Web editor of workflows to generate procedural terrain based on graphs

García García, Pablo Luis

Curs 2017-2018

Director: JAVIER AGENJO

GRAU EN ENGINYERIA INFORMÀTICA

Treball de Fi de Grau
For the friends that believed in me,
the overcomed past
and the years to come.
Acknowledgements

Many thanks to Javier Agenjo for his patience and advices.

Thanks to my family for supporting me every day.

And thanks to all the people who helped me go this far.
Abstract

The requirement of realism and wide open worlds in video games has risen as the hardware has become more powerful and efficient, which has favored the procedural terrain generation, in detriment of the modeling technique. Alongside, many web tools and editors are arising due to the possibilities they offer, such as multiuser interaction or accessing them from any device.

The objective of this project is to create a web editor that allows creating, previewing and exporting a graph-based workflow that generates terrain procedurally. For this purpose, I am going to modify an existing graph-creation library and explain the process of creating a web editor and how to represent real time graphics, along with the different techniques currently used to generate heightmaps procedurally and how to modify them to create realistic terrains.

Resumen

La exigencia de realismo y extensos mundos abiertos en los videojuegos ha crecido conforme el hardware se ha vuelto más potente y eficiente, lo cual ha favorecido la generación de terreno de forma procedural, en detrimento de la técnica de modelado. Junto a esto, muchos editores están surgiendo en la web debido a las posibilidades que ofrecen, como la interacción multiusuario o acceder a ellos desde cualquier dispositivo.

El objetivo de este proyecto es el de crear un editor web que permita crear, previsualizar y exportar un flujo de trabajo basado en grafos que genere terrenos de forma procedural. Para este propósito, modificaré una librería existente de creación de grafos, explicaré el proceso de crear un editor web y cómo representar gráficos en tiempo real, junto con las distintas técnicas utilizadas actualmente para generar mapas de altura proceduralmente y cómo modificarlas para crear terrenos realistas.
Prologue

I have always been passionate about computer graphics and how, mainly by interactive experiences, they show what reality mostly cannot. This is what motivated me to study the OpenGL API and also led me to learn the techniques and algorithms used in the different fields of real time rendering.

Increasingly over the past few years, internet and web browsers are arising as an interesting medium to widely and quickly share real time computer graphics contents, such as video games or any other artistic representation. Editors and web tools are also commonly used for its availability and platform-independent usage, which drove me to learn JavaScript, the lead technology for web pages’ interactivity.

Real time rendering performance and realism is itself very tied to the hardware it is being computed on, but it is even more critical when you have memory and performance limitations as when computed through a web browser, where minimum use of bandwidth is required.

Open worlds and great extensions of terrain are increasingly being used in videogames as a consequence of the demand from the players asking for more freedom and immersive experiences. The intention of this project is to contribute to the growing use of computer graphics in the web, through a web editor that will allow web developers to simplify their workflow when creating procedural terrain in their videogames and web applications.
## Contents

Abstract vii

Prologue ix

List of Figures xi

List of Tables xiii

1. INTRODUCTION ................................................................. 1
   1.1 Objective ........................................................................ 1
   1.2 Motivation ...................................................................... 1
   1.3. The concept of heightmap ............................................. 2
   1.4. Procedural generation ................................................... 3

2. STATE OF THE ART ............................................................. 5
   2.1 The web languages ......................................................... 5
   2.2 WebGL and real time graphics ......................................... 6
   2.3 LiteGraph ...................................................................... 8
   2.4 Similar applications ...................................................... 9
       a) World Machine .......................................................... 9
       b) Houdini .................................................................... 10

3. PROCEDURAL TERRAIN GENERATION TECHNIQUES ............... 13
   3.1 Common theory ............................................................ 13
       a) Frequency .................................................................. 14
       b) Amplitude .................................................................. 14
       c) Octaves ..................................................................... 14
       d) Offsets ..................................................................... 14
   3.2 Value Noise .................................................................. 14
   3.3 Perlin Noise .................................................................. 16
   3.4 Worley Noise ................................................................ 17
   3.5 Deformation filters ....................................................... 18
       a) Pow Filter ................................................................ 18
       b) Mix Filter ................................................................ 19
       c) Noise Filter ............................................................. 20
       d) Clamp Filter ............................................................ 20
       e) Remap Filter ............................................................ 21
       f) Blur filter ................................................................. 21
g) Invert Filter ................................................................. 22
h) Perturbation filter ......................................................... 22
i) Join filters ................................................................. 23
j) Max and min filters ....................................................... 23
k) Sum and Sub filters ..................................................... 24
l) Lerp Mask Filter ......................................................... 24
m) Height and Slope Colour Filters .................................... 25

4. IMPLEMENTATION ........................................................................ 27
4.1 Web tool design ....................................................................... 27
4.2 Base Framework ....................................................................... 27
  a) Vector objects ....................................................................... 28
  b) The matrix object ................................................................... 28
  c) Camera object ......................................................................... 28
  d) WebGL buffers ....................................................................... 29
  e) Texture object .......................................................................... 30
  f) Shader object ........................................................................... 30
  g) Framebuffer ............................................................................. 30
4.3 Custom LiteGraph nodes ................................................................. 31
  a) Output Node ............................................................................ 32
4.4 Generation shaders ..................................................................... 33
  a) Common vertex shader ............................................................. 33
  b) Common fragment functions ..................................................... 33
4.5 Terrain Object ........................................................................... 34
  a) Use of heightmap, normal map and colour map ...................... 36
4.6 Renderer object .......................................................................... 36
4.7 Editor object ............................................................................. 36

5. Conclusions and future work .......................................................... 37
5.1 Loading in Unreal Engine ............................................................ 37
5.2 Conclusions .............................................................................. 38
5.2 Future work ............................................................................... 39
List of figures

