BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

PROCEDURAL TEXTURE GENERATOR

GENERÁTOR PROCEDURÁLNÍCH TEXTUR

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **ILYA DOROSHENKO**
AUTOR PRÁCE

SUPERVISOR **Ing. TOMÁŠ MILET, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačové grafiky a multimédií (UPGM) Akademický rok 2021/2022

Zadání bakalářské práce

Student: **Doroshenko Ilya**
Program: Informační technologie
Název: **Generátor procedurálních textur**
Procedural Texture Generator
Kategorie: Počítačová grafika

Zadání:

1. Nastudujte techniky procedurálního generování. Prozkoumejte stávající řešení umožňující tvorbu procedurálních textur.
2. Navrhněte nástroj umožňující tvorbu procedurálních textur a export shaderů.
3. Implementujte navržený nástroj.
4. Proměřte a zhodnoťte.
5. Vytvořte sadu příkladů a demonstrační video.

Literatura:

- Tricard, Thibault and Efremov, Semyon and Zanni. Procedural Phasor Noise. ACM Transactions on Graphics, 2019.
- Lagae, Ares, et al.: A survey of procedural noise functions. *Computer Graphics Forum*. Vol. 29. No. 8. Oxford, UK: Blackwell Publishing Ltd, 2010.

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 a 2 a kostra aplikace.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Milet Tomáš, Ing., Ph.D.**
Vedoucí ústavu: Černocký Jan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2021
Datum odevzdání: 11. května 2022
Datum schválení: 1. listopadu 2021

Abstract

Textures are an essential part of modern 2D and 3D rendering. The most prominent texturing techniques are Texture Mapping and Procedural Generation. Both techniques have their set of demands on computational resources. Procedural generation provides rich detail resolution without memory consumption, but also requires processing power. Texture Mapping is quick to process, but images, that the texture consists of are using a lot of space in memory and have finite resolution and sets of complications around algorithms that try to overcome that problem. This thesis discusses techniques of texture generation their interchangeability and applications. Result is an extensible application, that can produce texture maps from algorithms and can export algorithms to be used in procedural shading techniques.

Abstrakt

Textury jsou nezbytnou částí současného 2D a 3D renderováni. Běžné techniky jsou Mapování Textur a Procedurální Generování. Obě techniky mají svoje požadavky na výpočetní zdroje. Procedurální generování poskytuje vysokou kvalitu obrazovky bez použití paměti, ale je výpočetně náročné. Mapování Textur je rychlé, ale obrázky, ze kterých se skládají textury jsou náročné na paměť a mají omezené rozlišení. Tato práce probírá techniky vytváření textur, jejich výměnu a použití. Výsledkem práce je aplikace, která je schopna vytvářet textury pomocí algoritmů a dovolí exportovat algoritmy pro použití v procedurálním vykreslování.

Keywords

Procedural Textures, C++, textures, shading, GLSL, Qt, noise simulation, shader compiler

Klíčová slova

Procedurální Textury, C++, textury, stinování, GLSL, Qt, simulace šumů, kompilator shaderů

Reference

DOROSHENKO, Ilya. *Procedural Texture Generator*. Brno, 2022. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Tomáš Milet, Ph.D.

Procedural Texture Generator

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Dr. Tomaš Milet The supplementary information was provided by Ing. Tomaš Starka I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Ilya Doroshenko
May 17, 2022

Acknowledgements

I would like to express my gratitude to Dr. Tomaš Milet and Ing. Tomaš Starka for inspiring me to work in field of graphics and helping me make this thesis with technical advises.

Contents

Chapter 1

Introduction

Modern graphics is composed of many different layers and the point is to deliver a believable picture, which may resemble reality. It may be stylized or simplified to pixels, or complex and photo-realistic. Textures are the essential part of 2D and 3D graphics, because they store information about drawn object. For example surface material information, which is then processed according to light and view position.

Scene objects like brick walls, gravel roads and even grass are usual to the real world, but delivering detailed representation of them is challenging for computers. As for gaming industry, even in modern days of Ray Tracing and high computational powers of graphics cards, the task is to render a game with 60 frames per second in real time, so the complexity of non-focused objects is often lowered.

Filming industry has an advantage over gaming, it can allow itself a greater rendering quality, because graphics is not required to be rendered in real time, so the approach of modelling detailed objects and texturing them „by hand“ is a viable option. It is expensive for gaming on both financial and computational sides, so various techniques, like bump, normal mapping and shadow occlusion are used to express details on surface, without a need to model them.

Drawing textures has a disadvantage - it is hard to make images repetitive. So to model an object like stone road, the whole road needs to be drawn to an image, which is not always possible.

Generating textures is another approach at creating textures, it uses algorithms, noises and shapes to create an image, which is then baked and used in rendering. Those textures are often exported in tiles, that are repetitive and easy to apply.

Texture maps, generated or drawn, have major issues when it comes to application, although they are sampled to the size of an object and resized to match quality, they still have a finite scaling capabilities. There is another approach, which takes the quality of generating textures and does it in time of rendering, which results in better resolution and less intake of memory to store images.

The product of this thesis is an application, that allows the user to produce textures by manipulating computational blocks via a simple user interface. Blocks are filters like Blur or Warp and basic operations like multiply or subtract. Also it provides interface for creation of those blocks and allows user to export a texture not only as an image, but as a code of algorithm, consumable by modern graphics engines. Chapter 3 gives an insight on theory behind the notable parts of the problem. Chapters 5 and 4 are primary focused upon building an application, and explains how it works internally. Lastly chapter A shows the results of experiments using working application.

Chapter 2

Existing Solutions

There are quite a few programs, that does some form of procedural graphical processing. The most notable examples of those are Allegorithmic Substance Designer with its later incarnation Adobe Substance 3D Designer[1] and Blender[2] with its PBR material workflow. Shadertoy[3] is a first thing, that comes to mind when thinking of shader creation and procedural texturing.

2.1 Substance 3D Designer

Substance 3D Designer is an application intended for creating 2D textures, materials, filters and 3D models in a node-based interface, with a heavy focus on procedural generation, parametrisation and non-destructive workflows. It is the longest-running application in the Substance 3D ecosystem and resources made with it are the most versatile and dynamic possible.[4]

[1] https://www.substance3d.com/

[2] https://www.blender.org/

[3] https://www.shadertoy.com/

[4] Adobe. Overview | Substance 3D Designer. Available at: https://substance3d.adobe.com/documentation/sddoc/overview-129368161.html

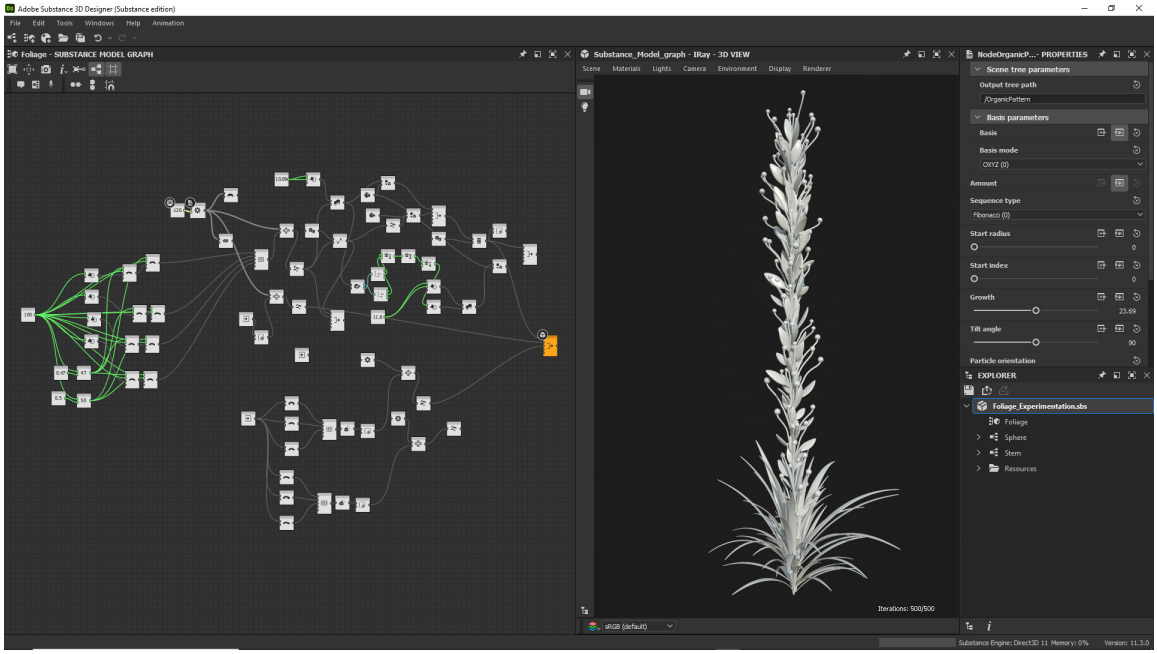


Figure 2.1: Adobe Substance 3D Designer

It provides a powerful tools for generating an industry standard textures, graphs, models, etc. But the main strength becomes a great weakness - it is quite hard to get accustomed to, it requires years to master and its features are proprietary. Also the program is quite expensive coming at 50 dollars a month. It has a large library of nodes, but most of them are closed-source and cannot be used anywhere else, but in the program itself.

Exporting complex graphs and textures is a great option, it does not provide any source code export, so procedural textures are only exportable as images or models, although some graphical engines are capable of interacting with substance designer format directly, such as Unreal Engine 4, Unity or Blender.

2.2 Blender

Blender is a free and open-source 3D computer graphics software toolset used for creating animated films, visual effects, art, 3D-printed models, motion graphics, interactive 3D applications, virtual reality, and, formerly, video games. Blender's features include 3D modelling, UV mapping, texturing, digital drawing, raster graphics editing, rigging and skinning, fluid and smoke simulation, particle simulation, soft body simulation, sculpting, animation, match moving, rendering, motion graphics, video editing, and compositing.[5]

[5]Wikipedia. Blender (software). 2022. Available at: https://en.wikipedia.org/wiki/Blender_(software)

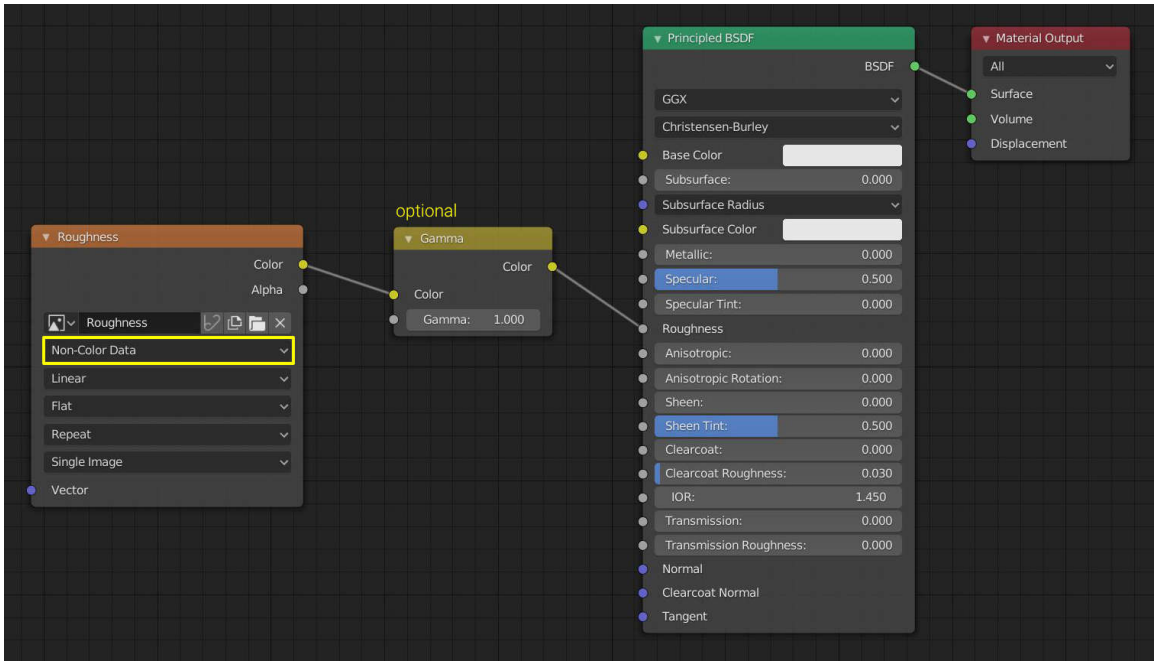


Figure 2.2: Blender's Principled BSDF shader for PBR material

It is open source, free and powerful in creation of 3D models and textures. In all of its glory it's also hard to master and work with. Its primary target is 3D rendering, and texture creation is a thing that goes by. Mostly, nodes are used to edit existing textures, working as filters for 2D and 3D textures.

But it lacks a possibility of exporting shaders for procedural texturing. Nodes may be written for further extension, but the language they are written in is not supported by graphics cards, making them possible to be used only in blender.

2.3 Shadertoy

Shadertoy.com is an online community and platform for computer graphics professionals, academics and enthusiasts who share, learn and experiment with rendering techniques and procedural art through GLSL code. There are more than 52 thousand public contributions as of mid-2021 coming from thousands of users. WebGL allows Shadertoy to access the compute power of the GPU to generate procedural art, animation, models, lighting, state based logic and sound.[6]

[6]Wikipedia. Shadertoy. 2022. Available at: https://en.wikipedia.org/wiki/Shadertoy.

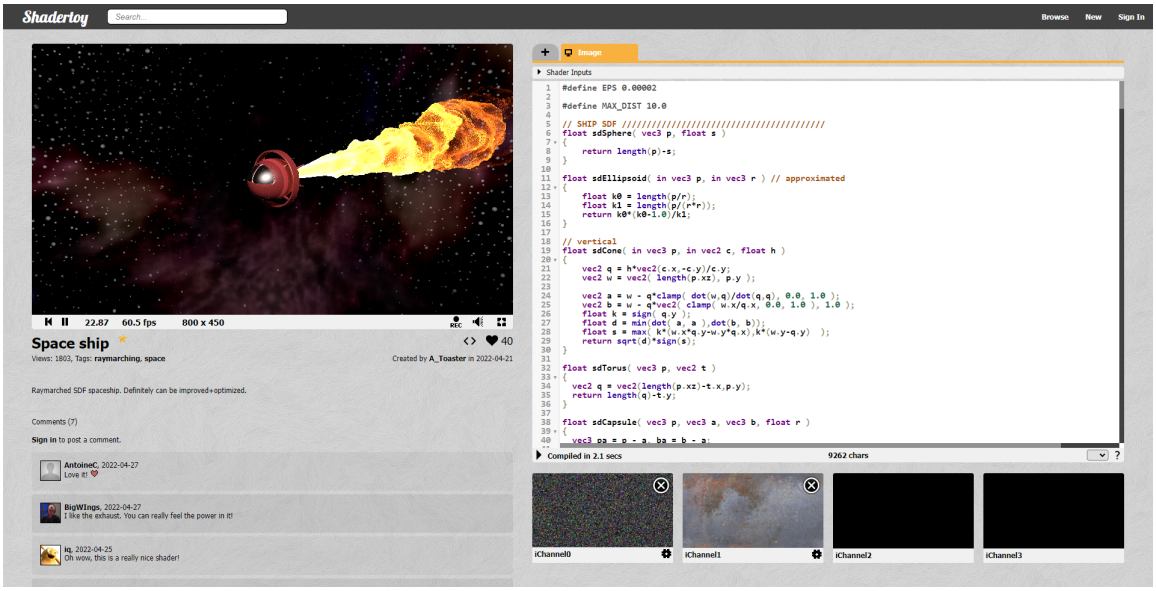


Figure 2.3: Shadertoy web page.

