the potential utility of node-based editors as a tool for testing and profiling shaders and render pipelines. The project is still in a prototype stage, at the time of writing, and there are many features which can still be implemented and improved upon. Yet I believe that even in it's current state the project can be quite useful.

The efficiency of the tool being comparable to hard-coded alternatives shows this tool does not suffer greatly from it's use of visual scripting. Although attempting to push the efficiency to industry standards may prove problematic as it would require optimisations which are difficult, if at all possible, to implement in such a generic application (such as optimising CPU caching, memory barriers, and threading). This however, is an issue that even industry standard tools like Unreal are aware of, stating in their official documentation that native C++ is preferable where a lot of operations or complex mathematics is occurring every frame. As such, while this tool should seek to maintain a strong and useable performance, there is no need to over-complicate the process to exactly match the efficiency of hard-coded methods.

6 Further Work

6.1 *Improvements*

6.1.1 *IMGUI vs RMGUI*

One of the main problems with the implementation of this project is the usage of an Immediate-Mode GUI (IMGUI). While useful for quick debug utilities, IMGUI based UIs are really not suitable for this type of application, and it's usage has caused no end of struggles in implementation. This is due to the nature of an IMGUI layout not knowing the size or state of elements until the moment they are drawn, making it difficult to size widgets relative to each other. The libraries I have used are also not without issue, such as drawing drop-down headers and popups incorrectly within the graph panel, and having no simple method of creating a scrollable area within a node.

Because of this, a Retained-Mode GUI (RMGUI) layout (such as Qt) would be more suitable, allowing for better control over the layout of nodes. RMGUI would also have the advantage of allowing view and control to be separated (which IMGUI struggles with). This would allow for significant improvements to both readability and organisation of many areas within the code-base, where the current implementation has used hacky workarounds to make the system work with a IMGUI layout.

6.1.2 *Pipeline Generation*

As of the current implementation of the project, at the time of writing, there are a few methods in which the generation of a render pipeline could be improved. The main improvement would be in sending data describing each render-pass instead of generating callbacks. This would allow for an algorithm to parse each pass relative to the other, and better optimise the joins between them. For example, the current implementation has each render-pass apply all settings every time, even if a given setting is already the same as a previous pass. Being able to see the information of preceding passes would allow for only the functions which would actually change something to be called.

Another improvement, which would likely relate to the previous in implementation, would be to remove, or at least reduce the number of, relied upon callbacks. This may involve generating a single large callback which handles the entire render pipeline, or simplifying and combining the existing callbacks. A result of this would be fewer indirections and better CPU caching during execution, as the CPU no longer has to jump to separate areas of memory to find instructions. While this level of optimisation may be somewhat overkill for the purpose of this project, it would still have utility in allowing the tool to be better used as a profiler for testing the performance of a given render pipeline.

6.1.3 *Implementation*

Initially I was expecting many of the more complex seeming features, such as subgraphs and loops, to be more difficult to implement. However I was pleasantly surprised to find they fit in quite easily to the systems already in place, with some care taken to keep everything organised. This is not to say however, that the implementations are without flaw. In particular, the current implementation handles flow-control by using a single recursive function to iterate through and check the types of every execution node, when this should be handled via virtual functions to keep per-class functionality within the bounds of their own respective class bodies.

In terms of code organisation and structure there are many areas of improvement. Particularly, future implementations should see functionality split into more relevant classes where they currently sit out of place, such as how the `Window` class handles some UI-specific calls and even manages saving and loading graphs. This functionality should be placed in relevant UI and handler classes, with the `Window` class only containing back-end window handling.

It is also worth mentioning that the port classes could use some work as well. The current implementation uses a header only file, needed due to the use of templates in the `Port` class. This makes the port classes an exception amongst the codebase, causing issues with readability and manageability of the code. To alleviate this, future implementations should see the port class functionality moved into it's corresponding *.cpp* file, with the use of the *impl* file idiom[4] .

6.2 Additions

6.2.1 New Nodes

Aside from the points specified in section 6.1 there are many features and changes which can be implemented to improve this project. A few of those changes cover utilities and tools often seen in game engines, like materials and SSBOs.

Shown below is a list of nodes which I plan to implement.