Figure 1: Example of what can be achieved with ShaderToy. Real time rendering. .......... 1
Figure 2: Terrain generated with Unreal Engine (Left) and its heightmap (Right) .......... 2
Figure 3: Procedural terrain visualized in Unreal Engine (Left) and its heightmap (Right) .................................................................................................................................................................................. 3
Figure 4: HTML and CSS (left) and the resulting web page (right) ......................... 5
Figure 5: Model, view and projection matrices results [3] ........................................... 6
Figure 6: The OpenGL’s graphics pipeline [4] ................................................................. 7
Figure 7: A simple example of the LiteGraph workflow .............................................. 8
Figure 8: A workflow created with World Machine ...................................................... 9
Figure 9: Preview of a procedurally generated mesh using World Machine .............. 10
Figure 10: Example of Hoidini’s disposition [7] ........................................................... 10
Figure 11: White noise texture (left), 10x10 portion of that texture (middle) and the portion scaled back (right) .................................................................................................................................................................................. 13
Figure 12: Example of noise texture generation using Value Noise algorithm and 4 octaves ........................................................................................................................................................................... 14
Figure 13: A grid cell, with four random numbers in its vertices and the offset of the pixel we are in (left). Part of Value Noise algorithm (right) ................................................................. 15
Figure 14: Example of Value Noise with smoothing not applied (left) and smoothing applied (right) ........................................................................................................................................................................... 15
Figure 15: A grid cell, with four random gradients in its vertices, the offset of the pixel we are in and the vectors from the offset to the vertices (left). Part of Perlin Noise algorithm (right) ........................................................................................................................................................................... 16
Figure 16: Example of noise texture generation using Perlin Noise algorithm and 4 octaves ........................................................................................................................................................................... 16
Figure 17: A grid with a random point in each cell, the offset of the pixel we are in and an arrow indicating the closer point (left). Part of Worley Noise algorithm (right)..... 17
Figure 18: The original cellular noise (left) and cellular noise with distances subtraction (right) ........................................................................................................................................................................... 18
Figure 19: Example of pow operation with exponent 3 over a heightmap. Original (left) and transformed (right) ........................................................................................................................................................................... 19
Figure 20: The mix function [12] .................................................................................... 19
Figure 21: Cellular noise (left), cellular noise with distance subtraction (middle) and the mix between them with a 0.5 threshold (right) ................................................................. 19
Figure 22: Cellular noise (left) and cellular noise with noise filter applied (right) .... 20
Figure 23: Cellular noise (left) and cellular noise with a clamp in the range [0.2, 0.8] (right) .................................................................................................................. 20
Figure 24: Remap function .............................................................................................. 21
Figure 25: Cellular noise (left) and cellular noise with a remap in the range [0.2, 0.8] (right) .................................................................................................................. 21
Figure 26: Cellular noise (left) and cellular noise with a remap in the range [0.2, 0.8] (right) .................................................................................................................. 22
Figure 27: Cellular noise (left) and cellular noise inverted colours (right) ............... 22
Figure 28: Cellular noise (left) and cellular noise with perturbation applied (right) ..... 23
Figure 29: Example of two heightmaps joined together ................................................. 23
Figure 30: Example of min filter between two heightmaps (left, and middle) ............. 24
Figure 31: Example of sum filter between two heightmaps (left, and middle) .......... 24
Figure 32: The base textures (first and second images), the lerp mask (third image) and the resulting heightmap (fourth image) ....................................................... 25
Figure 33: Example of Height Filter applied to a texture ................................................. 25
Figure 34: The web tool design ....................................................................................... 27
Figure 35: Perspective matrix (left) and view matrix (right) [14] .................................. 28
Figure 36: Roll, yaw and pitch directions (left), how to obtain front from yaw and pitch (right) .............................................................................................................. 29
Figure 37: Heightmap (left), normal map (middle) and colour map (right) .............. 32
Figure 38: Barycentric points displayed as colours ......................................................... 35
Figure 39: Creating indices for TRIANGLE_STRIP using degenerated triangles [16] . 35
Figure 40: Unreal Engine’s tool to import heightmaps .................................................... 37
Figure 41: Generated terrain after loading heightmap in Unreal Engine ...................... 37
Figure 42: Terrain rendering in Unreal Engine after colour map and normal map are applied .................................................................................................................... 38
Figure 43: Terrain after combining three different noise generators ......................... 39
List of tables

Table 1: Buffer object pseudocode ........................................................................................................29
Table 2: A template for LiteGraph custom node creation ............................................................ 31
Table 3: Common vertex shader ..............................................................................................................33
Table 4: Common fragment shader functions ...................................................................................... 34
1. INTRODUCTION

1.1 Objective

The objective of this project is to create a graph-based editor to generate terrain procedurally in the web. That is, a web page that allows developers to create an exportable graph-based workflow to create procedural terrain and be later used in their own applications. The web tool previews in real time the final result, and it is easily scalable with further procedural generation techniques.

For this purpose I will develop a framework to render 3D content in the web, I will expand the functionality of the graph engine LiteGraph, integrating the main algorithms used in procedural generation such as Value Noise, Perlin Noise and Worley Noise along with some filters to modify the final result. The algorithms for procedural generation and the filters will be executed in the GPU for maximum performance.

1.2 Motivation

One common problem faced when displaying graphics realistically or in a hardware limited environment is the memory limit. In modern applications, 3D meshes with high level of detail can take a huge amount of work and storage, which is not scalable, for example, if you want to render an open world map. Procedural graphics generation allows creating random and diverse content that can be saved for later use in a heightmap (also called displacement texture, explained in section 1.3), and also creating it at runtime for infinite worlds, which implies less memory consumption but higher CPU usage. This is normally not a problem since the procedural content is usually calculated only once and not in a per-frame basis, but this is of course dependent on the use case.

Web browsers are precisely a very limited scenario for rendering and calculating procedural data, where you have to prioritize fast loading times and high efficiency in your operations. It makes sense then to use procedural graphics generation whenever possible when rendering in the web. A good example is the web tool ShaderToy [1], which allows to create realistic real time graphics through shaders as an interesting way of procedural generation, but this technique's complexity and limited interactivity, since only receives the mouse position inside the canvas, made them difficult for videogames to proliferate.

Figure 1: Example of what can be achieved with ShaderToy. Real time rendering.
Nowadays, extensive, rich and open world video games are the more demanded between consumers. It is, then, a challenge trying to implement a web tool able to generate terrains that can be useful and fulfil the expectations of the developers. My intention is to make it straightforward enough so anyone can integrate a procedurally generated terrain in their application through the generated textures in my editor.

Although there are many implementations in the web of the main algorithms used to generate terrain procedurally, it does not exist an online tool to unify them and let you interact, preview and export directly the desired result.

### 1.3. The concept of heightmap

There are two ways to store terrain nowadays, one is by using a 3D mesh and another is baking the shape in a displacement map texture (i.e., a heightmap), where each of the pixels intensities are used as offsets for the height of every vertex in the shape. Each of the techniques has its advantages and disadvantages. For example, heightmaps are easier to store and edit, but they have problems when texturing steep surfaces. In the other hand, with 3D meshes it is easier to overlap terrain, but it is more difficult to calculate collisions.

An example of a heightmap-based terrain generator is, for example, the Unreal Engine’s terrain editor. The engine creates a base plain grid and allows the user to alter the height of its vertices with the mouse. Under the hood, it is creating a single channel (red, green, blue or alpha) 2D texture and using this channel variation (normally from 0 to 1) as a height offset.

![Figure 2: Terrain generated with Unreal Engine (Left) and its heightmap (Right)](image)

This method of manually painting the heightmap is only doable if the videogame’s terrain extension is small enough, for bigger ones the terrain is generated by either taking heightmaps generated by satellites over the world, or procedurally with, in some cases, a few manual fine tuning. The main reason for this is that, while procedural generation gives virtually infinite content, it can be a bit tricky to get the exact desired output. These terrains serve in most cases as a base building block for more game specific contents to be added.
Heightmaps are also used outside of the videogame and film industry, since it is a very good method for saving information about a terrain. There are many satellites around the world creating heightmaps of different parts of the earth that are later used in different applications, for example in Google Earth.

### 1.4. Procedural generation

Procedural generation is the process of creating data following an algorithm. It is widely used in applications to add variation and a sense of randomness. In the context of videogames, they can be used, for example, to create textures or place objects that need some randomness such as rocks or trees. Lately they are even used in entities animations to make them feel more natural, as used for example in the game Overgrowth [2].

Applying this to terrains, it means it will be created following some predefined rules in an algorithm with a set of input parameters, as opposed to manually placing the different desired reliefs of a terrain. The resulting terrains will vary between them when those parameters are changed. Most of these algorithms are based on a grid subdivision and the use of pseudo-random number generators, for example the Perlin Noise algorithm, which I will explain later.