Shadertoy is focused on development of shaders for procedural texturing. Shaders are written in GLSL and can be used in Web and in Open GL driven engines with minor adjustments. But it lacks code re-usability in some form; to reuse code, it must be copied from somewhere else, it is not processed or optimized.

2.4 Summary

All the above applications are capable of some form of procedural texture generation, however in some way there is a lack of functionality, that can be added. Most of the professional texture applications are intended to be used as a static material export in form of images, however there is a way to describe a surface material in form of an algorithm. Those forms can be interchanged, depending on the demands, and the application, that is being described in this thesis is capable of producing both an image and an algorithm.

Chapter 3

Theoretical Background

The following sections describe the working aspects behind the project. Those involve explanation of the algorithms used within the application as well as the main idea of the project.

3.1 Graphics and Textures

The word texture does not have a precise definition behind it, it is often described as the way the object looks or feels. In computer graphics texture often means mapped texture or bitmap, a 2D or 3D array that contains some information about object, such as roughness or color. So the texture mapping is a way of adding some detail to the surface of drawn object. There are two popular approaches, first there is a mapped image, so called raster textures - a way of projecting an array to the surface of polygons. And there is other approach, which will be discussed extensively throughout this thesis - an algorithmic approach. Unlike a pixel texture, a procedural texture describes the texture mathematically. Although it is not commonly used, the method is resolution independent and can create more precise textures, especially if there is great and varying depth to the objects being textured. The approach is called procedural texturing.

To describe how procedural texturing works, we need to describe how graphics works. Modern graphics pipeline is divided into multiple stages, some of them are programmable. Because of the programmable stages of the pipeline it is possible to describe the behaviour of the concrete objects, described by stages. Because of the modern graphics, that is focused around high resolution of the details, we are able to assign an algorithm, that will be executed for each pixel on the frame.

A Shader is a user-defined program designed to run on some stage of a graphics processor. Shaders provide the code for certain programmable stages of the rendering pipeline. They can also be used in a slightly more limited form for general, on-GPU computation.[5] The texture mapping happens at pixel shader stage, calculations there are performed for each on-screen pixel. Per-pixel information may be extracted from input texture using sampler or calculated in runtime. The topic of this thesis uses the latter approach to make creation of the algorithmic textures more feasible.

3.2 Texture Generation Algorithms

There are a lot of algorithms, that can be used to create textures. Also there are a lot of classifications for those algorithms, but for the thesis the division will be the following:

- Generator Algorithms or Generators - the algorithms, that create something using parameters, supplied to them. The example may be a noise generation with parameters being a Seed[1] for pseudo-random noise, or a shape generation, like a circle, with parameter being the radius.
- Filter Algorithms or Filters - an operation, that is focused on modification of the output value of another algorithms, that may also depend on parameters. The most prominent example is Blurring, but the set also include operations like addition, in which first value is modified by an addition to other value.

Combining them in different ways may result in texture that closely resemble real materials, such as wood or stone. The following sections contain explanations of algorithms that are common and used within application.

3.3 Noise generation algorithms

As was previously mentioned, there are several ways of generating an image, one of them is creating a noise. In computer graphics noise is a sequence of pseudo-random values. For visualization often two-dimensional texture representation is used. For that case noise is usually generated by a function, which takes N-dimensional normalized coordinates as an input and returns a floating-point value, typically in range from 0 to 1.

3.3.1 White Noise

White noise is the most primitive application of this function, each sample is generated uniformly random and output value does not depend on neighbour samples. For example for 2D image the algorithm output value depends on pixel coordinate and a seed, used in pseudo-random number generator, typically of uniform distribution.

White noise generator does not provide natural look to a texture, so it is rarely used on its own, but it has uses in stylized output(e.g. TV screen).

3.3.2 Perlin Noise

Lattice noises are the most popular implementations of noise for procedural texture applications They are simple and efficient and have been used with excellent results.[2] One of the most prominent examples of lattice noises is Perlin noise. It was released by Ken Perlin in 1985 and was used in film industry.

The Algorithm is described in three steps:

1. Define an N-dimensional integer grid of pseudo-random gradient vectors which lie within a unit sphere and have components between -1 and 1, any vector, whose length is greater than 1 is discarded, others are normalized to unit length.

[1] Seed - a number used to initialize random number generator

2. For each point in lattice select all of its 2^N neighbours, namely corner points, find a dot product of a lattice gradient and fraction part of the corner point relative to selected lattice point for each corner point.

3. Apply smoothed N-linear interpolation to get result value.

3.3.3 Simplex noise

Simplex noise is an algorithm, that is built upon Perlin noise, which improves its capabilities, reduces computational cost for higher dimensions and has less visible artifacts, that its predecessor. It was developed by Ken Perlin and was released in 2001 with article coming the next year.[6]

The N dimensional space is subdivided into equal simplest shapes, that can be repeated to fill all the space. For a one-dimensional space, the simplest space-filling shape is intervals of equal length placed one after another, head to tail. In two dimensions, the simplest shape that tiles a 2D plane is a triangle, and the formal simplex shape in 2D is an equilateral triangle. Two of these make a rhombus, a shape that can be thought of as a square that has been squashed along its main diagonal.

Perform a summation of contributions from each corner, where the contribution is a multiplication of the extrapolation of the gradient ramp and a radially symmetric attenuation function. In signal processing terms, this is a signal reconstruction kernel. The radial attenuation is carefully chosen so that the influence from each corner reaches zero before crossing the boundary to the next simplex. This means that points inside a simplex will only be influenced by the contributions from the corners of that particular simplex.[4] The graphical explanation is presented on Figure 3.1.

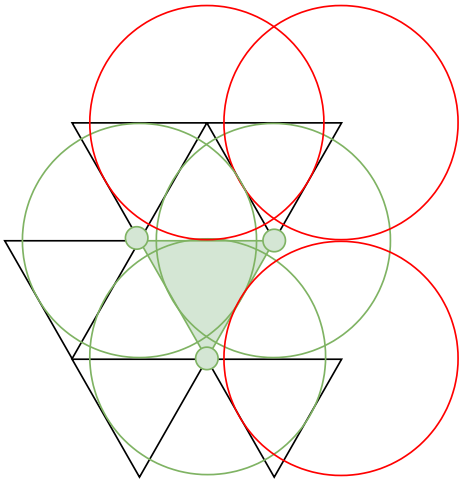


Figure 3.1: The explanation of corner contributions. The value of a point inside a green triangle is only affected by corner points of it. Attenuation function is a large circle and corner point is a small circle. Red circles with centers in adjacent points are decayed to zero, so they do not affect the result.

To find out what simplex is selected for the current point the coordinate axes are skewed in a non-uniform way. They are transformed along the main diagonal of the regular axes, forming squares out of pairs of horizontal triangles. After that step an integer coordinate is taken and correct simplex is determined. Graphical explanation is shown on figure 3.2

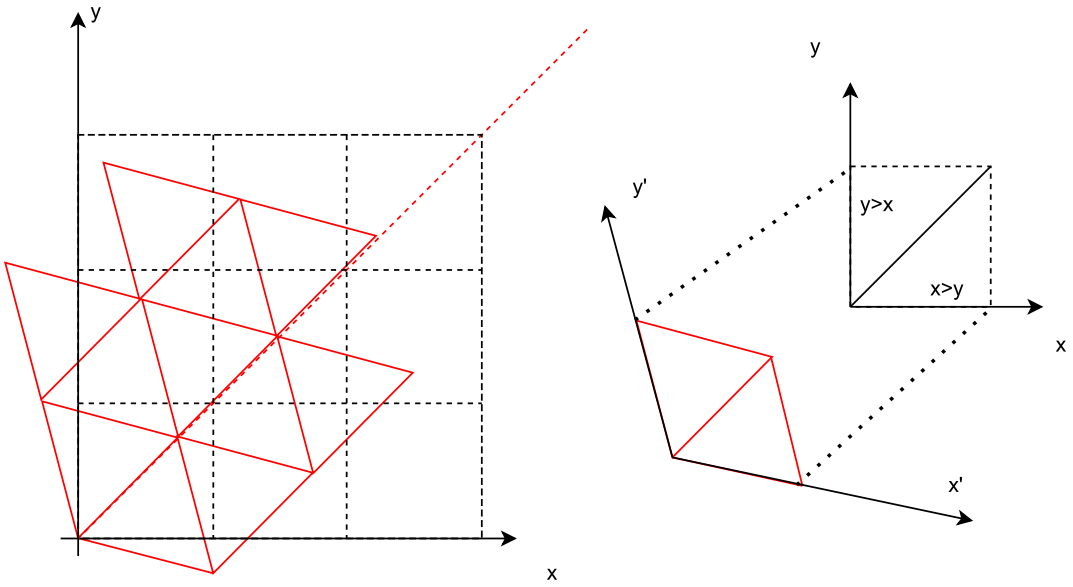
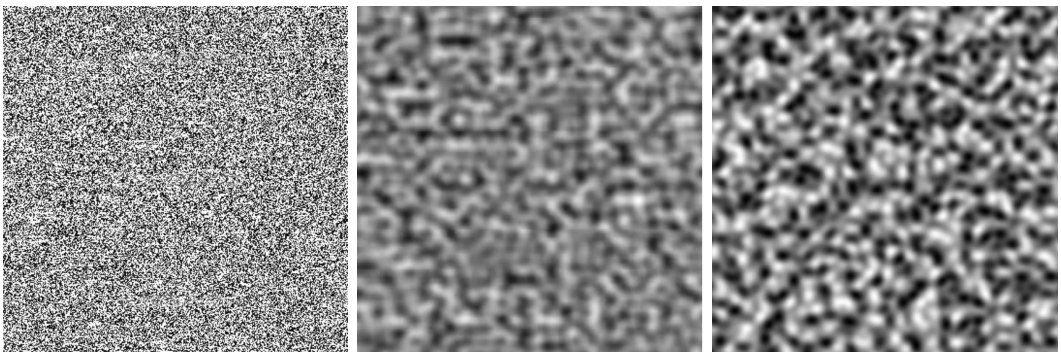


Figure 3.2: Axes transformation explanation. Two triangular shapes are transformed to the linear grid. Integer part of the transformed point coordinate will give an answer on what simplex is it in. If the x coordinate is greater, than y, the point is in the right triangle on the original grid.

The results of algorithms, described above are shown on Figure 3.3:



(a) White noise. (b) Perlin noise. (c) Simplex noise.

Figure 3.3: Results of noise generation algorithms.

3.4 Cellular pattern generation

Another approach at texture generation is making patterns. One of the most natural looking pattern is cellular pattern. It is based around so called feature points, which are essentially centers of cells.

The naive approach tells to calculate a distance between each pixel to each point. That means that we need to iterate through all the points, compute their distances to the current pixel and store the value for the one that is closest. However this approach is slow, because it requires computation of Euler distances to all of the feature points. The problem is solved

with application of the same subdivision that Perlin noise uses, this effectively reduces the number of comparisons leaving only 8 adjacent cells.

This algorithm can also be interpreted from the perspective of the points and not the pixels. In that case it can be described as: each point grows until it finds the growing area from another point. This mirrors some of the growth rules in nature. Living forms are shaped by this tension between an inner force to expand and grow, and limitations by outside forces. The classic algorithm that simulates this behavior is named after Georgy Voronoi.[9] The result of an algorithm is shown on Figure 3.4.

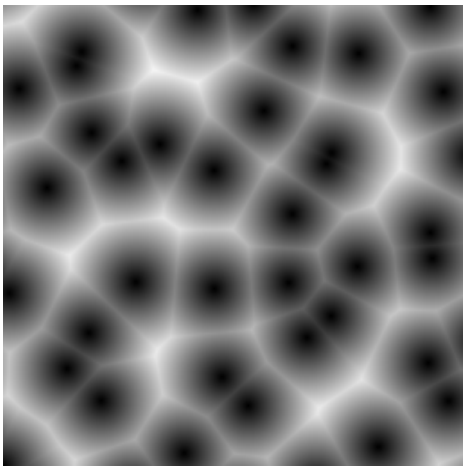


Figure 3.4: Voronoi diagram.

3.5 Fractal procedural systems

Another way of creating a natural looking procedural texture is trying to recreate another property of living organisms. Plants are self similar by nature, but are quite complex to express. To make it bearable Lindenmayer System or simply L-system can be used. In 1968 Aristid Lindenmayer created grammar to represent simple multi-cellular organism.

Self-similarity relates plant structures to the geometry of fractals. The technique is based on rewriting rules, so on replacing sub-terms of a formula with other terms. All formulas and terms are represented as sequences of characters. Every system has to have starting word, called axiom, rules of productions and number of iterations, so how many times rules must be applied to word.[7]

The L-system can be formally defined as a tuple $G = (\Sigma, \omega, P)$, where Σ is the alphabet of the system. It contains mutable and immutable elements. Mutable elements are called variables and immutable are constants. ω is a starting string or axiom and P is a set of rules, that define a way that variables are unwrapped using combined variables and constants. Rules are usually written in a before-after succession, where to the left of an arrow there is a predecessor and to the right there is a successor string. For example $A \longrightarrow BC$, A is a variable and BC is a successive string.

The simplest form of an L-system is context free 0L-system. IF there is only one replacement to the same character, the system is called Deterministic or D0L-system.

For example: is there is an axiom a, rules are set as $a- > b$ and $b- > ba$ we shall get endless sequence of unwrapping words, that form a repetitive pattern, shown on Image 3.5.

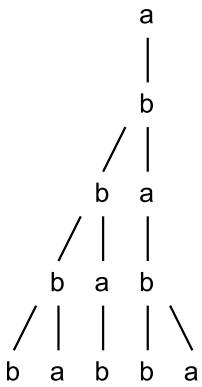


Figure 3.5: D0L-system example. 4 iterations are preformed on each character, forming the sequence *babba*.

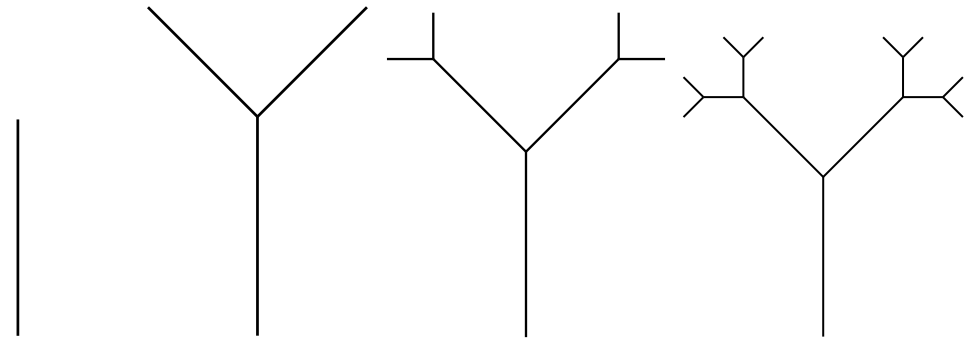
If the symbols are given special meaning in graphics, it becomes possible to apply a set of operations, that results in image to be drawn. The term is called Turtle graphics and it is essentially vector graphics with relative cursor.

For example, G_1 will define Σ as a, b as variables and [,] as constants, ω is a, and the rule set P as $a \longrightarrow b[a]a$ and $b \longrightarrow aa$.

Applying the rules we get:

- axiom: a
- 1st application : b[a]a
- 2nd application : bb[b[a]a]b[a]a

Let the cursor have a stack of pair - position and angle and a variable line segment length, then let [mean push(save) position and angle and turn 45 degrees to the left,] means pop(return to) previous position and angle and turn 45 degrees clockwise. b stands for draw line segment while a means decrease its length by a half and draw line segment. Application of those definitions for the cursor results in images shown below on Figure 3.6.