- *Model* - To represent all the information relevant to a given model (transformation, material, mesh, etc.)
- *Scene* - To represent and iterate through several models to make up a scene
- *Camera* - To encapsulate the properties of the view
- *Light* - To encapsulate the properties of a light in a more intuitive manner
- *Time* - To update shader uniforms over time
- *Tunnel* - To help clean up graphs by allowing long distance connections via pairs of tunnel nodes
- *Branch* - To allow more complex operations, such as conditional render passes

6.2.2 Performance Optimisation and Profiling

Implementing some performance optimisations for the application itself would help improve it's usability. In particular, multi-threading could be used to load assets like textures and meshes without freezing the application (especially for dealing with larger assets like 4k textures). Multi-threading may also be useful for pipeline generation, since some operations may be possible to do in parallel (such as complex mathematics or some form of procedural generation). However for most uses, just implementing multi-threading for assets would be enough.

On a similar note to multi-threading, a more complex pipeline optimisation could be to allow the use of memory barriers, or even automatically deduce where they

[4] See https://isocpp.org/wiki/faq/templates#separate-template-fn-defn-from-decl

should go. This would allow multiple render-passes to run in parallel when they don't depend on each other, or in the case of multi-threading in the pipeline can automatically decide when a thread needs to be finished by. This could significantly improve the performance of more complex pipelines.

Performance profiling support could be an incredibly useful feature for some applications of the tool, as it would allow the comparison of different shaders and render pipelines. Given the nature of the application as a tool for more easily building and testing different render pipelines, a built-in performance profiler would be a more than suitable addition to give more functionality and use to the application. This would likely take the form of a toggle-able switch, which enables per-node timers showing how long each render-pass takes. It may also be helpful to allow other features in this case, such as pausing and single-stepping the pipeline.

7 Acknowledgements

I wish to extend thanks towards my project supervisor Dr. Richard Davison for his guidance and willingness to answer my endless myriad of questions, and to my Mother for her support and advice throughout the project.

References

[1] Thor Brigsted. xnode, Jul 2010. Online; accessed May 2023. URL: https://assetstore.unity.com/packages/tools/visual-scripting/xnode-104276.

[2] Brendan Caldwell. A retrospective of unreal, from the people who made it, Jun 2018. Online; accessed May 2023. URL: https://www.rockpapershotgun.com/unreal-retrospective-from-the-people-who-made-it#comments.

[3] NVIDIA Corporation. The performance multiplier, powered by ai. Online; accessed May 2023. URL: https://www.nvidia.com/en-gb/geforce/technologies/dlss/.

[4] TO Ellis, John F Heafner, and WL Sibley. The grail system implementation. Technical report, RAND CORP SANTA MONICA CA, 1969.

[5] Godot Engine. Godot engine, 2019. Online; accessed May 2023. URL: https://godotengine.org/.

[6] GameMaker. Yoyo games ltd., 1999. Online; accessed May 2023. URL: https://gamemaker.io/en.

[7] Epic Games. Blueprints visual scripting in unreal engine. Online; accessed May 2023. URL: https://docs.unrealengine.com/5.2/en-US/blueprints-visual-scripting-in-unreal-engine/.

[8] Epic Games. Unreal engine, 2019. Online; accessed May 2023. URL: https://www.unrealengine.com.

[9] Electronic Arts Inc. Frostbite the engine, 2023. Online; accessed May 2023. URL: https://www.ea.com/frostbite/engine.

[10] Michael Kenzel, Bernhard Kerbl, Dieter Schmalstieg, and Markus Steinberger. A high-performance software graphics pipeline architecture for the gpu, 2018.

[11] Kristófer Ívar Knutsen. Visual scripting in game development. 2021.

[12] MIT Media Lab. About scratch. Online; accessed May 2023. URL: https://scratch.mit.edu/about.

[13] MIT Media Lab. Scratch timeline. Online; accessed May 2023. URL: https://en.scratch-wiki.info/wiki/Scratch_Timeline#May.

[14] MIT Media Lab. Scratch, 2018. Online; accessed May 2023. URL: https://scratch.mit.edu.

[15] Chris McClanahan. History and evolution of gpu architecture. *A Survey Paper*, 9, 2010.

[16] Yuriy O'Donnell. Framegraph: Extensible rendering architecture in frostbite. In *Game Developers Conference*, 2017.

[17] Denis Pahunov. Mapmagic 2, Mar 2023. Online; accessed May 2023. URL: https://assetstore.unity.com/packages/tools/terrain/mapmagic-2-165180.

[18] David Canfield Smith. *Pygmalion: a creative programming environment.* Stanford University, 1975.