![Procedural terrain visualized in Unreal Engine (Left) and its heightmap (Right)](image)

Generating realistic terrains using procedural algorithms can be tough, since they tend to create repetitive patterns and it requires some time to create the desired shape. In the figure 3 we can see a heightmap on the right, and the resulting terrain when the texture is used as an offset in a grid on Unreal Engine 4.
2. STATE OF THE ART

2.1 The web languages

HTML (Hypertext Markup Language) has been almost since the beginning of internet the standard language to create web pages. There is a wide disposal of building blocks called HTML elements, which can be hierarchically placed to create the desired design. On top of that, CSS (Cascading Style Sheets) is used to apply custom visuals and effects to the mentioned HTML elements. It allows setting custom sizes, colors and positions of individual elements, making it tightly coupled with HTML and indispensable for creating web pages. These two languages by themselves allow creating nice looking websites, but they are very limited when you want to add some dynamism and interactivity, JavaScript is used for that purpose.

JavaScript is very powerful interpreted language widely used in the web. The language is prototype-based (very similar to object-based languages) and allows adding custom behavior to a web page, such as performing a certain action when a button is clicked or directly executing a videogame in real time. The syntax is similar to Java, but the core design is very different. JavaScript, apart from being interpreted is also dynamically typed, meaning that a variable’s type is inferred from the type of the value that is assigned to it and can later be changed to a value with another type. Nowadays, JavaScript is not only used in client-side applications, but also in servers. For example Node.js executes JavaScript in the server, allowing the generation of custom web pages before sending them to the client.

Looking at the code from the figure 4, HTML elements are the components that start with "<"tagname">" and end with "</"tagname">". Every element inside of an existing element is considered a child and inherits the properties of its parent. For example, the line "<h1>HELLO WORLD!!</h1>" is a child of the HTML element "<body>" which is itself a child of "<html>". Everything written inside a "<style>" element is expected to be coded in CSS. In this case, I am changing the colours and font sizes of different elements used in the example. Similar to CSS, JavaScript code needs to be inside its own element, in this case the "<script>" element. In the simple example, I am replacing
a paragraph’s text with the result of calculating the cosine of 0.5, using dynamic typing and the Math object.

2.2 WebGL and real time graphics

WebGL (Web Graphics Library) is an API for rendering real time graphics in the web by allowing JavaScript to communicate with the host GPU. It is based on the embedded version of OpenGL, OpenGL ES, but it has been adapted to be interpreted by web browsers. WebGL acts like a state machine, meaning that when an API function is called, it will be either for changing the current state or to modify it.

WebGL is also a rasterizer, meaning that graphics and shapes are represented using vertices and vectors that are later rasterized to become a set of pixels. Most modern real time graphics applications such as videogames use OpenGL or DirectX as rasterizers, but as computers become more powerful this technique may decay in favour of raytracers. As opposed to rasterizers, raytracers generate images by creating a ray from the point of view to each pixel of a virtual image plane. When this ray collides with an object, the colour is calculated using the object’s material and how light impacts in this position. The visual fidelity of this technique is really impressive, but as a cost of high CPU load. This is why raytracers are used in the film industry, where there is no need for real time rendering and realism is expected to be high.

In order for WebGL to convert a set of vertices and vectors to an image, they have to be transformed to fit them inside a virtual 3D coordinate system, then inside a virtual camera and finally be projected to the monitor. This process is achieved with three matrices called the model matrix, view matrix and projection matrix.

In the figure 5 we can see a visual representation of the process. The model matrix transforms an object from its local space to world space, that is, it contains the translation, rotation and scale of the object with respect to the world coordinate’s origin. The view matrix transforms an object so that it is seen from a custom point of view, simulating the behaviour of a virtual camera. Finally, the projection matrix specifies how the 3D objects will be projected to the screen, being perspective and orthographic.
projections the more used techniques. In perspective projection the objects are smaller the more far away they are from the camera, while in orthographic projection they always have the same size.

The GPU makes WebGL to go through a number of stages, also called the graphics pipeline, in order to recreate the desired output. The GPU needs to be fed with a vertex shader and a fragment shader. Shaders are programs written in GLSL, a language with a syntax similar to C++, that are provided to the GPU at runtime, where they are compiled and used as the render algorithm for the current elements that are desired to be drawn on screen. Vertex shaders and fragment shaders are linked together after being compiled, creating a so called “shader program”. Since WebGL acts as a state machine, a draw call will use the current bound shader program to render what was requested, meaning that only one shader program can be bound at a time, but it can be changed as many times as needed at runtime.

![Figure 6: The OpenGL’s graphics pipeline](image)

In the figure 6 we can see a simplified version of the WebGL’s graphics pipeline. First, the vertex shader is fed with an array containing vertex data of the mesh that wants to be rendered, such as the vertices, the normals to simulate realistic illumination or the texture coordinates. When data wants to be submitted to the GPU, such as the vertex data in this case, it is done through buffer objects. When a buffer object is created, WebGL returns an identifier that can be used to bind the buffer whenever needed. The vertex shader is called for every position of the vertex data array, and allows manipulating this data as desired. It is normally used to transform the vertices using the model, view and projection matrices mentioned before, since doing these calculations in the GPU is way faster. In the second stage, after all the vertices have been transformed, shapes are assembled using a predefined primitive (usually triangles, but also supports lines and points). Geometry shaders receive an assembled primitive as input, and allow to discard it or to expand it creating new primitives, but they are not currently supported in WebGL.

In the next stage the primitives are rasterized, that is, they are converted into a set of fragments (used to compute pixels) as big as the number of pixels from the target resolution. The primitives are rasterized in the order they are given and the positions of
the fragments are calculated by interpolating the vertices of the primitive. After that, a fragment shader is called for every fragment of the set, where the colour of textures is normally read using the interpolated texture coordinates to be later modified with custom calculations, such as lightning algorithms. The last stage is used to check if the fragment is in front or behind other objects using a depth buffer, discarding them accordingly, and also to check if the fragment should be blended with a fragment in the same position using their alpha and the predefined blending equations.

All this process generates a framebuffer (array of pixels) that is normally used to be rendered at the screen for the current frame, but can be also used as a texture for other purposes, such as post-processing effects. In a modern videogame there are usually 60 frames per second, and the target resolution is generally 1920x1080 (i.e. 2073600 pixels), giving you an idea of how fast GPUs are.

2.3 LiteGraph

LiteGraph is a JavaScript module that can be used as a standalone library to execute already created graphs from disc or as a visual graph-node editor in a HTML canvas. In LiteGraph nodes represent functions or representations of data that are connected between them to implement an algorithm, which makes them quite useful since they are able to visually abstract them, without writing any code. Every node defines optionally a set of input parameters that are processed in some way and a set of output parameters that are used to export the data that has been processed. For example, a division node would take two input parameters which are the operands and it will have an output parameter which is the result of the division between them. An example can be seen in the figure 7.

LiteGraph provides many predefined nodes to start working, such as nodes to perform mathematical operations, audio processing, event listening or texture visualization, but the more powerful feature is the ability to implement your own nodes. Since the module is implemented in JavaScript, the node creation is very straightforward and gets included in the whole module with a simple export operation. Another important feature is the ability to export and import the workflows using a JSON file, which makes possible to replicate the mentioned workflow in any web application by just including the module, the required nodes and opening the file.

