CRANFIELD UNIVERSITY

MIGUEL MARQUES

A WEB BASED TERRAIN MODELLER USING FRACTALS AND CAD

SCHOOL OF AEROSPACE, TRANSPORT AND MANUFACTURING
Computational and Software Techniques in Engineering
Digital Signal and Image Processing

MSc
Academic Year: 2015–2016

Supervisor: Dr Peter Sherar
August 2016

CRANFIELD UNIVERSITY

SCHOOL OF AEROSPACE, TRANSPORT AND MANUFACTURING
Computational and Software Techniques in Engineering
Digital Signal and Image Processing

MSc

Academic Year: 2015–2016

MIGUEL MARQUES

A Web Based Terrain Modeller using Fractals and CAD

Supervisor: Dr Peter Sherar
August 2016

This thesis is submitted in partial fulfilment of the requirements for the degree of MSc.

ABSTRACT

The process of modelling man made objects is, classically, a manual process where a user deforms a 3D object though the use of several techniques. This process is time-consuming and, with the usage of procedural techniques, can be improved, enabling the creation of more realistic and detailed content. One such case where this techniques have been widely used is in the creation of virtual terrains, not only because of the realistic details, but also due to the possibility of creating huge random infinite terrains that are used in the video game industry. Procedural generation has the disadvantage of loss of detailed control over the modelled terrain, that is, using a purely procedural method it is complicated to specify where the major features of the terrain should be positioned. This is the main problem that this thesis addresses.

In this project a hybrid terrain modelling process is proposed. In this process the user creates a simplified version of the desired terrain containing the basic topology, which is then blended with a random generated surface to create a detailed version of the terrain using image processing techniques. Additionally a web application, where this process is implemented, was developed using *Angular.js* and *WebGL*. The architecture for this application is detailed in this thesis. In order to improve the performance of the random generation and the blending process some GPU computations were implemented using *WebGL 2.0*. This computations were then benchmarked and the comparison with their CPU counterparts is shown in this document.

Keywords

Procedural Generation; Terrain Modelling; Web Application; WebGL; Fractal geometry; GPU; Fourier filtering

ACKNOWLEDGEMENTS

- The author would like to thank

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF EQUATIONS

LIST OF ABBREVIATIONS

fBm	Fractional Brownian Motion
HPF	High-pass Filter
FFT	Fast Fourier Transform
IFFT	Inverse Fast Fourier Transform

1 INTRODUCTION

Procedural generation enables designers to create content algorithmically instead of manually. This technique is used in entertainment areas, such as, 3D animation and video game design.

One particular case in which procedural generation is used is the creation of random terrains, which is the focus of this thesis.

1.1 Problem

The main advantage of procedural content against manually created content is that, the former, is less time consuming and, as a result, enables designers to create detailed content faster. On the other hand the existent methods for random terrain generation are difficult to control and, sometimes, the parameters are unintuitive, which leads to a trial and error design approach, in which the designer changes the parameters and sees what happens.

1.2 Main Requirements

The main requirements for this application can be divided in technical and functional requirements.

In terms of technical requirements the application must be developed using web based technologies and the random surface generation process must be based in fractal geometry. For the functional requirements, the process must use a CAD surface as a base for the terrain and the details should be generated procedurally.

1.3 Proposed Solution

In this thesis a hybrid approach to content creation is proposed. This approach involves the modelling of a deterministic cad model that provides the basic topology of the terrain and the generation of a random terrain that will provide the details for that topology. This two surfaces are then blended together using a method that is explained in Chapter 3. The main focus of this project is the implementa-

tion of the blending process and, due to this fact, the developed application does not deal with the cad modelling phase, but only with the random generation and blending phases.