(a) Axiom. (b) First iteration. (c) Second iteration. (d) Third iteration.

Figure 3.6: Results of interpretation of several applications of rules, defined by an D0L-system. The pattern is endlessly applicable, resulting in a tree-like shape.

3.6 Flow Graph

The major part of the program is revolving around a concept of a flow graph. A graph is formed by vertices and by edges connecting pairs of vertices, where the vertices can be any kind of object that is connected in pairs by edges. In the case of a directed graph, each edge has an orientation, from one vertex to another vertex. A path in a directed graph is a sequence of edges having the property that the ending vertex of each edge in the sequence is the same as the starting vertex of the next edge in the sequence; a path forms a cycle if the starting vertex of its first edge equals the ending vertex of its last edge.[8] A directed acyclic graph is a directed graph that has no cycles.

The data-flow graph is an directed graph, nodes of which represent operations on data, including creation and edges represent paths, that data travels through the system. The best example is conveyor, on which parts are combined into a car. The particular acyclic variation will be used to implement basic texture combination. The example of it is shown on Figure 3.7

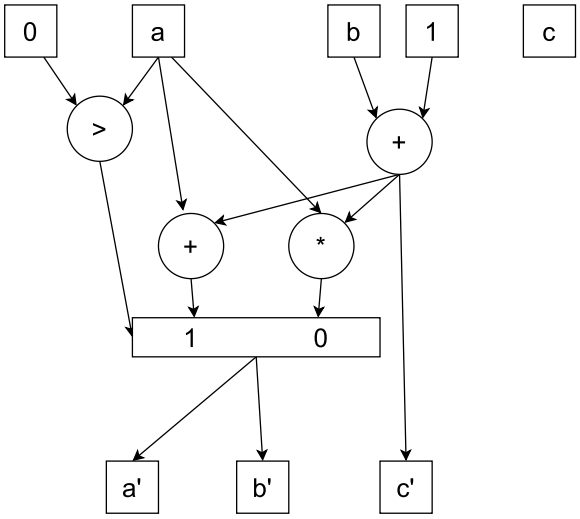


Figure 3.7: Example of a data-flow graph.[2]

A basic block N of Texture editor is a function code, that has zero or more inputs I to it, representing input data, which comes to a function and a set of outputs O, that represent return value of the function. $N \in \{I, O\}$, where $I \in \{i_1, i_2, ...\}$ and $O \in \{o_1, o_2, ...\}$.

A directed graph may be represented as $G = (N, E)$, where N is a set of nodes, and E is a set of directed edges $E \in \{(\{n_1, o_1\}, \{n_2, o_2\}), ...\}$, indicating, that edge comes from first node's output 1, to second node's input 2. Thus there exists a successor function:

$$S_G(n_i) = \{n_j | (n_i, n_j) \epsilon E\} \tag{3.1}$$

that results in a set of immediate successors, connected to node n_i outputs. The reverse function

$$S_G^-(n_i) = \{n_j | (n_j, n_i) \epsilon E\} \tag{3.2}$$

defines a set of immediate predecessors of node. Results of both functions may be empty.

[2]Source: https://www.researchgate.net/figure/Data-flow-graph-example_fig2_224517165

A subgraph G' of graph G is a subset of G, defined in such a way, that $G' = (N', E')$ where $N' \subseteq N$ and $E' \subseteq E$ from graph G definition. In case of this thesis, subgraph must also include edge to immediate successor if the immediate successor is present in N'.

The graph may be dissected from the perspective of the selected node n_i into two parts, namely pre sequence and post sequence. Pre sequence is a subgraph, that include immediate predecessors of A $S_G^-(n_i)$ and recursively all predecessors of theirs.

$$P_G^-(n_i) = B \cup \left(\bigcup_i P_G^-(b_i) \right) \tag{3.3}$$

where $B = S_G^-(n_i)$ and $b_i \in B$.

On the other hand post- sequence is defined as:

$$P_G^+(n_i) = A \cup \left(\bigcup_i P_G^+(a_i) \right) \cup n_i \tag{3.4}$$

where $A = S_G(n_i)$ and $a_i \in A$. Note, that n_i is included into the post sequence, and to define a primary rule of program graph it is necessary to state $P_G^+(n_i) \cap P_G^-(n_i) = \emptyset$.

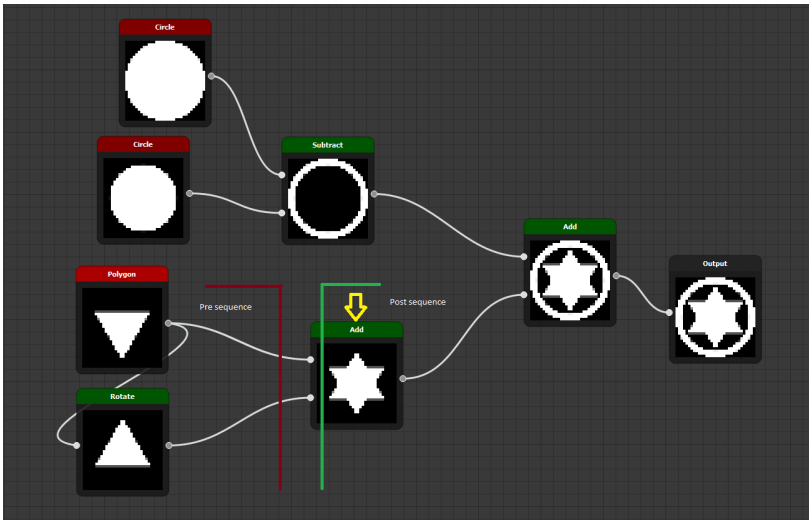


Figure 3.8: Pre and post sequences of a single node.

As shown on Figure 3.8 for selected node, pointed on by the yellow arrow, every node, that comes before red line and connected to it is a pre sequence, and after green line, including node itself is post sequence.

3.7 Graph traversal algorithms

Graph is a data structure, but to gather useful information out of it algorithms must be defined. One of the most used operation on graph is graph traversal. Graph traversal refers to the process of visiting, each vertex in a graph exactly once. This thesis will not dive into complex graph algorithms, as the elementary ones will suffice. The most common traversal algorithms are Depth-First search(DFS) and Breadth-First search(BFS).

BFS is a graph traversal algorithm and traversal method which visits all successors of a visited node before visiting any successors of any of those successors.[1]

The algorithm is great for traversing all nodes and gaining information from it.

DFS differs from a breadth first search in that the exploration of a vertex v is suspended as soon as a new vertex is reached. At this time, exploration of the new vertex u begins. When this new vertex has been explored, the exploration of u continues. The search terminates when all reached vertices have been fully explored. This search process is best described recursively. DFS uses a strategy that searches deeper in the graph whenever possible.[1]

It may be described like this:

1. Start by visiting the root node.
2. For each successor do: if a vertex is not visited, then invoke the algorithm on that vertex.

The change in order of visiting may lead up to different types of traversals. The example of depth first traversal is shown on Figure 3.9.

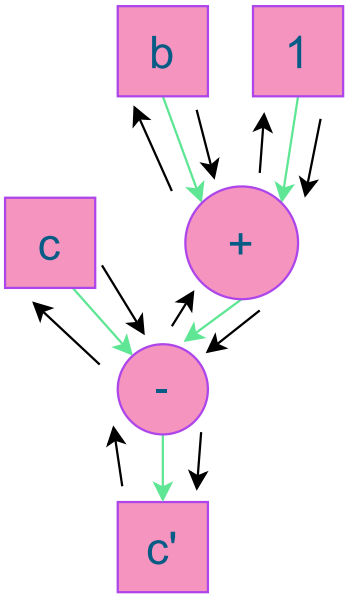


Figure 3.9: Example of DFS. The path is (c')->(-)->(c)->(+)->(b)->(1). This results in a prefix notation c' = -(c,+(b,1)), which may be translated as c' = c-b+1.

3.8 Lexical analysis

One of the main goal of a thesis is to compile a data-flow graph into a code, usable by applications. Nodes themselves also consist of code, for that matter a compiler needs to be introduced.

A compiler in simple words is system software that converts a source high-level programming language program into a target language program. The first stage of the compiler is the Lexical analysis or lexer for short. It scans the source program and produces meaningful sequence called Tokens. It has a lot of tasks, that include:

- It stores the tokens in symbol table and sends them to next phase as the input.

- Lexical analyser also cleans the source program, strips off blanks, tabs, newlines and comments.
- It keeps track of various errors in every phase and its associated line number in which error has been raised.[3]

The example scheme of lexer is depicted below on Figure 3.10

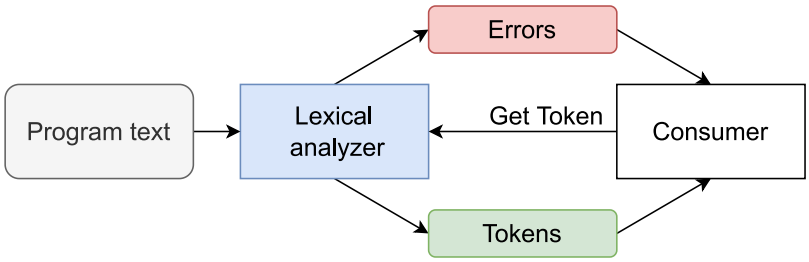


Figure 3.10: Example lexical analysis interaction with consumer. Lexer gets a program text to its input, and on each request from the consumer it produces tokens from the input. In case any error occurred, lexer informs consumer of it as well.

Token is a simple structure, that assigns a logical meaning to a sequence of characters. The meaning is assigned by the rules of language. The token values usually mean:

- identifiers
- keywords
- literals
- operators
- separators

Lexical analysis is essentially an algorithm, that picks characters from the input stream and matches them against language rules, for example floating point literal may contain a sign character at the beginning, a dot, which separates fractional part from the whole. Those rules are usually implemented using Finite state machines.

One of the uses of the Lexer lies in syntax highlighting. The highlighting is possible, because tokens are given a meaning during analysis. This meaning, with side information given, like position and length of the lexeme makes a viable point for highlighter to be a consumer of tokens.

Chapter 4

Application design

The application is divided into three major parts. Those parts should allow user to manage texture creation with relative ease of use. Interface should be convenient and easy to use. Those parts being Texture Designer, Node Compiler and Texture exporter. One of the main ideas is to make application multitasking reflect into user interface. For such a demand a browser layout is fitting pretty well, considering its tab view. Such interface will be easy to manipulate. The design idea is shown on Figure 4.1.

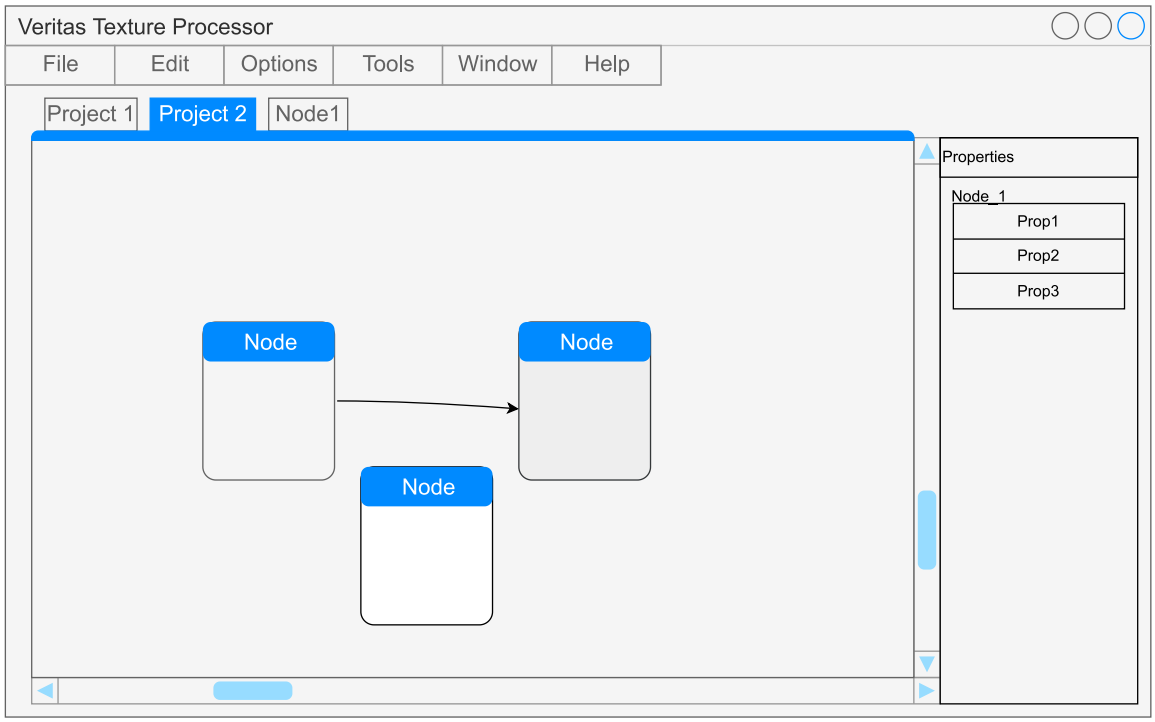


Figure 4.1: User interface design. The UI consists of several pieces. Top part is represented with a menu bar, that holds options, that control the project. In the center there is a tab widget, that holds multiple projects and allows easy switching between them. On the tab there is a scene with nodes, that form flow graph. Properties window is situated to the right of the screen and it holds properties for selected nodes, that are highlighted on the canvas (bottom one). The scene is dragable, meaning the user may roam around the graph and zoom in on parts of it.

In the center there is a graphics scene with the nodes, that contains flow graph. To the right there is a property window, that holds all the parameters for the selected node.

4.1 Texture Designer

Texture Designer is a simple interface, primary purpose of which lies in creation of sequences of patterns and filters, that results in desired image. Patterns and filters are encapsulated inside simple, yet powerful abstraction called Node. More on that in subsequent section 4.2. Sequences are described making an oriented graph of nodes. Graph, composed by the nodes, should be acyclic, to ensure that, a regular Depth-first search algorithm can be used.

Texture designer fits the concept of a flow graph of nodes, representing values and operations on them. Nodes are highly dynamic and require a highly adaptive user interface, that shows possibilities of node output, but also as simple as a toy constructor.

4.2 Node

Node is a simple abstraction around complex structure, composed of algorithm and parameters, supplied to it. Because this element makes a large portion of the work, it should be flexible. It should allow user to create different types of nodes, as well as edit once, that are present. The structure is a subject of change, so the main application goal of expansion is to provide a convenient way to create own implementations and types of nodes, store them and load others creations, forming a library, that expands with new variations and algorithms. Some of the algorithms are forming the procedural texture library, which is explained in its own section 4.6. The core of the application will be made using a node, that is capable of processing shaders and output an image called shader node.

Each node has a set of inputs, set of outputs and two sets of described operation sequences. Node inputs, also known as sinks, can be used only once, and outputs, namely sources, may be used with zero or more nodes in post sequence. Each node port, input or output, manipulates with images.

Both sinks and sources may have types, or use raw values from before. Typed sink may not be used with source of other type, but they both may be used in raw value throughput. Nodes hold an image of the operation, that can be exported at any time.

Nodes may have a set of modifiable properties, that internally regulate some values, resulting in different output, e.g. seed value for noise generator, or a radius of the circle.

4.3 Node Editor

Node Editor forms a connection of coding and graphical parts of the program. It allows user to create own shader nodes using GLSL language and test its capabilities right away.

Its interface is divided into three part, that are situated one beside other. They are Text editor, Graphics scene and Properties windows.