![Figure 7: A simple example of the LiteGraph workflow.](image)

I will use LiteGraph as the engine for my project to visualize and create procedural terrains, by implementing custom nodes that perform different noise algorithms and some filters to modify them.
2.4 Similar applications

Some desktop applications already exist that allow creating procedural terrains in a similar way to what I am implementing, by using a visual graph editor. They have been a source of inspiration for my project, but there are several differences since I developed a web tool. I will talk about the two that are more used by the industry.

a) World Machine

World Machine [5] is an advanced desktop application used for creating procedural terrains using a graph-node editor. Its main purpose is to export the necessary assets to recreate the generated terrain in game engines or other applications. Such assets include meshes and its heightmap, procedurally created textures and several data about the erosion of that terrain. The application requires paying for a license in order to unlock all the functionalities.

![Figure 8: A workflow created with World Machine](image)

The application assigns a colour to every node type to make easier building the workflow. Green nodes are used for base terrain generation, blue nodes are for filtering, brown nodes are for applying erosion algorithms and the red for specifying how to export the terrain. There are more advanced nodes that allow applying filters to certain areas, assigning custom colours and even selecting certain paths in the workflow, acting as switches. World Machine is a very powerful application, and it also has a good documentation in form of tutorials and a big community to help anybody build their desired terrain.
The application allows previewing all the generated assets in real time, and also to change several parameters such as size in kilometres or resolution. Its versatility makes it a good choice for relatively big projects and it has several licensing options to make it affordable to all kind of business.

b) Houdini

Houdini [6] is one of the main applications used in the videogame and film industries when it comes to generating procedural content. The first version was released in 1996 and it has been in used in films like Moana (Disney) and in videogames like Ghost Recon (Ubisoft). The application is not only limited to procedural terrain generation, it allows to simulate fire and fluids, particles, volumes, lightning, cloths, fur, crowds and complex physics, rigging of a character and perform animations.

The full workflow of the application is based on nodes. As we can see in the figure 10, the application has a big canvas on the left to preview the current workflow output, on the bottom right it has the actual graph workflow and on the top right it has a panel to change node properties.
Houdini has several licensing plans, since the functionalities are divided depending on the target applications you want to use it for. Houdini Core is used by 3D artists for modelling and animating characters, Houdini FX is used to create special effects for the film industry and Houdini Engine is to integrate the Houdini Assets inside your own applications. There is also a plan for indie developers, where they get a cheaper but limited application, also licensing options for students and even a free option for non-commercial projects.
3. PROCEDURAL TERRAIN GENERATION TECHNIQUES

3.1 Common theory

As I explained before, to create a procedural terrain we must first use an algorithm to create a procedural texture (i.e., a heightmap). Textures in computer graphics are nothing more than an array of values with a specific format that specifies how many colour channels it has, usually red, green, blue and alpha, and how these channels are distributed. We will use the variation of one of these colour channels on every pixel, which normally goes from 0 to 1, to apply a height offset on each vertex of a grid of vertices, effectively creating a terrain.

In order to give uniqueness to different generations, pseudo-random number generators are used to produce noise that is then used in some way, depending on the algorithm, to create the texture. These generators will return, in our case, numbers between 0 and 1, and are pseudo-random because they always return the same number for the same input (also called seed), which is useful to replicate a certain result.

The simplest way to create a texture from a pseudo-random number generator is to call that function in each of the pixels of the texture. This will create a texture where each pixel does not depend at all on the other pixels around it, meaning that there is a lot of variation between different pixels regardless of the distance between them. This texture is called “White Noise” and it is not of much use for generating terrain, since in nature random patterns are smooth between different points. That is, points that are close between each other tend to be almost the same, while distant points tend to be very different [8].

Figure 11: White noise texture (left), 10x10 portion of that texture (middle) and the portion scaled back (right)

In the figure 11 we can see a white noise texture on the left. As I said, the colour variation between different pixels is so high that it is not very useful to recreate natural random patterns, such as mountains. But, if instead of using the whole texture, we take a small portion and scale it back using cubic filtering, we now have the desired behaviour: closer points has almost the same colour, while distant points are usually very different. This method is not directly used to create procedural terrains since it has some limitations, but allows me to explain why the three algorithms I am going to use utilize a grid for the noise generation. In order to obtain a smooth-looking pattern from a noise texture, we have to create a grid, assign a random value to each vertex and
interpolate these values between adjacent vertices. The way the grid is used to calculate the final colour for each pixel using a certain interpolation is what differentiates the noise generation algorithms I am going to explain.

a) Frequency

In noise texture generation, frequency refers to the number of rows or columns in the grid (we are creating square textures so they will be the same). In the figure 11, if the middle texture would have been created using a grid of values, the frequency would be 10. For any algorithm, if the texture is small enough and the frequency is very big, the result will be white noise. In the final generated terrain it will determine the density of mountains.

b) Amplitude

Amplitude in this context refers to a value that is multiplied to each of the final pixel colours in the noise texture to increase or decrease its intensity. An amplitude larger than 1.0 will make the final texture whiter, while an amplitude less than 1.0 will make it darker. This will determine the main height variation of the terrain.

c) Octaves

Octaves are used to give more detail to a noise texture. To generate a noise texture with two octaves, first we need to calculate the colour for this pixel using the noise algorithm, then multiply the base frequency and divide the base amplitude of the noise by two, calculate a new colour using these parameters and sum the two colours.

![Figure 12: Example of noise texture generation using Value Noise algorithm and 4 octaves](image)

For each new octave added, we have to multiply and divide again the frequency and the amplitude by two, making the grid bigger and the colours darker. In the figure 12 we can see this process and the final texture, which has way more granularity than the original. The colours tend to gray because the range goes [-1, 1] and after all the sums it is mapped to the range [0, 1]. The product of applying several octaves is also called Fractal Noise.

d) Offsets

Since the grid has an infinite length, an offset can be applied to it to move the terrain in a certain direction. Also, if an offset big enough is given the generated terrain will be completely different, meaning we can use these offsets as a custom seed.

3.2 Value Noise
The value noise algorithm generates blocky noise patterns that are by themselves not very realistic when applied to terrains, but it gives very good results when a number of octaves are applied, and it is also quiet efficient to calculate. First of all, we need to subdivide the texture with a virtual grid and assign a pseudo-random generated value in the range [0, 1] to each of its vertices. Then, for each pixel of the texture, we check the grid cell it is in and the ‘x’ and ‘y’ offset inside the cell.

Once we have the offset inside the cell of the pixel, we have to perform three linear interpolations (lerp), one for the values in the ‘x’ axis for the upper vertices, another for the values in the ‘x’ axis for the bottom vertices and a final one for the ‘y’ axis using the two interpolations calculated before. This final interpolation will give us the colour intensity for this pixel.

One thing I did not mention is that before calculating these linear interpolations, the point is usually smoothed out with an “S” shaped function, such as the smoothstep function or the quintic interpolation function. This is used to avoid sharp edges when calculating the colours.

Figure 13: A grid cell, with four random numbers in its vertices and the offset of the pixel we are in (left). Part of Value Noise algorithm (right).

Figure 14: Example of Value Noise with smoothing not applied (left) and smoothing applied (right)
In the figure 14 we can see the result of applying the Value Noise algorithm, with a smoothstep function applied and without smoothing. The smoothed texture is way cleaner and naturally more suited for my purpose of terrain generation.

### 3.3 Perlin Noise

Ken Perlin presented Perlin Noise in 1985 as an innovative way of generating noise textures to be used in the film industry, specifically for the film “Tron”, the first movie that used computer-generated visual effects. The computers of that time were very limited in memory, and noise functions was a way of applying variation to the plain colours they had to use without consuming much memory. The problem was that the textures generated, such as white noise and value noise textures, did not look very natural [9]. Perlin Noise solved this problem and it is still being used nowadays for procedural texture generation.