1.4 Thesis Outline

This thesis is divided in 6 Chapters. In Chapter 2 some mathematical background is presented and the current available procedural terrain generation methods are reviewed. In Chapter 3 the proposed method and the parameters that are available to the user are explained. In Chapter 4 the architecture of the developed system is specified with the usage of the ”4+1” View Model proposed in [1]. In Chapter 5 some outputs of the system are presented and the measurements obtained by the performance benchmarks are discussed. Finally in Chapter 6 the implemented features and some further work are discussed.

2 LITERATURE REVIEW

Procedural generation of terrains has been the focus for many graphics researchers for some time now. Through the years this research has been mostly based on fractional Brownian motion and its similarity to a skyline of mountains, first noticed by Mandelbrot in [2].

2.1 Mathematical Background

2.1.1 Fractal Dimension

> "A fractal is by definition a set for which the Hausdorff Besicovitch dimension strictly exceeds the topological dimension." [2, p.15]

The Hausdorff Besicovitch dimension, also called fractional or fractal dimension, is a measure used to characterize a fractal. This dimension D_f does not need to be an integer and, for a surface in $\mathbb{R}^n$, $D_f \in [n, n+1)$.

2.1.2 Fractional Brownian Motion

Fractional Brownian Motion (fBm) was introduced by Mandelbrot and van Ness in [3] and it is an extension of Brownian Motion that uses the Hurst Exponent (H) as a parameter to control the correlation between successive values, such that, $0 < H < 1$. More specifically if $H = 0.5$ then fBm is just normal Brownian Motion and, due to that, the increments are independent and not correlated; if $H > 0.5$ the increments have positive correlation and this results in smooth curves while if $H < 0.5$ the increments have negative correlation and this results in erratic curves. [4]

In the field of fractal terrain generation fBm is approximated by $1/f^{\beta}$ noise, where β is the spectral exponent of the noise. Considering D_f as the fractal dimension and D_E as the Euclidean dimension, $1/f^{\beta}$ noise follows the following rule:

$$D_f = D_E + 1 - H = D_E + \frac{3-\beta}{2}$$

2.2 Terrain Modelling

In this section the different methods used for terrain generation are analysed. All the analysed methods try to approximate fBms and they are divided in six categories:

- Poisson Faulting
- Subdivision Methods
- Fourier Filtering
- Successive Random Additions
- Noise synthesis
- Generalized Stochastic Subdivision

2.2.1 Poisson Faulting

The Poisson Faulting technique, also known as Random cuts algorithm, consists in applying Gaussian random displacements to a plane at Poisson distributed intervals. [6]. This method was initially applied by Mandelbrot in [2]. Although this method has the advantage of working for both planes and spheres, in the creation of random planets, it is $O(n^3)$ complex in time.

2.2.2 Subdivision Methods

The methods presented in this section derive from the midpoint displacement algorithm that consists in successively subdividing a line and displacing the division points. These methods are classified in two categories [7]:

- Wire-frame Midpoint Displacement
- Tile Midpoint Displacement

2.2.2.1 Wire-frame Midpoint Displacement

In this type of methods the surface is thought of as a wire-frame mesh and the displacements are applied to the midpoints of the edges. This is the case of the Carpenter's Method [8] in which the wire-frame forms triangles which are recursively subdivided until there is no triangle with a side bigger than a specified length. Wire-frame methods are context independent as the only inputs that impact the shape of the generated surface come from the altitude values at the vertices. This context independence is the opposite of what happens with fBm, which incorporate an infinite span of context dependence [7].

2.2.2.2 Tile Midpoint Displacement

In tile midpoint displacement methods the surface is seen as a collection of tiles and the displacements are applied to points in the middle of every tile. This methods are context dependent.

Triangular tile midpoint displacement In [7] Mandelbrot proposes a method of tile displacement with triangles. In contrast with the Carpenter's method (section 2.2.2.1), the displacement is applied to the midpoint of the triangles using:

$$H(P) = \frac{H(A) + H(B) + H(C)}{3} + rand$$

where A, B and C are the triangle vertices, P is the triangle midpoint and $rand$ is the random displacement value.