Text editor part is supplied with syntax highlight tool, which makes text easier to read. This highlighter works off of non-allocating code parser,that allows relatively fast performance on highlighting.

The editor subpart allows testing the node capabilities in real time, if the node is successfully compiled, it can be attached to other nodes from the node library via sinks and sources. Node can also be exported in a application-specific format and used later in other

projects. The properties subpart makes possible the property modification as well as port assignments. Those variables may be later referred to in code.

4.4 Texture exporter

One of the crucial parts of the project is texture export. The texture can be exported as an image, as was stated in the previous chapters. But the graph may also be interpreted as an algorithm, and can be translated into procedural code, that is exported as a GLSL code. The resulting code may then be used in any OpenGL application or game engine.

The translator should go through the nodes, collecting code and data within the node, unpack the data into the code and then it should create a shader with information from all the nodes with possibility of code optimizations. For that purpose a slightly modified Depth-first traversal is used. The module should provide several actions to it, namely export as image, export as unrefined GLSL code and export as optimized GLSL code.

4.5 Lexical analysis

One of the major components of the program is a custom lexical analysis tool, created for syntax highlighting of a GLSL code. It should be capable of parsing through the code in real time. The requirements on parsing capabilities however are not that strict, meaning that not the whole language is required to be implemented. A defined subset of the language will suffice. The subset consists of basic keywords, basic literals, vector types and functions to them. The set does not include uniform keyword, no samplers and no invariant and no varying keywords. Lexical analysis in a periodical form should identify types, macros and function declarations. But it is not require to be a full-fledged compiler.

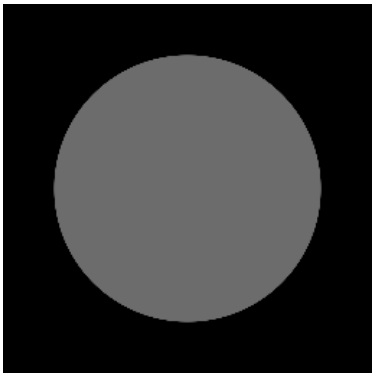
4.6 Procedural Texture Library

Node editor has to be supplied with some amount of preinstalled nodes. Nodes themselves work with images and have a simple shader code inside. This should be a rule for the most blocks within the library and newly created ones and should have variables to be adjustable in the editor. The most prominent are shapes, they are generated using polar space subdivision and are simple to work with. Examples are found on Figure 4.2.

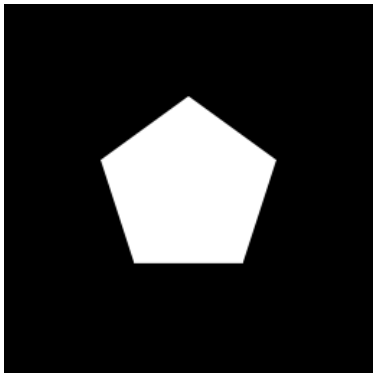
4.6.1 Sine wave generator

This generator works with math sin function. It processes position of the pixel in a normalized[1] space and brings back a grayscale value of pixel color. The function itself can be controlled with variables supplied into node. For example function can be lifted up or down, increasing or decreasing its value by a fixed number, or it can be stretched or shrunk, depending on what count of periods is supplied. Also this generator has controls that rotate, multiply value of the function, change function position by moving coordinate left or right. Examples with numbers are shown on Figure 4.3:

[1] Normalized - meaning coordinates are mapped between 0 and 1



(a) Circle with adjustable radius and shade.



(b) Pentagon, a polygon with sides parameter set to 5.

Figure 4.2: Basic shapes example.

4.6.2 Operations

The editor is supplied with elementary mathematical operations, like addition, multiplication or subtraction. The set of operations also includes such operations as rotation and scaling, those operations work with coordinates supplied with the image, transforming them according to parameters.

Addition

Addition can be expressed as

$$F_r(x, y) = MIN(F_a(x, y) + F_b(x, y), FMAX) \tag{4.1}$$

where FMAX is maximum value of pixel (white color) and $F_r(x, y)$ is a resulting pixel. Minimum is required, because color in graphics lies within bounds from (0,0,0) to (1,1,1), greater values do not make sense and may become a problem in later operations.

Subtraction

Subtraction is opposite

$$F_r(x, y) = MAX(F_a(x, y) - F_b(x, y), FMIN) \tag{4.2}$$

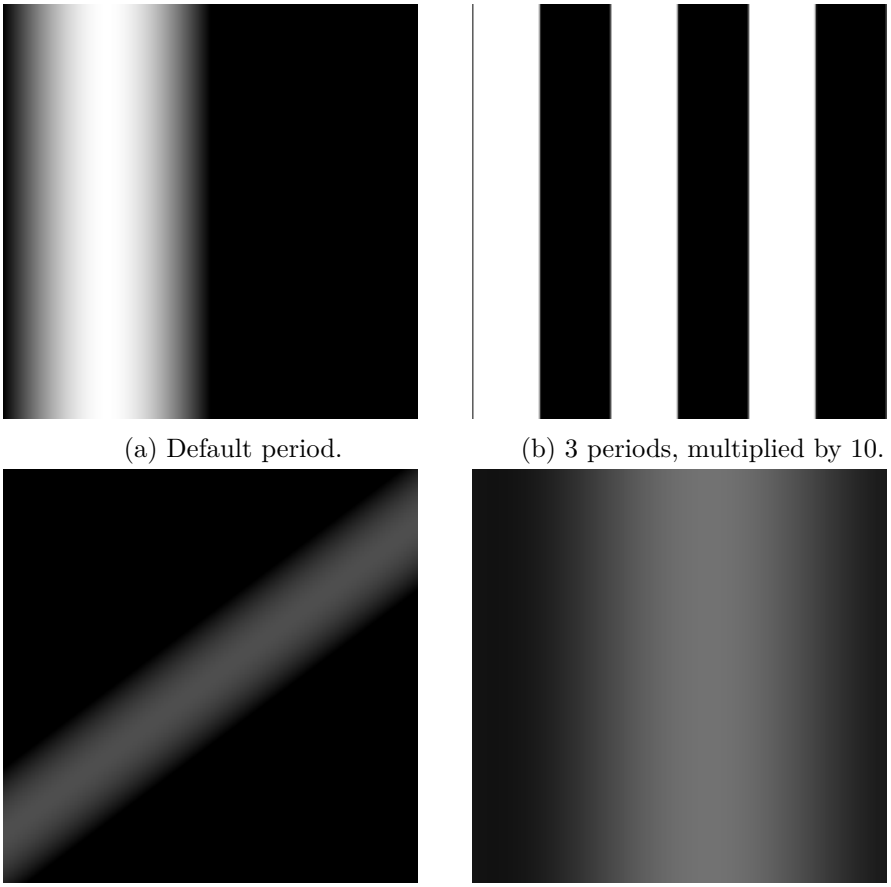
Maximum plays the same clamping role as in the addition.

Multiplication

Multiplication is the simplest of the operations, and can be depicted as

$$F_r(x, y) = F_a(x, y) * F_b(x, y) \tag{4.3}$$

It does not require any clamping, because values would never exceed 1 due to value range of a pixel lying between 0 and 1.



(a) Default period.

(b) 3 periods, multiplied by 10.

(c) Rotated by 45 degrees and lowered.

(d) Displaced by Pi

Figure 4.3: Sine Function results.

Rotation

Rotation is applied around the center of an image. The operation creates a rotation matrix from a supplies angle α,

$$A = \begin{pmatrix} cos(\alpha) & -sin(\alpha) \\ sin(\alpha) & cos(\alpha) \end{pmatrix}$$

The texture coordinates are shifted to the center, then the rotation transformation is applied to it, then coordinates are shifted back to the beginning of the image.

Scale

Scaling is also applied from the center, the matrix applied to the texture coordinates is created from supplied parameters x and y:

$$A = \begin{pmatrix} x & 0 \\ 0 & y \end{pmatrix}$$

The application of matrix is the same as in Rotation operation.

Shift

Shifting is a simple texture coordinate translation. the operation can be expressed as $F_r(x, y) = F_a(mod(x + dx, 1.0), mod(y + dy, 1.0))$, where dx and dy are translation coefficients. Modulo operation ensures, that the coordinate is not greater than 1 and produces wrapping effect. The example is shown on Figure 4.4

(a) Before transformation. (b) After.

Figure 4.4: Transformation results by a 0.5 on the x axis.

4.6.3 Filters

The editor is supplied with some filter functions for image processing. The most prominent are a Luminance filter, that turns colored image into grayscale using a formula $F_r(x, y) = 0.299R + 0.587G + 0.114B$, where R,G and B are color components of the filtered pixel and threshold filter, coming in two variations - pure threshold which sets values below some constant to zero and a comparison filter, which sets the value of the pixel with the lesser one from two supplied images.

The definition for threshold function is

$$F_r(x, y) = step(F_a(x, y), F_b(x, y)) * F_a(x, y) \tag{4.4}$$

and for comparison is

$$F_r(x, y) = step(F_a(x, y), F_b(x, y)) * F_a(x, y) + step(F_b(x, y), F_a(x, y)) * F_b(x, y) \tag{4.5}$$

where

$$step(a, b) = \frac{sign(a - b) + 1}{2} \tag{4.6}$$

Chapter 5

Implementation

This chapter gives a insight on how things were implemented internally. The topic will be divided into several categories according to relation between program modules.

5.1 Libraries and Resources

Universal requirement for most applications is portability, to reach this goal resources are limited to only those, that can run on multiple platforms. The Texture Editor will suite PC platforms, but because of great amounts of graphical and lexical computations performance becomes a great factor amongst functionality and usability. The chosen language is C++, because it provides zero-overhead abstractions like classes, while being native compiled language, which gives a lot of control over memory and resources without sacrificing computational speed.

Qt

Qt is a cross-platform application development framework for desktop, embedded and mobile.[1] It comes with a widget toolkit for creating graphical user interfaces as well as cross-platform applications that run on various software and hardware platforms such as Linux, Windows, macOS, Android or embedded systems.

Being written in C++ results in great performance, and Widget encapsulation of windows provides convenient programming interface for user interface programming. Qt also provides handles for native graphics libraries, such as OpenGL, which can be utilized to compile shaders and produce textures.

The control version used for the application is 6.3, it has extended string literals and views support.

C++20 Standard Library

Primary language is C++, with top standard being c++2b(23), which is only partially implemented by the time of this thesis in 3 major compilers [2]. Compiler of choice is MSVC, because it is the only compiler supporting text formatting features. Primary features to be explored are ranges and string_view.

[1] https://wiki.qt.io/About_Qt

[2] Compiler support table: https://en.cppreference.com/w/cpp/compiler_support

Described features are used to make non-allocating code parsing program, which will be described later. Standard library also provides containers for efficient storage, lookup and information transfer, as well as low-level memory safety features for fast communication between program modules.

GLSL optimization library

Although briefly mentioned, the code compiler has an optimizer built into it. It is a static library, that comes from a small project called tiny GLSL optimizer[3]. It is a small library, built around Mesa GLSL compiler code, but only several parts of it were extracted, that rebuild code with HIR-tree[4]. Then the tree is traversed with visitor class, that builds up optimal code.

The code is complex, and although the project is maintained by the author of the thesis, it will be only briefly mentioned and no details of the project will be included.

5.2 Infrastructure

The application model does not resemble any particular kind of pattern. Instead it uses several types of data sets to operate information on each level independent from each other. It is depicted on Figure 5.1.

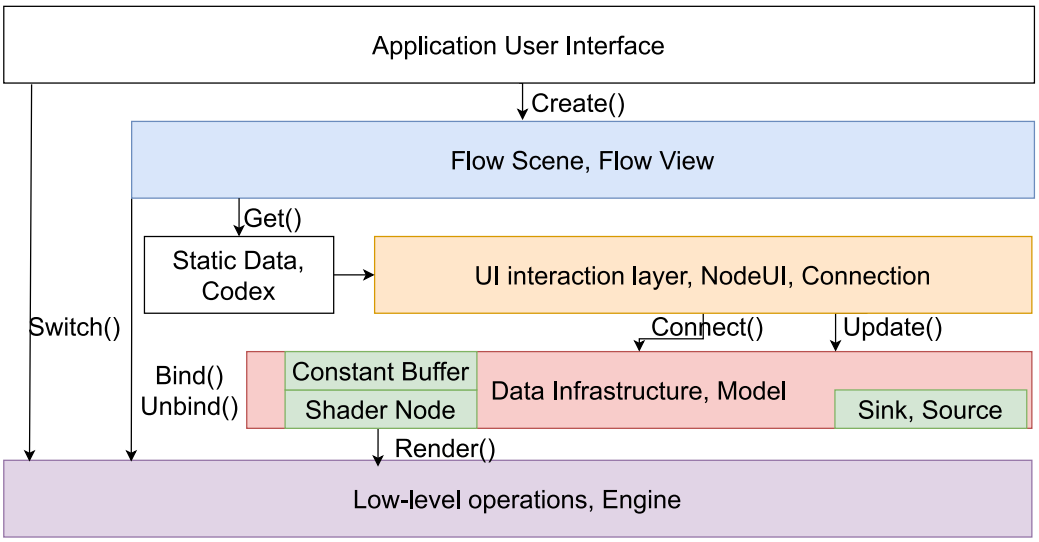


Figure 5.1: Project workflow diagram.

Engine is a single container of frame buffer objects, it is the lowest level of the model and it works with raw data and outputs images. For each scene a new frame buffer is created, and if the scene is changed, the frame buffer should be changed as well, and if scene is destroyed, the frame buffer is also erased.

The next level is a data infrastructure level. It is meant to carry data through nodes, provide interface for node connection and solve data propagation. It does not use Qt, it is lightweight and can be applied to any other data-flow node system. It was carried from

[3]Project link: https://github.com/Agrael1/tiny-glsl-optimizer
[4]High-Level Intermediate Representation

another project of mine called VeritasD3D[5], where it solved graphics multipass rendering system.

The primary purpose of infrastructural system is to provide connection between user interface and low level data. Subclasses of Node, Sink and Source provide resource disposition and data interaction to the upper level. Infrastructure also manages dynamic User Interface creation that depends on data and type the model is. The shader node implementation does a call to engine's Render function each time the update is demanded, supplying all the data required for the render, including Constant buffer, pixel shader, input textures and output textures.

Next upper sub-level is a static subsystem or codex. Codex in its essentials is a node model loader. It works with files, creating static descriptors, that are used to create instances of nodes, acting like a factory in some regard.

Codex comes in pair with user interface sub-level or UI. UI in this context means graphical representation of nodes and their parts. UI consists primary from NodeUI class, secondary, there is a Connection class, that is never used directly and has a separate codex for optimized access. NodeUI is the same for all the nodes, it extracts generic information about sinks, sources, style and graphics objects from the model and puts them into layouts.

Upper level is composed of Flow Scene and Flow View. Flow scene connection with lowest layer was described in the previous section. But in general Flow Scene is a large container of nodes. It also has capabilities of placing new nodes on the canvas, owns Flow and Connection codices and provide user-level interaction with nodes and their connections (e.g. drag, connection, deletion, etc.)

The highest level is represented with application UI. It is composed of several dock windows and tab relay, that manages multiple opened projects.

5.3 Shared parts

Program has a set of shared classes that are consumed by multiple parts of the program. First thing to mention is a big set of constants and constant functions. These define file extensions and default values of the program. Among those are:

- .vtxproj - project file extension
- .vtxc - configuration file extension (for dock windows or last opened projects)
- 256x256 - standard node image resolution
- and others.

Those values may be quickly changed and are used for filter extension calculations, that happen at compile-time. Compile-time calculations are extensively developed and consumed by this project. Another example, that heavily relies on constants is a buffer structure, namely constant buffer.