Conceptually, the algorithm is very similar to the Value Noise algorithm I explained before. First we need to subdivide the texture using a grid, but now instead of assigning a pseudo-random number to each of its vertices, we assign a pseudo-random gradient. In mathematics, the gradient of a function in a point is a vector that indicates the direction of the greatest rate of increase. Since we are working in a 2D space, we just have to create a random vector with two coordinates for each vertex of the grid, but because the algorithm must return a value in the range [0, 1] and not a vector, we need to perform and extra operation to transform a vector to a single scalar. The dot product is an operation that multiplies two vectors and returns the cosine of the angle between them if the vectors were normalized, so this function can be used to retrieve our desired value, we only need another vector. Ken Perlin proposed to use the vectors that go from the current pixel position inside the grid cell to each of the corners of that cell.

```plaintext
vec2 u = vec2(0.3, 0.7); // Offset point
u = fmod(u, (3.0-2.0radians)); // Smoothed

// Upper
var x0 = lerp(dot(grad0, vec0), dot(grad1, vec1), u.x);

// Bottom
var x1 = lerp(dot(grad2, vec2), dot(grad3, vec3), u.x);

// Y-axis
var y = lerp(x0, x1, u.y);
```

![Figure 15: A grid cell, with four random gradients in its vertices, the offset of the pixel we are in and the vectors from the offset to the vertices (left). Part of Perlin Noise algorithm (right).](image)

In the figure 15 we can see a graphical representation of that process. The purple lines represent the vectors from the offset position in the cell to each of the corners, and the green arrows represent the gradient vectors. First we have to smoothly interpolate the offset point, as I explained before, to avoid sharp edges in the texture. I am using the smoothstep function in this case. The lerp operations are exactly the same that we saw...
in the Value Noise algorithm, but now we take the dot product between these vectors instead. We first calculate the linear interpolation in the ‘x’ axis between the top corners, then between the bottom corners and lastly another linear interpolation between these two values using the ‘y’ axis offset. Since the dot product returns 0 when the vectors are perpendicular and 1 when they are parallel, this operation will produce darker colours in the former case and whiter in the second.

![Figure 16: Example of noise texture generation using Perlin Noise algorithm and 4 octaves](image)

In the figure 16 we can see the result of the Perlin Noise algorithm, and the fractal texture it produces after applying 4 octaves. We can clearly see how this algorithm produces a more smoothed result compared to Value Noise.

### 3.4 Worley Noise

Worley Noise, also known as Cellular Noise, was presented by Steven Worley in 1996 in a paper called “A Cellular Texture Basis Function” [10]. It is called cellular because it resembles the disposition of cellules, which makes it useful for texturing stones and water, but also for terrain generation.

This algorithm is based on distance fields. Conceptually it is very simple, we have a set of points scattered over a texture and, for every pixel, we loop over this set and save the distance to the closest point. If we are very close to a point the resulting colour will be darker and if we are far away the colour will be whiter. The problem is that for a large set of points, this can be very inefficient, since every pixel needs to be compared to all the points. Worley proposed to subdivide the texture using a grid, just as the previous noise algorithms, and place a random point in each of the cells. This way, for a given pixel offset inside a cell we only have to check with the 8 cells around the current one.

```cpp
def worley_noise(x, y):
    dist0 = 1.0;  // Minimum distance
    vec2 offset = currentOffsetInCell;
    for (int y = -1; y <= 1; y++) {
        for (int x = -1; x <= 1; x++) {
            vec2 neighbor = vec2(x, y);
            // Point in neighbor cell
            vec2 point = pointInNeighbor(currentCell, neighbor);
            // Vector between the pixel and the point
            vec2 diff = neighbor - point + offset;
            // Distance to the point
            float dist1 = length(diff);
            // Save closer distance
            dist0 = min(dist0, dist1);
        }
    }
    return dist0;
```

![Figure 17: A grid with a random point in each cell, the offset of the pixel we are in and an arrow indicating the closer point (left). Part of Worley Noise algorithm (right).](image)
In the figure 17 we can see a visual representation of this process. The green point is the current pixel we are going to render, the blue dots are the random generated points in each cell and the purple arrow is a vector from the pixel to the closer point, from which the length will be taken. The code in the right shows exactly that, using two for loops we go through all the neighbours, comparing the length to each of the points and saving the closer distance.

An interesting thing that can be done with this algorithm is to change how the distance is calculated, which will return cellular noises with different patterns. For example, instead of saving the distance to the closer point, we can save the distance to the two closest points and then return the subtraction of these two distances.

![Figure 18: The original cellular noise (left) and cellular noise with distances subtraction (right)](image)

In the figure 18 we can see the different patterns that we can obtain by just changing how the distances are used. The two heightmaps were created using the same seed and parameters.

### 3.5 Deformation filters

Deformation filters are used to apply certain effects to the base heightmap texture we already created, changing the resulting terrain. Normally, it will consist on performing a certain operation for each the pixels in the texture. Here I will explain the filters I have implemented in the editor.

**a) Pow Filter**

This filter performs an exponentiation operation with a certain exponent for each of the pixels in the texture. Higher exponents (above 1) push middle elevations down into valleys and lower exponents (below 0) pull middle elevations up towards mountain peaks [11]. The operation is $x^y$, where “$x$” is the color intensity and “$y$” is the exponent.
As we can see in the figure 19 the texture becomes darker, increasing the number of valleys and lakes.

b) Mix Filter

The mix filter performs a mix operation between the pixels of two different heightmaps, resulting in a fusion of the two original textures that depends on a certain threshold in the range \([0, 1]\). The resulting heightmap will be more similar to the first texture provided when the threshold is closer to 0, and more similar to the second when it is closer to 1.

\[
x \times (1 - a) + y \times a.
\]

Figure 20: The mix function [12]

In the figure 20 we can see the formula of the mix function, where ‘x’ and ‘y’ are, in this case, each the two heightmaps and ‘a’ is the threshold.

In the figure 21 we can see the result of mixing two heightmaps with a 0.5 threshold, meaning that the pixels from the two base textures are mixed equally. Note how both original patterns are present in the result.
c) Noise Filter

This filter creates a white noise texture that is later added to the base heightmap, resulting in a heightmap with bigger granularity. Bigger impact can be achieved with a higher amplitude (i.e. colour intensity) of the pixels, while a bigger frequency will increase the granularity.

![Cellular noise (left) and cellular noise with noise filter applied (right)](image)

In the figure 22 we can see the result of adding a noise filter to a heightmap, note how the resulting texture has a bigger granularity.

d) Clamp Filter

This filter clamps the colour intensities of the pixels in the original texture between a certain range. Since the range of the colour intensity in each pixel is [0, 1], it makes sense to give a range that is inside this one.

![Cellular noise (left) and cellular noise with a clamp in the range [0.2, 0.8] (right)](image)

In the figure 23, we can observe how all the colour intensities outside the clamp range are lost.
e) Remap Filter

The remap filter maps the original pixel colour intensity range from [0, 1] to another desired range. It can be used to globally increase or decrease the intensity of all pixels without losing information.

```c
float map(float value, float inMin, float inMax, float outMin, float outMax) {
    return outMin + (outMax - outMin) * (value - inMin) / (inMax - inMin);
}
```

Figure 24: Remap function

In the figure 24 we can see how the remap function is performed. The function takes as input the value to be mapped, the original range and the target range.