Diamond-Square Algorithm The Diamond-Square algorithm [8] consists in the subdivision of quadrilaterals in two steps: in the first step, known as diamond, the midpoint of the square is displaced using a random value; in the second step, known as square, the midpoint of the original square sides are interpolated from the value of the two square vertices and the two closest diamond vertices and displaced by another random value. In practice this is a hybrid between wire-

frame and tile displacement method as when the initial structure is composed of squares it is not enough to just displace the tiles midpoints but also interpolate the edges midpoints.

Square-square subdivision In [10] Miller proposes a method adapted from the CAD/CAM field. This method, denominated Square-Square subdivision, generates new points that form a square which is half the size of the existing one using the proportions 9:3:3:1, in which the nearest points have the greater weight.

Hexagonal tile midpoint displacement In [7] Mandelbrot also proposes the use of hexagonal tiles in the displacement method. This was due to his belief that the nesting properties of the displacement methods, that is when the generated points are nested in the old structure, was the cause of the creasing effect. Given the hexagon's properties, a structure like this will never nest but, instead, will create a crumpled boundary that fails to catch the eye, contrary to the "creases" that stand out.

2.2.3 Fourier Filtering

Another method to generate fBm surfaces is using the Inverse Fourier Transform. This is done by generating a two dimensional Gaussian white noise signal, applying a $1/f^{\beta}$ low-pass filter in the frequency domain, and using the result of the Inverse Fourier Transform of the filtered signal as a height field [6].

2.2.4 Successive Random Additions

The Successive Random Additions algorithm [11] builds on top of the midpoint displacement algorithm. If old points are reused in subsequent phases of subdivisions, they are displaced with a random variable with an appropriate distribution [6]. In terms of complexity this method is comprised of approximately twice the number of additions of the midpoint displacement algorithm [12].

2.2.5 Noise synthesis

Noise synthesis consists in the addition of successive frequencies of tightly band-limited noises [6]. This can be done using several noise algorithms, such as, Perlin Noise [13], Simplex Nosie [14] or OpenSimplex Noise [15].

2.2.6 Generalized Stochastic Subdivision

All the methods previously presented have a basis function that is, usually, implicit in the generation algorithm. This basis function can affect the final surface in different ways: the saw-tooth wave in polygon subdivision methods explains the creasing effect and the sine wave in Fourier synthesis causes the terrains to become periodic. In [9] Lewis proposes a method that interpolates several local points based on a autocorrelation function [4], that is, this algorithm is able to generate surfaces with any basis function.

2.3 Terrain Representation

2.3.1 Height field

The height field, in some literature denominated as height map, is one of the most used forms of representing terrains in Computer Graphics. A height field stores an altitude value at regular intervals using a two-dimensional array [5], and, as such, can be saved as a grayscale image. Due to the fact that only one altitude value is retained for a pair of coordinates this method can only represent surfaces, that is, it does not support overhangs or caves.

3 METHODOLOGY

- Write Chapter Introduction

3.1 Process Overview

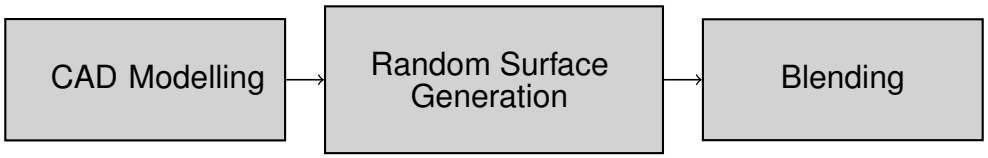

Figure 3-1 Process Phases

The proposed workflow consists of three high level phases which are presented in Figure 3-1. The first is the CAD modelling phase (section 3.2) which consists on the creation of a simplified model of the terrain, which will be denominated as base surface, containing only the main characteristics, such as, mountains and valleys. The second phase (section 3.3) consists in the generation of the random surface from which the details will be acquired. Finally the third and final phase is the blending phase (section 3.4) in which the base surface is combined with the random surface to create a more detailed version of the terrain.