Constant buffer is a dynamic structure of predefined types that is essentially a byte storage. Its primary task is to manage constant buffers that come into shaders in graphics side. It allows you to efficiently supply shader constants data to the pipeline. It is also commonly referred to as uniform storage or just uniforms. It has a fixed set of possible types, that can be passed to the graphics side. This set is defined as an enumeration of

[5]https://github.com/Agrael1/VeritasD3D

types. Their size, their offset in float sizes and other useful characteristics are stored within a compile-time template structure called Map.

Map is then used in processing of a layout. The values stored there take part in calculations of the byte offsets of a buffer elements. Enum values may be used to create a data layout and when applied to a buffer defines a strict view over data, that can be calculated, changed and stored accordingly. Essentially acting as a packed heterogeneous container shown on Figure 5.2 it is a great solution to push data to the graphics pipeline without a need to allocate several buffers and compromising on cache locality.

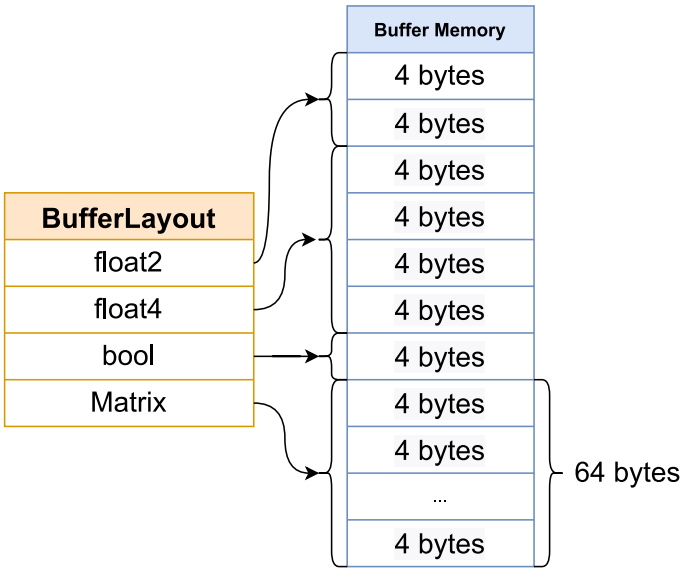


Figure 5.2: Constant buffer example. The data is tightly packed to be later bound in a single call.

5.4 Backend

This section is dedicated to the Backend layout and its implementation. The core concepts are pretty low-level, so the main goal of it is to make simple abstractions out of existing low level code, providing higher levels of abstractions needed outputs using simple interfaces. It also provides an information about compilers, working behind node graph and code compilation.

5.4.1 Engine

Graphics pipeline is represented with Engine class. Its sole purpose is to create images with shaders on-demand. Because OpenGL is quite restrictive in its behavior on multiple threads, the class is a singletone. For each entity, that demand a graphics a Frame Buffer is created and mapped onto the entity's memory address. The demanding entity is a graphics scene.

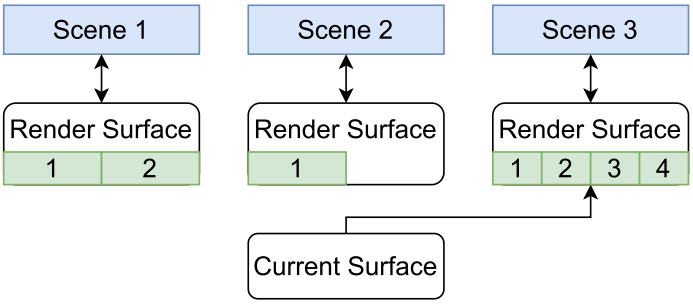


Figure 5.3: Render surface binding example. Each Render surface is bound to the scene, and is referenced by it. scene acts as a key to binder. Render surface may have several render targets (green rectangles), those depend on the scene parameters. If a current surface is active, its render surface gets bound to the graphics pipeline.

This allows for flexible frame switches between several scenes. Engine has a Context subclass, that initializes all needed variables for graphics pipeline. Each scene receives a render surface, which has the dimensions defined in the project. Dimensions of the surface are fixed for the current version of application, but that may be a subject to change in following releases. Render surface may have several render targets, viz 5.3. They are enumerated and the maximum amount is 8. This is due to limitations of hardware. The technology utilized is called Multiple Render Targets. The property, that defines current number of render targets is explained in the section 5.4.2, that is dedicated to the scene element called node.

Render targets are created according to the demands for each scene, the number may change with time, but only to a greater value. If the consumed render target count is less than allocated, then extra buffers would not bound to the pipeline.

The engine has a fixed Vertex buffer and vertex shader. Because OpenGL does not provide 2D rendering capabilities, unlike DirectX, the render should be done in a 3D domain. The polygon, that is drawn is a big triangle, that covers the entire render surface, but only a part of it is rendered. To map a correct texture coordinates a transformation is required. The correct texture coordinates, output by a vertex shader is called sv_texc. This value will be important in other chapters, dedicated to shader production.

The only information left to make a render call is a uniform buffer, and a pixel shader, optionally texture inputs. Those are supplied with each render call to engine, when there is something required to be drawn with graphics card. The engine has a default immutable texture defined as all values as zeros. If some of the texture inputs are missing, it will be substituted with an empty texture. The resulting image is stored into a Qt data structure called QImage. It is both a data holder and an interface unit, that can be drawn onto UI.

5.4.2 Node

The graphical element, that requires rendering and composes a large chunk of an application is called Node. Current section will explain the processes that work behind scenes of user interface. Node is a flexible interface, that provides structural integrity of a flow graph on a low level.

The node itself may have different relations to other instances. Those may be explained as:

1. There are multiple instances of different implementations of nodes, derived from the main interface.

2. There are multiple instances of the same node type, but with different algorithms, stored in it (e.g. one shader node stores Addition operation, while other in a Fractional Brownian Motion generator)

3. There are multiple instances of the same node with the same algorithm, but with the different set of parameters.

For clarity the implementation of node is an implementation of an interface, algorithm will substitute a name of a same node type with different algorithm from 2nd list item and an instance word is a concrete node element, described in 3rd list item.

To make this kind of interface several more objects were introduced to the system. First of them is a descriptor.

Descriptor solves the second relation, allowing to introduce same node type with different meaning to them. Its primary role is to create instances of nodes. As there is a quite obvious connection between node types and their possible number on single canvas, being 1 node type may be 0 or many times present on the canvas, it is crucial to create them with as less effort as possible, as there may be tens or even hundreds of them in a single graph.

Descriptor is constructed from node files, located in the ./nodes folder. The default format for node files is .json, it comes with all the information required to construct a node. The information is processed and passed into descriptor data set. That makes up a serialization requirement.

Data, stored in the descriptor may vary and depends on implementation, but all of them have reference count. The reference count solves the third relation from the list above. Now all the instances have their instance index appended to them, which makes them unique inside the system.

Second part is a model. Model represents an instance which stores current data, that can change with usage. All the models share common interface, so they can communicate with each other in an independent way. They have to provide information about connections with sinks and sources, as well as the instance name. Visualization of nodes is depicted on Figure 5.4.

The model has two kinds of submodules, that provide connection interface between nodes. Those are Sink and Source. Sink is a port, that can be used only once, so it is convenient to have a connection information stored in it, as it may uniquely identify the predecessor. Source on the other hand, provides a direct data connection for its user in form of shared resource. It is also worth mentioning, that in common node types data is present in form of shared resources, as it helps reducing memory consumption. The submodules can also be customized for any other node implementation, composing communication, between different types of nodes with different data passed and stored within each instance.

As was stated earlier ports may have types. They are currently:

- Grayscale - painted gray, store only luminosity values.
- Color - painted green, store color information.
- Normal - painted blue, currently unsupported.
- Universal - painted transparent, used for raw information transportation.

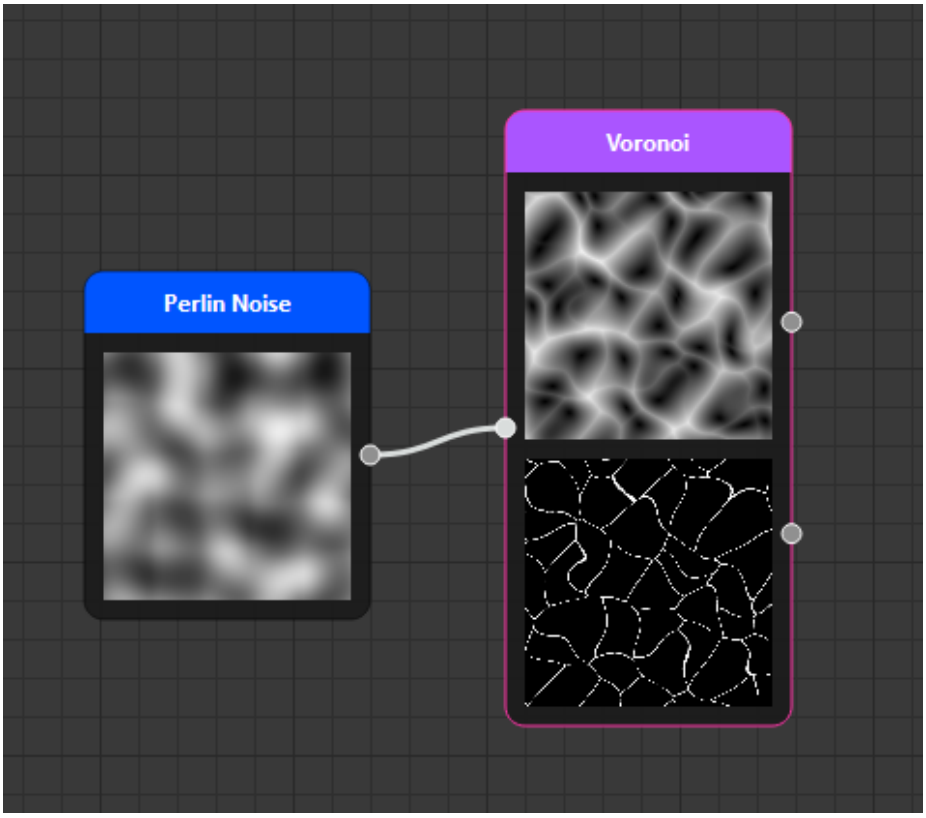


Figure 5.4: Nodes example. Two nodes of the same class have different algorithms, and different inputs and outputs. Perlin noise to the left has one output and zero inputs. To the right, Voronoi has one input and two outputs. Connected input is highlighted white, while output stays the same color, because it has no use limits.

The core of the data-flow graph is composed of a particular Node implementation called Shader Node. It is a common node, that represents manipulation with image data. Its descriptor holds data, that is shared between nodes of the same type. It consists of style information, which is used by UI, descriptors of ports, that consist of their name and type, Constant Buffer layout, which stays the same for one algorithm, with the parameters that define value borders of layout elements, and the algorithm itself, stored in form of compiled pixel shader, the implementation for which is provided by the Qt library.

The Descriptor is also capable of validating the input shader, and if the algorithm is wrong in some way it will be substituted with an empty one, filling all the outputs with the default texture and outputting an error message to the debug interface.

Because the Descriptor holds all the shared data within itself and its lifetime is greater than all the nodes of that descriptor, it is possible to keep a reference to that node in the model, which significantly reduces per-instance memory consumption. The model itself stores only the data, that is subject to change. Those are input and output images references and a constant buffer.

The shader model can also be serialized, the tree nature of the node file format allow them to use the same interface for serialization as for the project file. Thus it can be uniquely stored and identified by the name, the instance number and values of the constant buffer.

As was mentioned in section 4.2, the node is working with algorithms. Thus it is directly connected to the engine. Each instance of the node has buffer and input images, and it has referenced descriptor with a pixel shader. All those objects are sufficient to make a render call, which passes all the data down to engine. Maximum amount of inputs and outputs depends on graphics card, implementation defines 8 to be maximum outputs count.

The output count of a single node states how many render targets are to be bound to the pipeline, and the overall amount of the render targets, linked to a single render surface of the engine is determined by a maximum number of the outputs to the single node. However the model has to be present in the scene at least once for the render target number to change.

The output of the current node is an image. It can be exported in any format available in Qt library, those include the most common .png .jpg and .gif.

5.4.3 Code processing

Code compilation and composition is the meat of the project. Compilation is provided by Qt library. The OpenGL pipeline was chosen as it was the only cross-platform option available on Qt version 5.13. OpenGL version used is 4.2, because it supports multiple render targets, deferred rendering and has a nicer syntax compared to its predecessors.

Compilation of shaders is not the only option available in the project. Code editor uses its version of lexical analysis, developed solely for the project.

The Lexical analysis, or lexer for short, is made using coroutine generator, that produces tokens. The whole Lexical analyzer is a new type. It had to be quick to proceed with syntax highlight and for the later iterations even may result in intellisense, so the regular lexical analysis creators like Yacc and Lex did not fit the purpose. To define the lexer's abilities it is necessary to state the common performance bottlenecks, that will slow down its work.

The main problem with those applications is that they reallocate a lot of memory, and the whole process is slowed down by those operations. The new syntax analyser does not allocate any memory, and it works with the inline code.

The second problem was that Qt's syntax highlighter class works by code blocks. The block is a partial code extraction, meaning the code may not be parsed fully, hence parser would become context dependent, which would require complex context controls over a large amount of tokens, stored inside. Invoking such lexer would require separate thread and on a larger scale may become slow. The resulting analyzer had to make in-line assumptions without context.

The third problem is that full compiler is actually not required by the syntax highlighter at all, which was mentioned in section 4.5. That means there is no sense in parsing each and every token. In future releases it may be extended, but for now the feature was deemed unimportant. So much so, the lexer is not capable of parsing for example floating point literals, they are separated by a dot, which is read like a new token.

Problems were solved with introduction of string_view parser. String_view is a low level view over string data, that consists of pointer and length. The pointer does not own a string memory, nor does it contain a reference to the original string, so it is considered volatile if data is a subject of changes, however it is not the case for the lexical analysis, because it actually copies data from Code editor, when highlight event has occurred. Over this data a view may be constructed and freely manipulated, and because string_views are immutable, the information cannot be changed throughout parsing. Also they are lightweight, meaning there may be a lot of sub strings, that can be constructed from original view.

Being a coroutine means that the execution of the parser may be interrupted and resumed from the point of suspension whenever needed. It is a second allocation fix, as there is just no need in storage for the tokens in the first place, as they emerge from the function one by one, leading to latter suspension of the generator. So the only allocated memory is requested only by the context of the coroutine, which, in case of constant execution can be elided by the compiler, meaning the memory can go to the static space (or the .bss page), defined by the compiler, leaving no allocation at all. I hope that this type of lexical analysis will be recognized and will be developed, producing general language compilers, that does not rely on allocations to analyse the code, as they are fast and efficient.

Highlighter also relies on full lexical analysis, which is provided by the Node Parser class. Node parser works with the whole document at the beginning and is being revoked with timer events by the application side. By the time parser finishes, all the functions, types, macros and other tokens are passed to the syntax highlighter, forming sets of tokens, that highlighter searches through, when it is parsing by lines.

The lexical analysis uses a simplified finite state machine, that provides only partially correct tokens in several cases, but the highlighter, at least for the present release, does not require to abide by strict GLSL language rules.

After node compilation the dynamic node, that is located in the scene, changes, residing all the connections. The partial scheme of the editor is shown on the Diagram 5.5.