![Figure 25: Cellular noise (left) and cellular noise with a remap in the range [0.2, 0.8] (right)](image)

Note how, as opposed to the clamp filter, all the values have been mapped to the new range [0.2, 0.9] without losing the colour intensities that were outside.

f) Blur filter

This filter performs a blur over the base heightmap, making the slopes of the final terrain way softer. The technique consists on averaging each of the pixels with its neighbours within a certain radius.
We can observe in the figure 26 how the heightmap in the left has been blurred.

**g) Invert Filter**

This filter takes a heightmap texture and inverts the colour values of all the pixels inside. In case of having a terrain full of valleys, the output would be a terrain full of mountains. The operation is “color = 1.0 – color”.

In the figure 27 we can see the original heightmap on the left and the resulting texture after inverting the colours on the right.

**h) Perturbation filter**

This filter performs a domain warping over a texture. Warping or domain distortion is a very common technique in computer graphics for generating procedural textures and geometry. It's often used to pinch an object, stretch it, twist it, bend it, make it thicker or apply any deformation you want [13]. The technique consists on applying an offset obtained procedurally to the current pixel to be read. That is, we use the base heightmap
texture to apply an offset to the current pixel to be read, meaning that every offset will be different.

![Figure 28: Cellular noise (left) and cellular noise with perturbation applied (right)](image)

We can see in the figure 28 how the texture has been twisted, giving a pattern that is similar to marble. This technique is very good to generate interesting terrains.

**i) Join filters**

This filter performs a linear interpolation between two heightmaps in a certain offset position and within a certain range, resulting in a new texture were part is from the first heightmap and another part is from the second, with an area where they are mixed smoothly. It can be done horizontally and vertically.

![Figure 29: Example of two heightmaps joined together](image)

In the figure 29 we can see the result of mixing the two heightmaps. The position where they are mixed can be specified with a custom offset and also the range to be mixed. The linear interpolation operation is the same that was explained in the “Mix Filter”.

**j) Max and min filters**

These filters receive two heightmaps as input and perform a max or a min operation respectively between each of the pixels colour intensities, resulting in a texture were both heightmaps are mixed together.
In the figure 30 we can see the result (in the right) of applying a min filter between two heightmaps.

**k) Sum and Sub filters**

These filters receive again two heightmaps as input and perform an addition or subtraction respectively between the colour intensities of the each of the pixels from both textures. A threshold is used to control how much addition or subtraction is applied between them.

In the figure 31 we can see the result (in the right) of applying a sum filter between two heightmaps.

**l) Lerp Mask Filter**

This filter performs a mix operation between two heightmaps, but instead of providing a custom threshold to interpolate all the pixels, the colour intensities of a third texture are used instead.
As we can see in the figure 32, the lerp mask determines the final mix operation between the base textures.

**m) Height and Slope Colour Filters**

The height and slope colour filters allow giving up to four custom colours to the final terrain, depending on the height or slope of each vertex respectively. A “dispersion” property is used to make the join between colours smoother. To calculate the slope, I calculate the normal map at this point and then I perform the dot product with the up vector, which gives me the cosine of the desired angle.

In the figure 33 we can see how the filter produces the different colours. The height is divided in four parts, one for each colour. For the first part (the lowest) I used red, for the next I used green, then blue and finally white.
4. IMPLEMENTATION

4.1 Web tool design

In this section I will explain how I implemented the web tool, from the base mathematical and graphics framework to the custom LiteGraph nodes and shaders to perform the procedural algorithms.

In the figure 34 we can see the interface of the web tool. The main background canvas is used to create the graph workflow and the top right canvas is used to preview the currently generated terrain. The terrain preview canvas allows you to move the camera around freely. The three textures in the top are, from left to right, the resulting heightmap texture, the normal map and the colour map. The generation of the normal map and the colour map will be explained later. These textures can be downloaded to be used in external applications by right clicking at them. At the top we have the toolbar which have, from left to right, the “Calculate” button used to recalculate the textures from the current state of the graph, the “Center Camera” button used to restore the camera to the initial position, the “Wireframe” switch used to display the current mesh wireframe, an input field to change the terrain size, a combo-box to select different sample terrains and the “Save” and “Load” buttons, used to save and load already created graph workflows.

To make a working workflow, the web tool needs at least one generation node (Perlin Noise, Value Noise or Cellular Noise) and an Output node connected. The output node is in charge of taking the final heightmap after all the filters have been applied and calculate the normal map and the colour map (in case it was not calculated before) out of it.

4.2 Base Framework

Every application able to render 3D content in real time needs a base framework to perform mathematical operations and to communicate with the GPU. Operations between vectors and matrices are crucial for camera movements and to transform a set
of vertices to be visible in the screen using WebGL, as I said in a previous section. When I was starting the project I decided to implement this base framework myself for two reasons: it will serve me as a learning experience and I will have full control over what I am doing. Now I will give a brief overview of the objects I implemented for this framework.

**a) Vector objects**

I implemented three vector objects, one with two components (vec2), one with three components (vec3) and another with four components (vec4). They allow you to perform addition, subtraction, dot operations, normalization and in case of vec3 also the cross product. These objects are used to store all the terrain vertex data that is uploaded to the shader.

**b) The matrix object**

I implemented a 4x4 matrix object that allows creating translation, rotation and scale matrices, building a perspective projection matrix and also a view matrix through a “lookAt” function.

I have a static method that builds a perspective projection matrix with the given parameters and a “lookAt” function to build the view matrix. Both matrices are used for transforming mesh vertices in order to be rendered as I explained in the section about WebGL.

Figure 35: Perspective matrix (left) and view matrix (right) [14]

In the figure 35 we can see how to create both matrices. The perspective matrix needs the field of view angle (FOV), the aspect ratio of the target resolution (width/height) and a far and near plane, which defines the minimum and maximum distance the camera can see. The view matrix needs an up vector, which normally has the direction (0, 1, 0), the look vector, which is the direction the camera is looking to, and the right vector, which is perpendicular to the previous ones.

**c) Camera object**

After understanding the view and projection matrices, the camera object is straightforward to implement. We just need to instantiate the view and projection matrices, modify the view matrix whenever the camera is moved and multiply both matrices when we want to update the camera state.
In the figure 36 we can see the methods used to change the “front” vector of the camera, allowing me to change the direction it is facing by providing the yaw and pitch angle. I use the yaw and pitch angles to change the direction of the camera when the mouse is dragged over the canvas. To change the camera position I just need to translate the “eye” point. Every time the camera status wants to be updated we just have to multiply the view an projection matrices.

**d) WebGL buffers**

In order to communicate with the GPU, WebGL provides functions to generate buffers for uploading data. There are many types of buffers depending on the data that is uploaded, but the syntax is the same.

```javascript
Buffer {
  create() {
    this.vboId = gl.createBuffer();
    gl.bindBuffer(bufferType, this.vboId);
    gl.bufferData(bufferType, data, drawType);
  }

  bind() {
    gl.bindBuffer(bufferType, this.vboId);
  }

  unbind() {
    gl.bindBuffer(bufferType, null);
  }
}
```

Table 1: Buffer object pseudocode

As we can see in the table 1, first we call the function “gl.createBuffer”, which creates a buffer and returns an id to let us bind this buffer whenever we want. Then we bind this buffer and upload the data. The buffer type variable is normally “gl.ARRAY_BUFFER” when uploading vertex data, and “gl.ELEMENT_BUFFER” uploading the indices of a mesh (I will explain this later). The “drawType” parameter is used as a hint for WebGL to tell if we are going to change the contents of the buffer very often or not.