3.2 Phase 1: CAD Modelling

The purpose of the CAD Modelling phase is to enabled the user to specify the main features of the terrain, such as mountains and valleys, using manual modelling. This phase results in the generation of a height map that contains the base surface.

3.3 Phase 2: Random Surface Generation

With the base surface obtained from the first phase it is now possible to generate a random surface with the same dimensions. This can be done using several

methods, as discussed in Chapter 2. For the purpose of this application the chosen methods were Fourier Filtering (section 3.3.1) and Noise Synthesis (section 3.3.2).

3.3.1 Fourier Filtering

The fourier filtering method, shown in Figure 3-2, consists in applying a $f^{-\beta}$ filter to a white noise surface in frequency domain.

This method only needs one parameter, the filter power (β).

Using a fork and join approach it is possible to parallelize this method as there is a parallel version of each of it's steps.

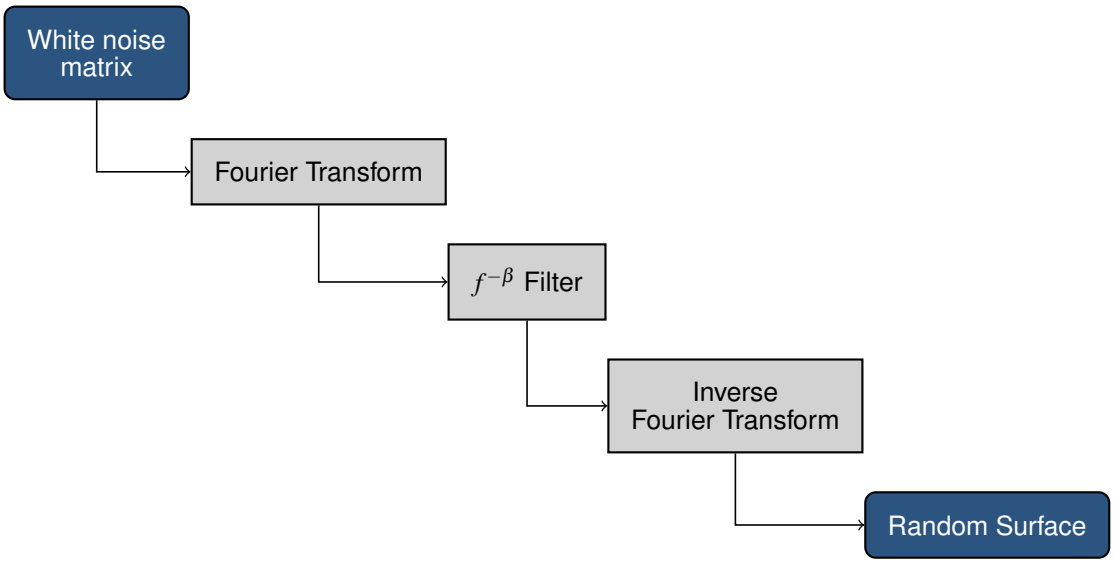

Figure 3-2 Fourier Filtering

3.3.2 Noise Synthesis

$$\frac{\sum_{i=0}^{octaves-1} noise(x \times lacunarity^i + base, y \times lacunarity^i + base) \times persistence^i}{\sum_{i=0}^{octaves-1} persistence^i} \quad (3\text{-}1)$$

The noise synthesis method consists on using a weighted average of several random values to approximate an fBm surface, as described in equation 3-1. This assumes that the random number generation is a structured noise function.