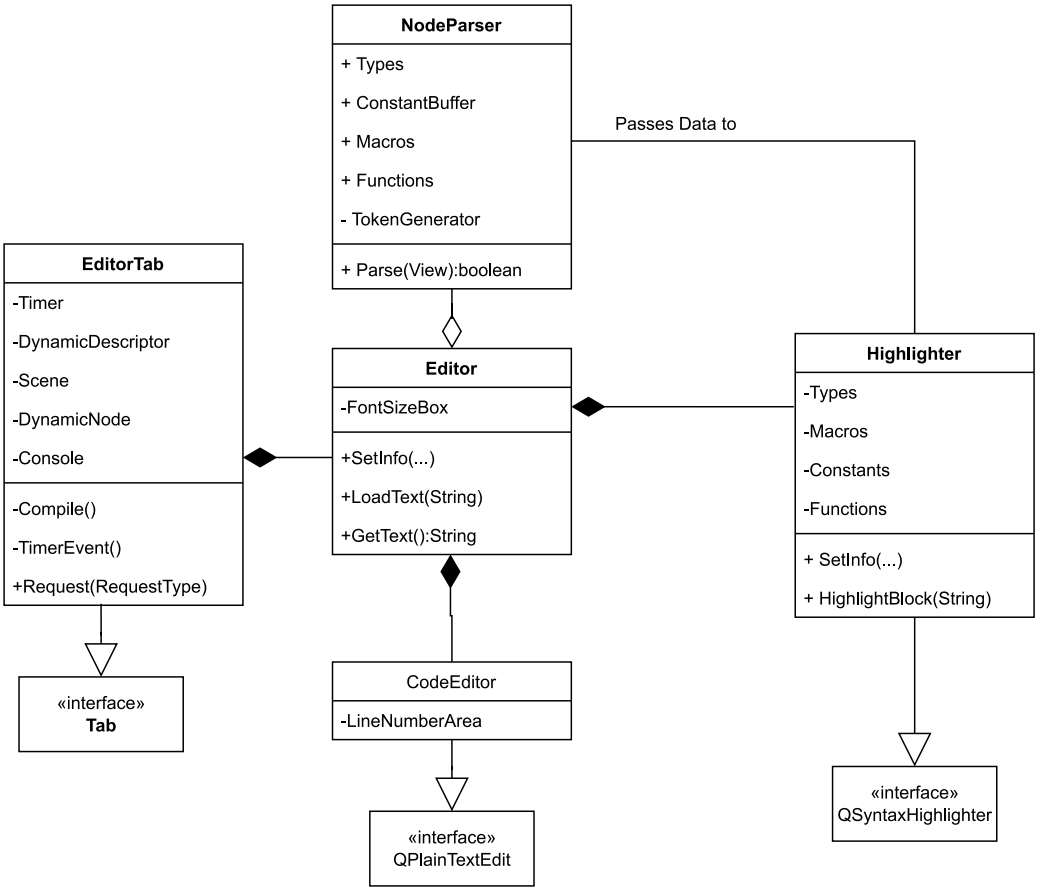


Figure 5.5: Node Editor class diagram. The Editor tab holds and manages FlowScene from Diagram 5.6, console and Editor. The data is transferred with SetInfo calls from NodeParser to Highlighter objects.

5.4.4 Graph compiler

The main feature of the program is its ability to make a code from the flow graph. The application has several rules about how that could be done:

1. The only flow graph to be exported is always a project graph. That means that, at least for now, there is no possibility to export the code from Node editor, because, as of now, dynamic node, present in the editor is considered volatile and should not be present in the connected graph. That results in only the project tabs have a code export capabilities.

2. Second, Output node is equivalent to shader output. If some node is connected to the output node, then it will be processed.

3. Third, compilation takes into account values of the node inputs, if the node input is empty, it will be processed as a black color (0,0,0) constant.

Several assumptions are to be considered, one of them is that node can be implemented by a 3rd party, so the regular visitor is not enough to compose the code. Hence the probe is used. Probe is a visitor class, that has storage capabilities, it collects data from all the nodes and produces an output, acting like a builder pattern of sort. The process is simple.

It starts with creation of instance of the probe. It requires information about nodes, present in the scene. Then the traversing is initiated on the array of output nodes in the current graph. The analysis takes into account the pre sequence of each output node supplied to the call. The algorithm then quickly descends to the first generator nodes using Depth-first recursive traversal. The only variation it has to the traversing is that it uses unordered_set, which has search complexity of O(1), that greatly enhances the speed of traversing by excluding the same node visiting. Nodes are processed by their names, reasoning for this lies in model structure, because every node sink stores whence the connection came as a string and this string acts like a key for map, that leads to the node by its name.

At the point of visiting the shader node the probe then receives the information from the full lexical analysis on about function declarations, custom data types and macros. The node provides information about its inputs and outputs. It also analyses if the node is complex or not. Complexity of the node is determined by the number of outputs it has. If the node exports more than one value, because GLSL cannot return an array from a function, the least resistant way of solving the problem is to introduce output structures. Those structures are unique to the algorithm and have their own storage, unique to the type. That eliminates situations, where several same nodes with different parameters causing structure duplication.

Each node should register its contents using set of functions, provided by Shader Probe. Most of the nodes will just register themselves as being simple and pass a shader to the probe. Because the graph is a tree the shader bodies of the nodes may just be stored and written as they come, the ordering is ensured by the structural connections themselves, meaning the node knows what it is connected to in advance.

The shaders coming to the probe are being slightly tweaked by the binder. In case of shader node, the changes are made by a builder class called Code Transformer. It gathers information on sinks, sources and constants and starts parsing code with the same lexical analysis coroutine, used in syntax highlighting. Changes are context dependent, but the general rules are:

1. All function name and macro tokens are formatted to `NodeName_instance_main`.
2. If the `main` token is located, change the return type of the function to the return tupe of node, being `vec4` for simple nodes and corresponding structure name for complex.
3. Add input arguments into empty brackets, for now only being `(vec2 sv_texc)`.
4. After curly bracket starts add instance of the output type and name it by the name of the node output if the node is simple and `sv_output` if complex.
5. If token matches output name and the node is complex, prepend `sv_output.`
6. If token is a uniform, its value is placed instead.
7. If `texture2D()` with matching input is located, `texture2D` is changed to name of the main function, corresponding to node, connected to that input. Example is shown in the Appendix A.
8. At the end of the function or at the return statement the output value is appended.

The GLSL source code is then formed in several steps.

1. For each Output node a corresponding output string is generated. The generation utilizes formatting pattern

   ```
   "layout(location = {})out vec4 {};"
   ```

   where location is defined as iteration number and second curly brackets are replaced with the name of the output node.
2. Each output structure for complex nodes that is stored in its dedicated array appended to the code.
3. All nodes shader codes are appended in order in which they came.
4. The `void main()` function is added, all the shader outputs are filled with values returned by corresponding functions, if the value comes from the complex node, a structure is created to hold the return values and the output, which corresponds to the named output is returned. If the structure is to be reused in multiple outputs, the value is copied, but the node function is called only once.

After the process `#version 420` is prepended to the shader and the shader is complete. However it can undergo an optimization process if demanded, in which the output of the Shader Probe is passed to a customized GLSL optimizer library. Code is then vectorized, functions are inlined and reprocessed several times. Cycles are unwound if iteration count does not exceed 5. If statements are usually replaced by `step()` functions. The whole process may take several seconds. In the end the optimized texture generation algorithm is produced, which can be used in fragment shaders.

5.5 User Interface

The user interface of the application is revolving around tab structure, that allows multitasking and working on several projects and/or nodes simultaneously. This section will mention most of the UI elements, that are implemented within projects and describe in a brief how they are implemented. Most of the UI is created either statically or dynamically using Qt Widgets library. The description will go the other way around this time, starting from the top entity and finishing with node representation.

5.5.1 Tab Relay

The tab system acts like a switch to the projects. It does not require an introduction, because it has the same properties as of web browser tabs. However, to make things more clear it is worth mentioning a small detail. The signals from the menu or from user input have to somehow be translated to the proper tab, the way this is achieved is by introduction of a request system. The request is a simple enumerable value, that has some meaning behind it. How this message is translated is up to the tab. There are 7 common requests within the system, they are:

- Save,
- Save As,
- Compile,
- Delete,
- Clear,
- Clear Selection,
- Export,

The idea of it is that calls may be issued to any tab implementation, they are the same in idea, but they may do different things. For example Compile request, coming to the code editor shall compile a code, that is within the constructed node, but for the Texture editor it means creation of the code from composed nodes.

Tabs also keep track of the file path they represent, meaning if the „save as“ is pressed and the file name differs, then the path, stored inside should also change to correspond the last file saved. The tab interface is described like this:

The tab can also be temporary, that means, that it has no path assigned to it. If that happens, the „Save“ request may trigger different code. By default it is set to „Save As“. Tab interface also provide behavioral functions, that may vary depending on the implementation, but they cover up several cases, that are extremely hard to do otherwise, like context switch on engine after tab is switched.

The tab contains modules, that are then shown within it, meaning if the tab is closed, then the whole module is deleted. The example system of a tab, that is depicted on working Texture Editor, so called scene module can be seen on Diagram 5.6.

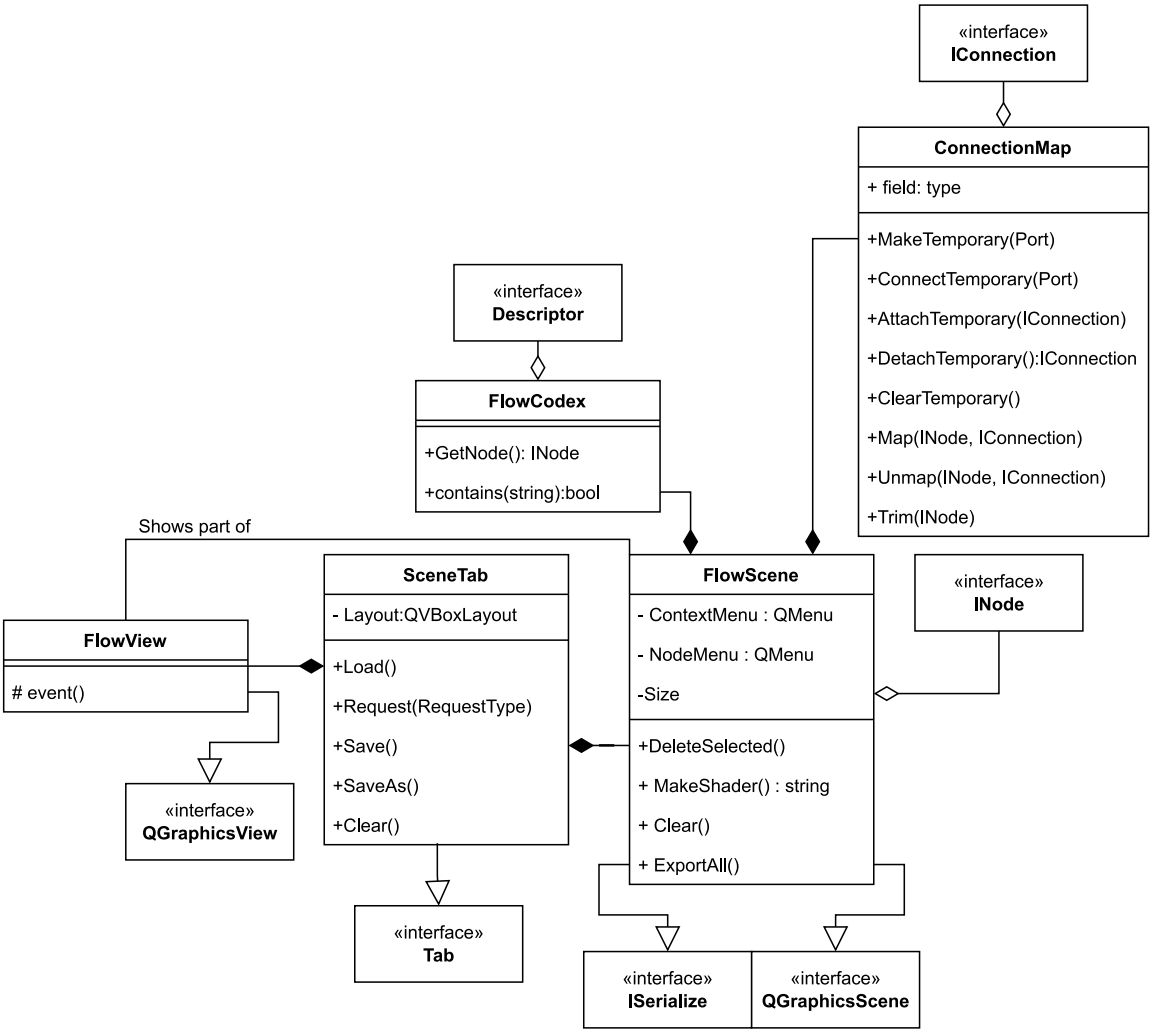


Figure 5.6: Flow Scene subsystem. The main class is SceneTab, that contains and manages all the other objects.

5.5.2 Flow Scene

Flow Scene is a graphics scene, represented by a large canvas with around 96000 x 96000 pixels. Even with the dimensions this huge, it is only partially rendered and redrawn only on-demand. The class, that makes a view over scene is called Flow View. It creates a rectangular frame, that could be dragged along the scene. It is also capable of translating user input and passing the events down to the scene.

The scene has a context menu, that contains all the objects, that can be added to the canvas. It is also capable of serialization, meaning, that the scene may be reloaded with all the objects on it. That makes a foundation to the project file for the application.

Scene comes as a part of a Texture editor, but it is also a part of Node Editor. Each scene may be unique, demanding own resolution from the engine.

5.5.3 Node Representation

Scene consists of data-flow graphs, those graphs are visualized using nodes and connections. Node UI consists of several parts, that are created dynamically. They are header, view and ports. The example is shown on the image 5.7.

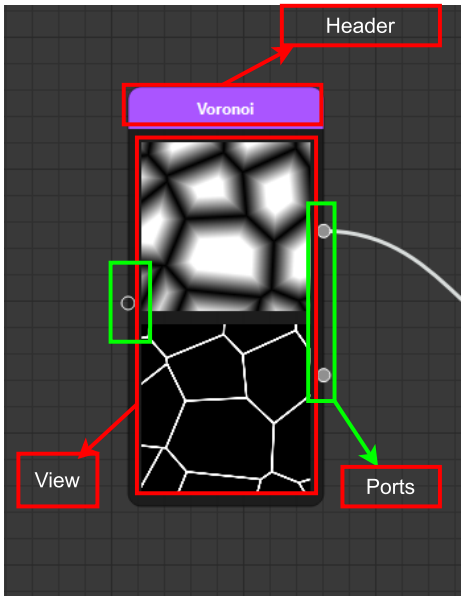


Figure 5.7: Example node parts location. Green rectangles represent ports, left one is sink, right ones are sources. View is located i the center of the node. It consists of images that it produces to the outputs. Node header is located in the top part of the node and it is defined according to model parameters.

View is dynamic and depends on several key functions in model. They provide information on what to draw and in what order. Mostly it is being built with images, that come from the engine, they even share the same memory pool. But the size is fixed and stands at 128x128 pixels.

Images are laid out one below the other, their number is mostly the same as the number of node outputs. The sink, source and order layout information is extracted to construct a view. Ports have their own classes for UI as well, they define how UI should handle operations. Sink port also stores a connection if it is present. That is explained by the earlier mentioned rule: sink may be only used once and the source - multiple times, and single connection does not require searching, hence there is a speed and space improvement.

Header gets its information from the style of the node, including name, font color and background color. Node style is compulsive, but it has default values, so it does not have to be initialized.

Connections are essentially splines, that are attached using simple collision detection with node port, provided by Qt library. They hold an information about two classes connected, number of ports being connected. They can be detached and reattached to other node, causing the post sequence of the node, whence the connection started, to be updated with new information from the connection.