[19] Ars Staff. Unity at 10: For better—or worse—game development has never been easier, 2016. Online; accessed May 2023. URL: https://arstechnica.com/gaming/2016/09/unity-at-10-for-better-or-worse-game-development-has-never-been-easier/.

[20] Unity Technologies. Unity engine, 2019. Online; accessed May 2023. URL: https://unity.com/.

[21] Unity Technologies. About shader graph, 2023. Online; accessed May 2023. URL: https://docs.unity3d.com/Packages/com.unity.shadergraph@16.0/manual/.

[22] Unity Technologies. How to make a video game without any coding experience, 2023. Online; accessed May 2023. URL: https://unity.com/how-to/make-games-without-programming.

[23] Unity Technologies. Unity user manual 2023.2 (alpha), 2023. Online; accessed May 2023. URL: https://docs.unity3d.com/2023.2/Documentation/Manual/index.html.

[24] Unity Technologies. Visual scripting with bolt, 2023. Online; accessed May 2023. URL: https://docs.unity3d.com/2019.3/Documentation/Manual/VisualScripting.html.

[25] Mike Thomsen. History of the unreal engine. 2010. Online; accessed May 2023. URL: https://www.ign.com/articles/2010/02/23/history-of-the-unreal-engine.

Glossary

any Anys are a type of C++ data structure which represents a value of any type (similar to a void). See: CPP-Reference. 8, 28

backbuffer The backbuffer represents the pixel data being actually drawn on the screen. It is often used by default if no framebuffer is specified. 12

Blender Blender is a free and open source 3D sculpting/animation tool. 29

cross-platform Cross-platform programs are capable of executing on multiple platforms/operating systems. 7

framebuffer Framebuffers are graphics structures which determine which texture is being written to by a given GPU operation. 8, 12, 18, 28

Godot Godot[5] is a completely free and open-source game development tool, attributed as being a good engine to work with for beginners. 1

material Materials describe the appearance of a given model by determining which textures, or sometimes which shaders to use. 24

mesh A mesh is a data structure which represents a 3D model of an object using vertex positions and attributes.. 2, 3, 8, 11, 12, 16, 18, 24, 28

OpenGL OpenGL (Open Graphics Library) is an open-source, cross-platform graphics API for use in rendering 2D and 3D graphics. 1

rasterize Rasterization is the process of converting mesh primitives into pixels. 3

render pipeline A render pipeline is a model which describes the steps taken to render a 2D or 3D scene to the screen. 1–3, 13, 17, 20–23, 25

Scratch Scratch[14] is a visual scripting language designed to teach children how to program. 3, 5, 6

shader Shaders are platform and hardware agnostic programs which run on the GPU. 1–3, 5, 6, 8, 11, 12, 16–18, 20–22, 24, 28

texture Textures are images which can be uploaded to the GPU. 3, 11, 18, 21

uniform Uniforms are per-shader global variables. 2, 3, 8, 9, 12, 16, 20, 24

Unity Unity Engine[20] is a game development tool designed with many beginner friendly features and a wide array of supported platforms. 1–6, 20, 21, 29

Unreal Unreal Engine[8] is a game development tool commercialised by Epic Games. It is generally considered to have more advanced graphics than many of it's competitors, making Unreal a good choice for photorealistic games. 1–5, 21, 22

variant Variants are a type of C++ data structure which represents a value which may be one of a pre-defined number of types. These are generally considered the more type safe version of anys. See: CPP-Reference. 8

visual script Visual scripting is a method of programming via GUI elements, rather than lines of code. This is commonly associated with node-based editors like those in Unreal Engine's blueprints. 1–6, 20–22, 28

Wavefront .obj The Wavefront *.obj* format is a commonly used file format for mesh geometry (often used or supported in 3D graphics applications like Blender and Unity). 16, 17

Acronyms

API Application Program Interface. 8

glsl OpenGL Shader Language. 5, *Glossary:* glsl

GPU Graphics Processing Unit. 3, 18, 28, *Glossary:* GPU

hlsl High-Level Shader Language. 5, *Glossary:* hlsl

HUD Heads-Up Display. 4

IMGUI Immediate-Mode GUI. 23, *Glossary:* IMGUI

OOP Object-Oriented Programming. 4, *Glossary:* OOP

RMGUI Retained-Mode GUI. 23, *Glossary:* RMGUI

SSBO Shader Storage Buffer Object. 12, 24, *Glossary:* SSBO

UI User Interface. 3, 7, 20, 23, 24