There is an special object called ArrayBuffer which is used as a wrapper for several buffers.
e) Texture object

Textures objects are used to upload texture data to the GPU and to store information about them. I use this object to store the heightmaps, normal maps and colour maps used in the application.

First we create the texture with “gl.createTexture”, which returns an id for this texture, and then we bind this texture and upload the data with the specific format. The calls to “gl.texParameteri” are used to specify how the texture will be scaled when the target size is different to the texture size, and how to behave when we read of the texture bounds. In this case we scale using a linear interpolation, and use the value at the edge when reading out of bounds for every texture.

f) Shader object

As I explained in the section about WebGL, shaders are programs that are executed in the GPU that interprets the data provided in order to render the content we want. I will be using vertex shaders, who are executed for every vertex of a mesh, and fragment shaders, which are executed for every pixel in the target resolution. Since I perform all the procedural generation algorithms in the GPU, this is one of the more important objects in my application. The algorithms are implemented as fragment shaders to benefit from the efficiency of the GPU, as we will see later.

Since the shaders are stored as separate text files, and JavaScript load files asynchronously to prevent stalling the web page, I had to create a callback that tells me when both vertex and fragment shaders have been loaded and another for when they are compiled. After they are loaded a function is called to compile them.

The syntax to compile the shaders is very similar to the ones seen in the WebGL object pseudocode. First we create both vertex and fragment shaders, we attach the source code and then we compile it. After they are compiled they are linked together in a shader program that can be used to render our meshes.

I save all the shaders in a shader map that is used to prevent compiling the same shader more than once. I have “getShader” static function, which just checks if the shader has already been compiled, and returns the corresponding instance. Since I precompile all shaders when the web page is loaded, all the shaders are ready when they are required.

To precompile all shaders first I register the shader files names, to let me know which and how many shaders will be required to precompile and then I precompile them at the same time. This is used to avoid compiling a shader in a middle of an execution, which stalls the application until it is compiled.

g) Framebuffer

In computer graphics, framebuffers are used to draw the current frame into a texture instead of drawing it directly to the screen. This texture can later be used to apply post-processing effects. I use a framebuffer on each of my node implementations from generation nodes to filter nodes, since I perform all the algorithms in fragment shaders
and I never want to render the texture to the screen. All the filter nodes are nothing more than fragment shaders that use a framebuffer and the base heightmap texture to apply post-processing effects.

The framebuffer object receives the size it is going to have, the texture we are rendering into, the shader we are going to use to render and an uniforms callback used to upload custom variables to the shader. Again, the creation process is very similar to previous WebGL objects. We create the framebuffer and then we bind it and attach the texture object we are rendering to. We also have to define a quad that fills the framebuffer to render the texture into it.

After the framebuffer is created, we can call the render function to actually fill the texture. The function clears first the texture date if there was any, and then makes use of the uniforms provided, the shader and the quad data to render the frame.

Since all the texture data is in the GPU, in order to display the texture as an HTML image I need to use the function “gl.readPixels” to retrieve the texture data from the GPU. After that, I just simply create a 2D canvas with the texture data and then save it as a JavaScript image that I can later use with HTML to display the heightmap, normal map and colour map in the web page.

4.3 Custom LiteGraph nodes

As I explained in a previous section, LiteGraph is a very powerful graph engine that allows extending its functionality very easily.

```javascript
// node constructor class
function MyNode() {
    // Define input and output parameters

}

// name to show
MyNode.title = "My Node";

// function to call when the node is executed
MyNode.prototype.onExecute = function() {

    // Execute the desired operations

}

// register in the system
LiteGraph.registerNodeType("heightmap/myNode", MyNode);
```

Table 2: A template for LiteGraph custom node creation

In order to create a custom node, as seen in the table 2, we just have to create an object that implements an “onExecute” prototype method, add a title name and then register as a type in the engine. After this process, the node type will show up in the nodes menu of the engine.

For the generation nodes, that is, the nodes that performs the Perlin Noise, Value Noise and Cellular Noise algorithms the code is exactly the same, since the input and output
parameters are identical, the only thing that changes is the shader used to perform the algorithm and the name of the objects.

In order to let know all the custom nodes how the data needs to be interpreted, I created a custom object called “heightmapObj” which contains the heightmap, normal map and colour map textures, with their size and the height scale that is being applied to the terrain. This object is the output parameter for all the nodes I have implemented.

In the “onExecute” method I first take all the node input parameters, which are the same that were specified in the procedural generation section, setting a default value in case no value is specified. Then I define a uniform callback function, which will always consist on uploading these input parameters to the shader to be later used in the algorithm, and finally I create the texture we are going to render into, using the provided size, and I pass it to a framebuffer. The framebuffer receives this texture and also the shader it is going to use to generate the texture and the uniforms callback. To conclude I render the framebuffer and I set it to the “heightmapOBJ” custom object, before exporting it as the output parameter of the node.

The “onExecute” method is called every frame, so to avoid recalculating the textures when no change occurred, I calculate a hash in every node every time a property changes. This way I know if any property is different and the node can be executed. The hashes are also used to change the “Calculate” button colour to hint when the textures need to be recalculated.

This functionality is exactly the same for all custom nodes, including the filter nodes. The nodes receive some input parameters, create a uniform callback to upload them to the shader, and they render using the custom shader to create, in case of generation nodes, or modify, in case of filter nodes, a texture.

**a) Output Node**

This node is a special case, since it is the responsible of generating the normal map and the colour map. The node receives the heightmap after all the filters have been applied and also four colours that will be used to calculate the final colour texture.

![Heightmap, Normal Map, Colour Map](image)

To calculate the normal map, the horizontal and vertical slope of each of the pixels is calculated, generating a vector with the neighbour pixels. When the final vector points closer to \((1, 0, 0)\) the colour will change to red, when it points to \((0, 1, 0)\) it will change to green and when it points to \((0, 0, 1)\) it will change to blue.
To calculate the colour map I use the four provided colours and apply several mix operations between them depending on the height so they are gradually changed.

4.4 Generation shaders

As I said before, all algorithms are implemented using shaders to benefit from the efficiency of the GPU. JavaScript as an interpreted language would take way more time to perform all these algorithms.

a) Common vertex shader

The vertex shader is responsible of processing all the vertex data and then passing this data interpolated to the fragment shader. Since its purpose in our case is quite trivial, I use the same vertex shader for most of the cases, with the exception of the shader for terrain rendering, which I will explain later.

```glsl
layout (location = 0) in vec2 aVertex;
layout (location = 1) in vec2 aUvs;
out vec2 oUvs;
void main(void)
{
oUvs = aUvs;
gl_Position = vec4( aVertex, 0.0, 1.0 );
}
```

Table 3: Common vertex shader

In the table 3 we can see this vertex shader. It receives the vertices and texture coordinates and exports them to the fragment shader. No matrix multiplication is done since there is no camera for the generation and filter shaders.