Connections do not have model under them, instead they directly interfere with models of nodes, applying connections on their level, referencing the names of the ports, as well

as names of the nodes, connected to them. They can not be created directly, instead connections use a connection map, that produces, maps and stores them into proper nodes. connection amp also stores information about each nodes connection in searching structures for several operations.

5.5.4 Property Subsystem

On the right side of the screen, by default there is a dock window for properties of nodes. It is a dynamic interface builder for node properties, that represents user-end connection with constant buffer of the current selected set of nodes. It is built using visitor pattern. The Property Element class instance is passed to each selected object on the scene, appending widgets as it goes through the node. Model interaction with Property Element may be changed by the selected node implementation. For the shader node it is built to create a specific widget for every type of buffer value. The example is shown on Figure 5.8. Default widgets are sliders for every float vector and integer, check box for boolean, and for matrix it is defined as 4 by 4 matrix of text boxes.

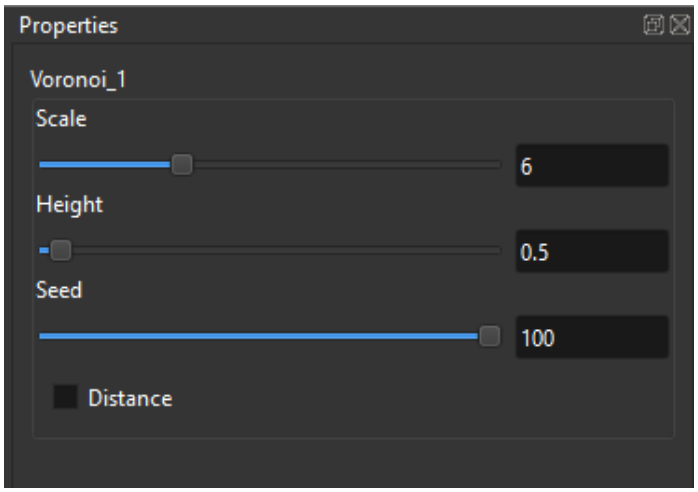


Figure 5.8: Property window. A Voronoi_1 node holds three floating point values and one Boolean value

5.5.5 Node Editor

Node editor is a tab widget, that allows user to create, test and export nodes for later usage in projects. Its user interface consists of three windows - Flow Scene, Text Editor and Console output window. The editor can be used with two options: creation of a node from scratch and loading of existing node. Latter sets all the ports, constants and UI, as well as pasting code into editor with compiling and checking if its working.

Flow Scene is the same throughout the entire program, but there is a key difference to it - in the center there is a Dynamic node present. It cannot be deleted, but can be used in any interactions like any other node. Dynamic node is derived from Shader node and does not vary much in functionality, but its descriptor is different. Although it is based on TextureDescriptor class, it has a set of instructions, that allow changes to the node structure.

Properties are also generated, it is allowed to change properties on-the-fly, bringing in new and editing the old ones with property editor mode. As shown on Figure 5.9 in addition to regular uniforms, there are inputs and outputs window.

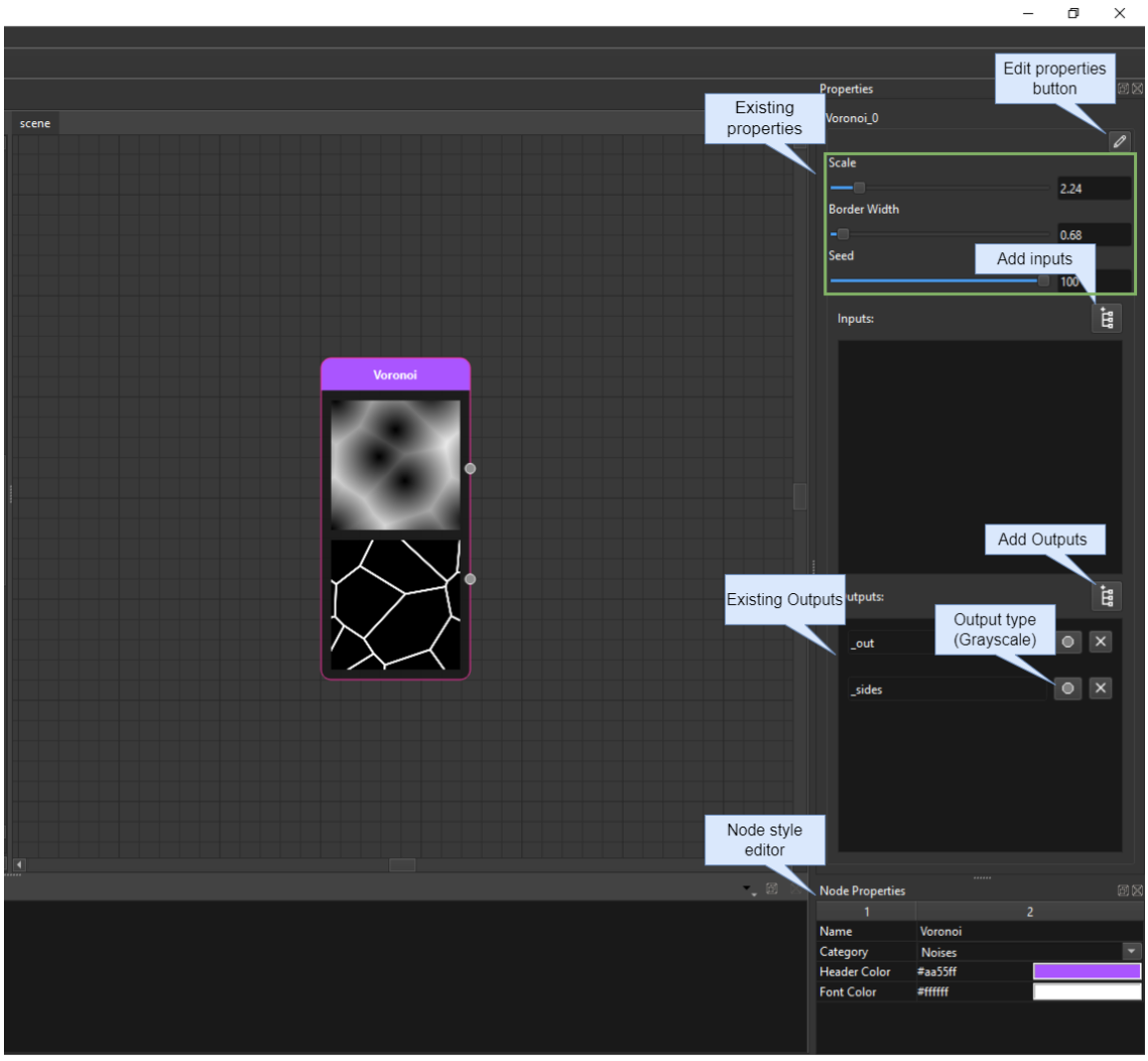


Figure 5.9: Editor Properties example interface. New sinks and sources can be added, changed and removed, inside inputs and outputs boxes, but the changes take effect only when compile command is issued. Node Style editor changes the looks of the node and its name.

Each type from constant buffer has a default value, several have minimal and maximum values limitations. those limitations may be put with the expand defaults. If values are not aligned, meaning max is less than min, and default value does not lay within the bounds, the property behaviour is undefined. Example is on Figure 5.10. Unless saved or compile action is issued changes will not take place. Note that code name and Regular name may be different, this is needed for better user experience, because code name is restricted by the rules of GLSL variable declaration. Code Name and type are compulsive arguments, if not set to some value, the save cannot be done. The Name will be shown only in editor and Code Name is telling how is the variable named in the code. If some uniform has no Name,

the interface propagates code name as an element name in the final interface of constant buffer. The min and max value may be unchecked, then there is no limit, the slider will be limited to values from -20 to 20, but text box will accept any number beyond the excluded limit.

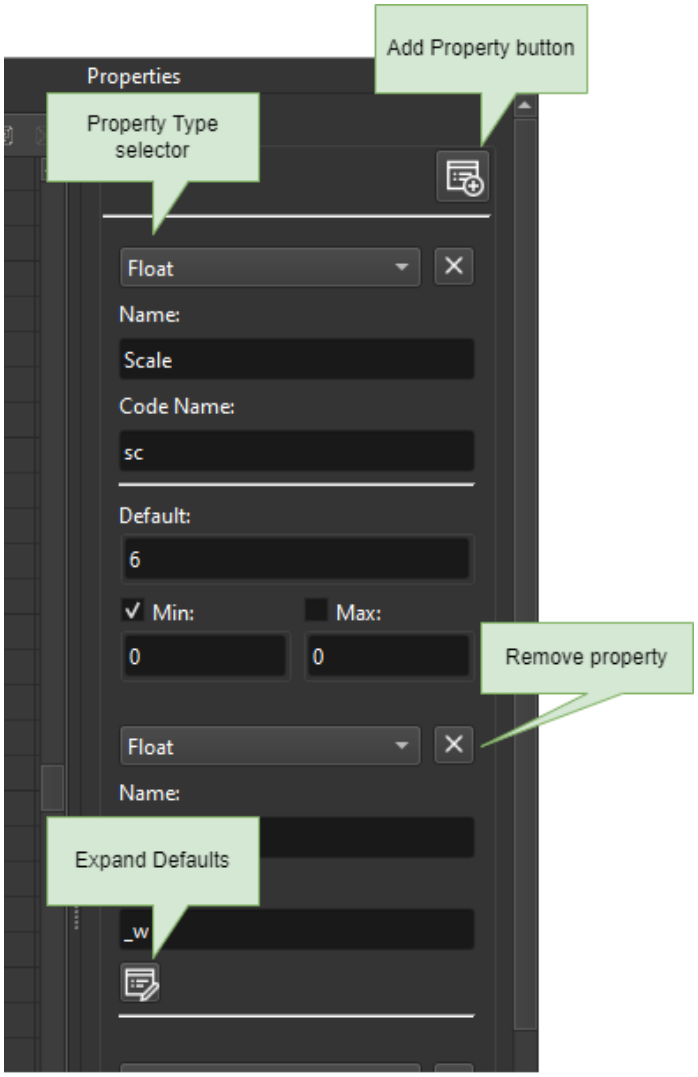


Figure 5.10: Change properties mode. The expand defaults button shows what default min and max values are, but the content may vary with implementation. e.g. Boolean values have neither max, nor min value.

To the left of the scene sub window a code editor is situated. It is a basic Text editor with syntax highlighting capabilities. The console is a simple output module, that collects errors from compilation and shows them to the end user. After recompilation is issued, the console is cleared and filled with errors of compilation.

The code, that is written in the editor is a GLSL code with several restrictions:

- The code should contain „void main()“, that is an entry point to the shader. If the shader has no main function, the behavior is undefined, input and output arguments are prohibited.

- The usage of „gl_FragCoord“ uniform is not recommended, because the polygon, that is rendered by the Engine class is not a square, but a triangle, that extends beyond the screen. Instead there is „sv_texc“ uniform. It is a normalized coordinate, ranging from 0 to 1, that resolves to a valid texture coordinate and it is highlighted by the editor.
- There is no way to pass in a resolution of the image. This is due to the fact, that the node can participate in shader construction and may become procedural texture, and procedural texture have no knowledge of an item they are applied to in the most cases. This restriction may be resolved in later releases, but the best substitution is to pass in a quality uniform.
- To access any input values only „texture2D()“ function should be used, the reasoning for this is that code creator does not know about any texture, different from 2D.
- Direct usage of „uniform“ and „sampler“ is not advised, all the uniforms and textures are managed with properties interface.

Chapter 6

Closing thoughts

To make up the application, capable of teaching those things in practice that is also able to create textures a lot of effort was put into developing a quick and easy-to-use user interface. A compiler studies and knowledge of new programming features, brought up by C++20 standard lead to a new kind of highly memory efficient lexical analyser being introduced in section 5.4.4.

The result of this thesis is a working multi-target cross-platform application, that can be used in a various ways by different groups of developers and artists. It allows creation of procedural texturing algorithms code generation with relative ease, as well as allowing users to introduce their own nodes to the application for further interaction and testing.

The results of chapter A shows the possibilities of the application, however there are a lot of modifications to be made, several of them are:

- Changeable resolution in the scene for image export purposes.
- Texture application on the model.
- Expansion of the standard nodes library.

The application is extensible and shall be released under free license allowing people to modify and extend it. The examples of a work in application are listed in the appendices.

Bibliography

[1] Chahat Monga, Richa. GRAPH TRAVERSALS AND ITS APPLICATIONS IN GRAPH THEORY. *International Journal of Computer Science and Mobile Applications*. 1st ed. january 2018, vol. 6, no. 1, p. 38–42. DOI: 10.47760/ijcsma. ISSN 2321-8363.

[2] Ebert, D. S., Musgrave, F. K., Peachey, D., Perlin, K. and Worley, S. *TEXTURING & MODELING A Procedural Approach*. 3rd ed. Morgan Kaufmann Publishers, 2003. ISBN 1-55860-848-6.

[3] Farhanaaz and Pillai, S. An exploration on lexical analysis. In:. March 2016, p. 253–258. DOI: 10.1109/ICEEOT.2016.7755127.

[4] Gustavson, S. *Simplex noise demystified*. Linkoping University, august 2005.

[5] Kronos group. *Shader*. 2019. Available at: https://www.khronos.org/opengl/wiki/Shader.

[6] Perlin, K. Improving Noise. *Proceedings of the 29th annual conference on Computer graphics and interactive techniques - SIGGRAPH '02*. 1st ed. july 2002, vol. 21, no. 3, p. 681. DOI: 10.1145/566570.566636.

[7] Rynkiewicz, F. and Napieralski, P. *Procedural fractal plants generation*. 1st ed. December 2016. 2434 p. ISBN 9788372838148.

[8] Thulasiraman, K. and Swamy, M. N. S. *Graphs: Theory and Algorithms*. 1st ed. USA: John Wiley & Sons, Inc., 1992. ISBN 0471513563.

[9] Vivo, P. G. and Lowe, J. *The Book of Shaders*. 2015. Available at: https://thebookofshaders.com/12/.

Appendices

List of Appendices

Appendix A

Simple node graph

Starting with simple graphs it is important to show how Editor acts with a simple node. The resolution is set to 64x64 pixels.

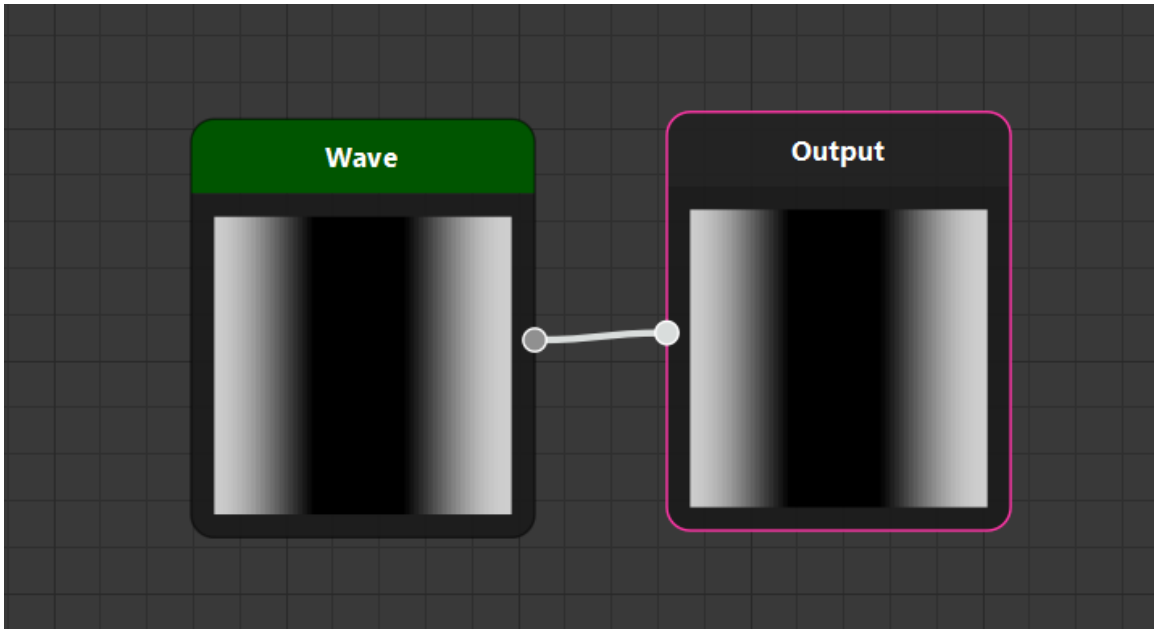
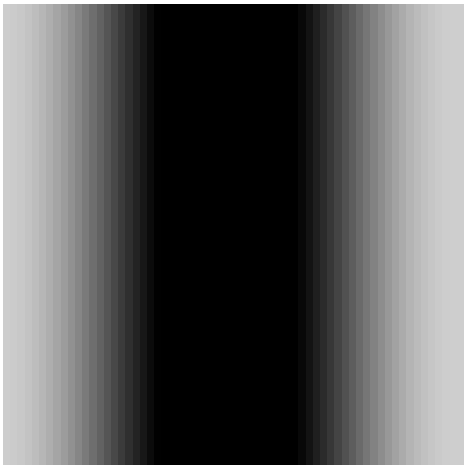


Figure A.1: Simple input, sine wave generator.



(a) Prescaled.

(b) Original.

Figure A.2: Simple output.

The optimized output listing:

```
#version 420
layout(location=0) out vec4 Output_0;
void main()
{
  vec2 st_1;
  st_1 = (gl_FragCoord.xy + vec2(−0.5, −0.5));
  st_1 = (((vec2(1.0, −0.0) ∗ st_1.x) + (vec2(0.0, 1.0) ∗ st_1.y)) + vec2(0.5, 0.5));
  vec4 tmpvar_2;
  tmpvar_2.w = 1.0;
  tmpvar_2.xyz = vec3(((sin((6.283185 ∗ (st_1.x + 0.263))) + 0.5880001) ∗ 0.51));
  Output_0 = tmpvar_2;
}
```

The unoptimized output listing:

```
#version 420
layout(location = 0)out vec4 Output_0;

#define Wave_1_PI 3.14159265359
mat2 Wave_1_rotate(float r){
    float a = sin(r);
    float b = cos(r);
        return mat2(b,−a,
            a,b);
}

 vec4 Wave_1_main(vec2 sv_texc){
    vec4 _out;
    vec2 st = sv_texc;
    st −= vec2(0.5);
    st = Wave_1_rotate(0∗Wave_1_PI) ∗ st;
```

```
17      st += vec2(0.5);
18
19      vec2 c = st;
20      float f = (sin((c.x+0.263)*2*Wave_1_PI*1)+0.58800006)*0.51;
21      _out = vec4(vec3(f),1.0);
22      return _out;
23  }
24
25  //-----------------------------------------------//
26
27  void main(){
28      vec4 tmpvar_Wave_1 = Wave_1_main(gl_FragCoord.xy);
29      Output_0 = tmpvar_Wave_1;
30  }
```

It is clear that there is no rotation applied, so the code does not produce rotation matrix in optimized case.

Appendix B

Complex node graph

Now for a bit complex graph, the simulation of rocks and cracks. The output is at 256x256 pixels.

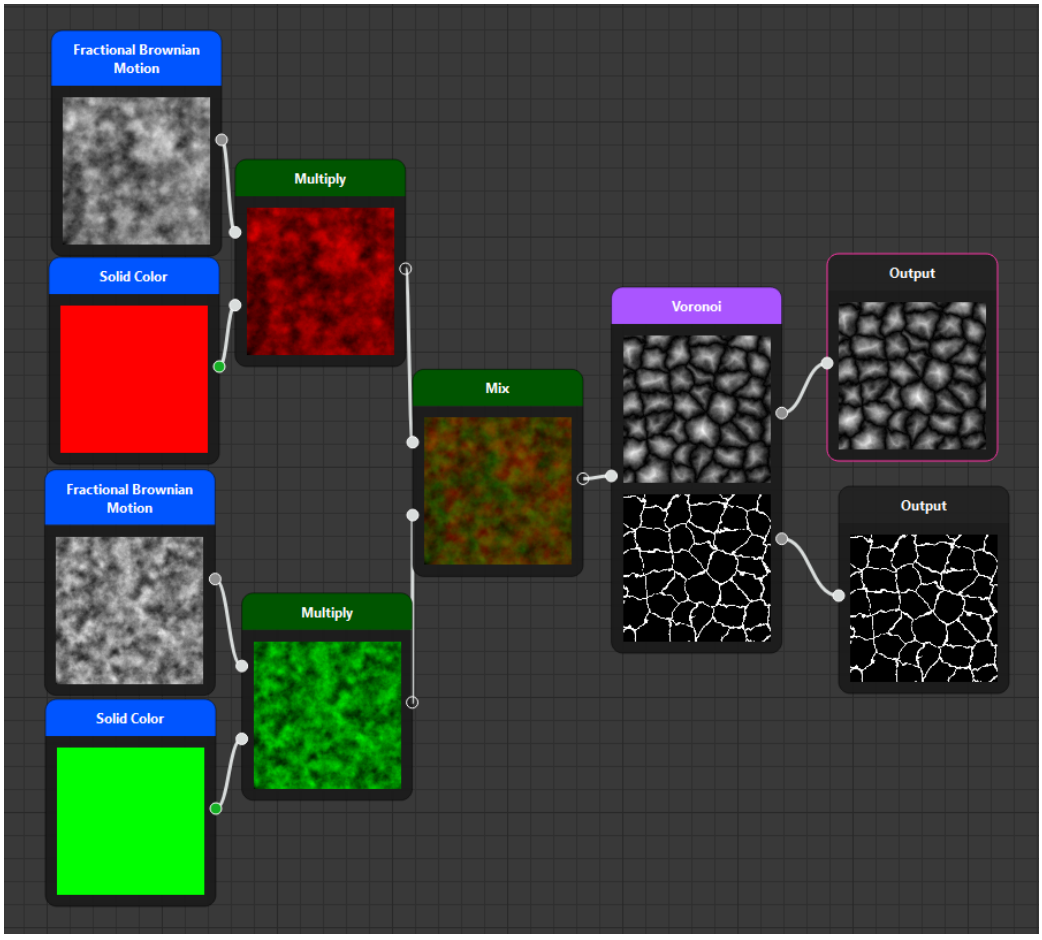
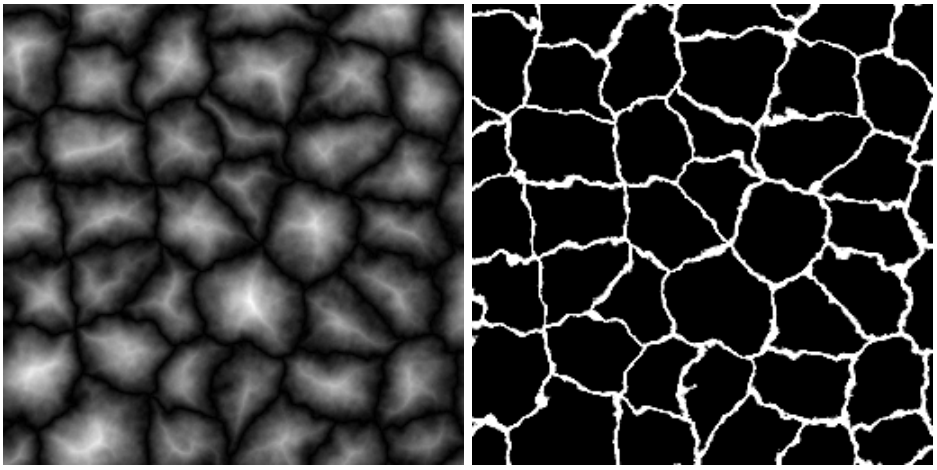


Figure B.1: Rocks pattern graph.



(a) Rocks output. (b) Cracks output.

Figure B.2: Images from outputs.

The listings will not be applied, because they are extensive (>200 lines each).

Appendix C

Node Editor

A simple code is written, to generate tiles.

```
void main()
{
	vec2 I = floor(sv_texc*_sc);
	bool vert = mod(I.x+I.y,2.)==0.;
	if (vert)
		_out = 1.-_out;
	_out *= vec4(.9,.85,.85,1);
}
```

Figure C.1: Code in editor.

A node after compilation can be immediately used in test graph:

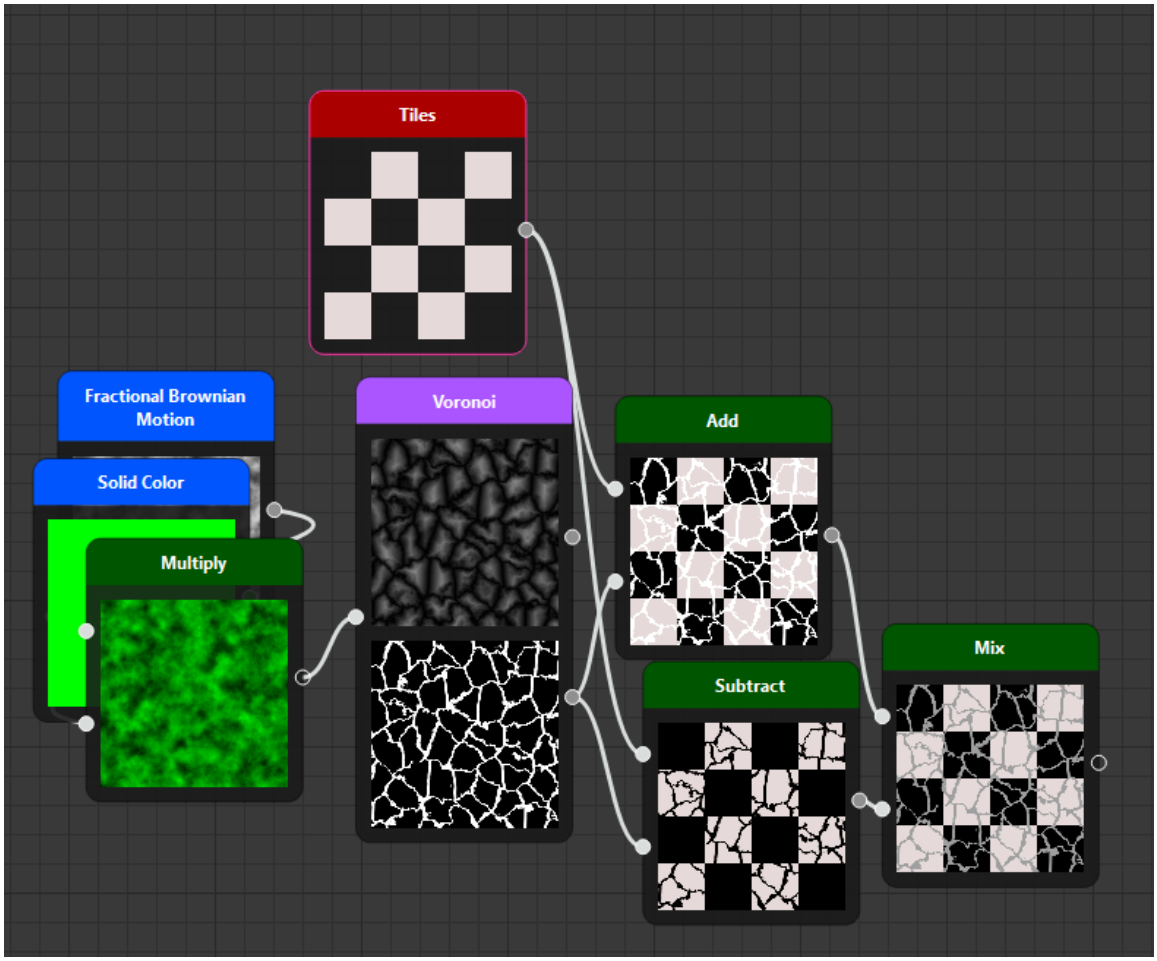


Figure C.2: Tiles in graph.

Result can be exported with resolution of 128x128 pixels:

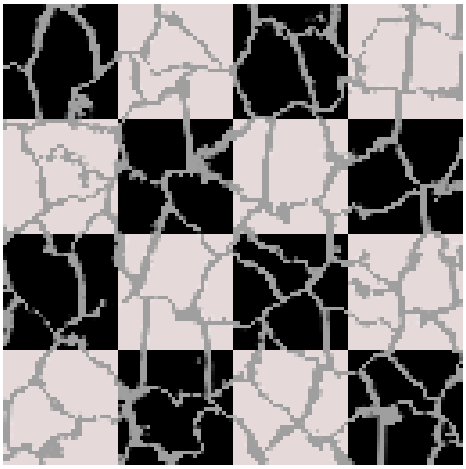


Figure C.3: Result image.

Appendix D

Editor Interface

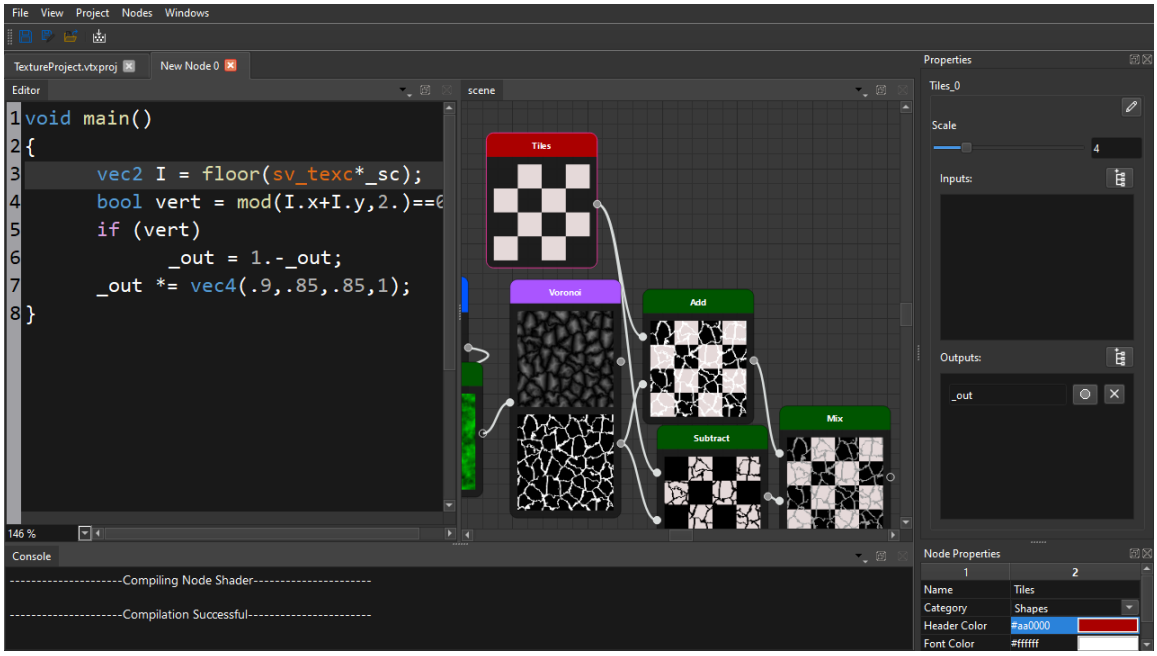


Figure D.1: Editor Interface example.