Filter nodes use the same custom LiteGraph node structure and the same common vertex shader. They are post-processing effects that receive the base heightmap as input and perform some operations to modify it. Each of the operations are defined in the section about filters.

b) Common fragment functions

Since all the generation shaders use the same input parameters, in the fragment shader the way to increase the frequency, the amplitude and the octaves between them is the same.

```glsl
uniform int u_octaves;
uniform float u_amplitude;
uniform float u_frequency;
uniform float u_perturbation;
uniform float u_xOffset;
uniform float u_yOffset;

// Fractional Brownian Motion, generates fractal noise
float fbm(in vec2 uv) {
```
float f = 0.0;

// Apply offset to generation
uv.x += u_xOffset;
uv.y += u_yOffset;

// Apply frequency
uv *= u_frequency;

float amplitude = u_amplitude;
for (int i = 0; i < u_octaves; i++) {
    f += amplitude * noise( uv );
    uv = 2.0*uv;
    amplitude /= 2.0;
}
return f;

void main() {
    // Calculate perturbation offset
    vec2 q = vec2(fbm(oUvs + vec2(0.0,0.0) ),
                    fbm(oUvs + vec2(5.2,1.3)));

    // Retreive color intensity with perturbation
    float c = fbm(oUvs + u_perturbation&q);

    // Map to range [0, 1]
    c = 0.5 + 0.5*c;

    fragColor = vec4(c, c, c, 1.0);
}

Table 4: Common fragment shader functions

In the table 4 we can see the uniforms, which are also shared between all generation shaders, the main function, which is responsible of applying the perturbation filter to the noise and the Fractional Brownian Motion function, which is responsible of applying the offsets, the frequency, the amplitude and the number of octaves for the current noise function. All these properties and the following noise functions are explained in the section about procedural generation.

4.5 Terrain Object

The terrain object is the responsible of receiving the heightmap, the normal map and the colour map and generating the mesh used to preview the terrain.

A 3d mesh is formed from a set of vertices connected between them, usually forming triangles. Since we are using a heightmap to displace the mesh, we need to create a flat mesh with at least the same number of vertices as pixels there is in the heightmap, or some information will be lost. The first to do then after receiving all the heightmap information is to create an array of vec3 with a length equal to the width times the height of the texture (which are the same in our case). The texture coordinates can be easily calculated by dividing the current vertex coordinate position by the mesh size. Since I am creating a flat mesh, the “y” component is 0 for all the vertices. This coordinate is the one that will be displaced later. After this I calculate the center and the radius to place correctly the camera.
Since WebGL does not have support for displaying the wireframe of a mesh, I had to manually implement it myself. To do this, I assign to each vertex a barycentric point \([15]\). These points are either \((1, 0, 0)\), \((0, 1, 0)\) or \((0, 0, 1)\) and in the shader I make use of the “fwidth” function, which returns the sum of the absolute value of the partial derivatives over the point. One way to understand how they work is to visualize the mesh using the barycentric points as colours.

![Figure 38: Barycentric points displayed as colours](image)

As we can see in the figure 38, the colour difference between different vertices is the maximum. After applying the “fwidth” function over the interpolated barypoint and a smooth function, the result is the desired wireframe.

In order to store as minimum data as possible, I make use of indices and the “TRIANGLE_STRIP” primitive to avoid repeating vertices in the mesh. With this primitive a triangle is created with the first three indices and then a new one is added for each index, meaning that two triangles can be created with only four indices (instead of 6 if I use the “TRIANGLES” primitive). The problem with this technique is that when we reach a border of the mesh we have to create two degenerated triangles (a triangle whose area is 0 and is discarded by WebGL) in order to continue.

![Figure 39: Creating indices for TRIANGLE_STRIP using degenerated triangles](image)

As seen in the figure 39, the trick is to repeat the last index in the right border and the first index of the left border in the next row.
a) Use of heightmap, normal map and colour map

After uploading all the data to the GPU, I use the heightmap in the vertex shader to displace the “y” coordinate of each vertex, multiplied by a height scale.

In the fragment shader, I implement the phong shading model to apply realistic illumination to the terrain. This shading model makes use of the normal map to calculate how light impacts on each of the pixels. The colour map is used as a base diffuse texture.

4.6 Renderer object

The renderer object is in charge of precompiling all the shaders used in the web tool, keeping the instances of the terrain and origin axes meshes and rendering them every frame. It also initializes the WebGL context, setting the clear colour and the blend equation. The WebGL context itself is created as a global variable before any script is loaded, to make sure all the objects have access to it, including the LiteGraph nodes.

4.7 Editor object

The editor object is the main object of the application. It is the responsible of instantiating the camera object, the LiteGraph object, the renderer object and a sample workflow, along with taking control of mouse dragging and clicking in the preview canvas. The “init” function of this object is the first one to be called when the web is loaded. It also takes care about the behaviour when the different buttons in the toolbar are pressed.
5. Conclusions and future work

5.1 Loading in Unreal Engine

Unreal Engine is one of the most used videogame engines, both by companies and enthusiast. The engine’s workflow is based on graphs (called Blueprints), where different functionality can be tied together in a simpler way.

![Figure 40: Unreal Engine’s tool to import heightmaps](image)

The engine has a direct way to load heightmaps, just by selecting the Landscape tool and providing the desired texture. As we can see in the figure 40, the engine automatically gives the right properties to the terrain based on the heightmap imported.

![Figure 41: Generated terrain after loading heightmap in Unreal Engine](image)
In the figure 41 we can see the result of loading the heightmap. In order to provide the normal map and the colour map, we have to create a “material”. Materials are the way to apply textures and to indicate how they behave in Unreal. To create a material, simply import the colour and normal maps as project assets, then right click in the colour map and then “Create Material”. The material can then be easily edited to also take into account the normal map.

![Figure 42: Terrain rendering in Unreal Engine after colour map and normal map are applied](image)

In the figure 42 we can see the final terrain rendering after all the textures have been used. This terrain can then be used in any part of a videogame by an artist.

### 5.2 Conclusions

This project was quite enriching for me, since it pushed me to learn some advanced algorithms that I never thought I would learn. There are many things I did not have time to implement, but I am proud of what I achieved to do. The web tool allows creating procedural terrains with a decent amount of detail and exporting the data necessary to replicate this terrain in other applications.
In the figure 43 we have an example of what can be achieved after combining several noise generators and filters. The amount of terrains that can be created is infinite.

Also, I would like to mention that thanks to how LiteGraph is implemented, the workflows generated in my web editor can be loaded in other web tools using LiteGraph, just by importing my custom nodes and my WebGL objects.

There is a web page where the web tool can be tested, the name is: “https://pzerua.github.io/”.

5.2 Future work

There are many things I did not have time to implement. For example, It would be great to export the terrain data directly and not through heightmap textures. There are also many more filters that can be implemented, as many as post-processing effect techniques exist, but not all of them are useful for terrain generation. An effect I would have loved to implement if I had more time is the erosion filter. I read some paper explaining some techniques and it would be something great to implement in the future. There are also more noise generation algorithms, such as the Diamond Square algorithm that could be implemented.

Giving the ability to save how the light impacts on each of the pixels can be useful to apply light saving some calculations. It would be also useful to have an option to display in terrain preview in an orthographic projection.

I am currently saving pixel colours as unsigned bytes, meaning that the final colour intensities will only have 255 heights. This can cause artifacts when exporting the heightmap to other editor.

Giving more freedom with the light, for example allowing to move the light source or to add more light sources could have been a nice feature.
Overall, I feel that these kinds of applications are never really finished. I am happy with the final result of the web tool and I hope to continue with its development in the near future to make it even a more useful tool for developers.
Bibliography