The implementation for this method is based on the code presented in [4]. There are several parameters used in the method, namely:

- **Frequency:** noise arguments divider
- **Number of octaves:** number of noise samples to use
- **Persistence:** contribution gap between successive octaves
- **Lacunarity:** gap between successive frequencies
- **Base:** base value for noise arguments
- **Noise:** noise function used to generate random numbers (eg: perlin noise [13], simplex noise [14])

3.4 Phase 3: Blending

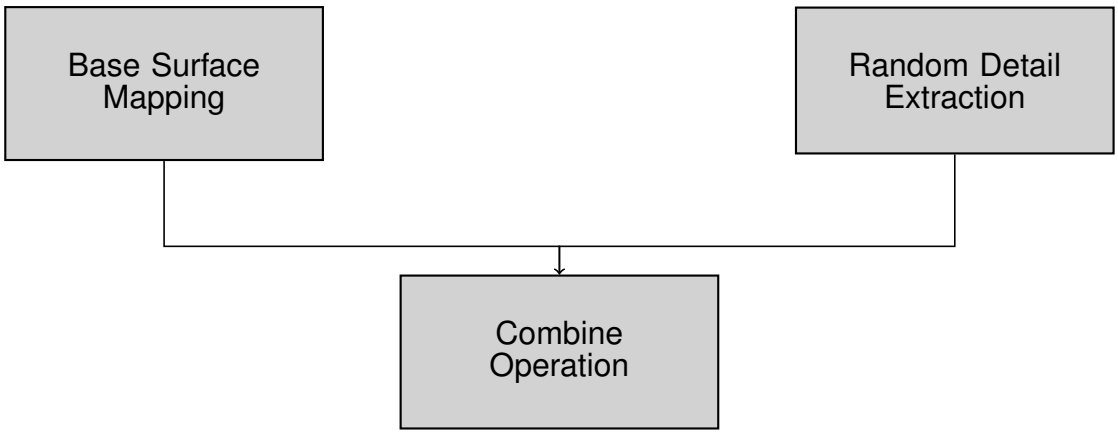

Figure 3-3 Blending Phase

The Blending phase is divided in three parts that are shown in Figure 3-3.

3.4.1 Base Surface Mapping

In the Base Surface Mapping part the user has the possibility of adjusting the details that will be added to the base surface. This is done by manipulating a

spline which represents a function that maps a height value to a multiplier. This mapping is then applied to the base surface generating a multiplier matrix.
The spline function is obtained using a Picewise Monotone Hermite Cubic Interpolation of the control points specified by the user.

3.4.2 Random Detail Extraction

In order to extract the details from the random surface, a high-pass filter is used. In the case of this application the HPF is implemented by subtracting a smoothed version of the surface to the surface itself. The smoothed version is obtained by applying a Gaussian Blur filter in frequency domain, using, as cut-off frequency, a value calculated from a user-specified parameter, called Blend Strength, using 3-2.

$$f_{Cut-off} = \frac{0.5}{BlendStrength} \tag{3-2}$$

3.4.3 Combine Operation

Using the outputs from the Base Surface Mapping and the Random Detail Extraction parts, the detailed surface can be computed. This is obtained by multiplying the mapped base surface by the random details resulting in an adjusted version of the details. The latter is then added to the Base surface.

3.4.4 Result Normalization

The final step in the blending phase is to normalize the result. This is done using user-defined bounds that need to be comprised between 0 and 255 in order for the surface to be stored in a single channel image using, for each pixel, an 8 bit unsigned integer.

3.4.5 Surface Normals Calculation

In addition to the previously described steps, a normal map is also computed for rendering purposes. This is done by filtering the final surface with a 3 by 3 Sobel

filter in both x and y directions.

3.5 Parameters Summary

Phase		Name	Description	Type	Minimum	Maximum
Random Surface Generation	**Fourier Filtering**	Filter Power	β value	*float*	1.0	2.9
	Noise Synthesis	Frequency	Noise arguments divider	*integer*	1	2000
		Octaves	Number of noise samples	*integer*	1	10
		Lacunarity	Gap between successive frequencies	*float*	0.01	10.0
		Persistence	Contribution gap between successive octaves	*float*	0.01	10.0
		Base	Base frequency value	float	0.01	20.0
Blending		Spline Control Points	Mapping between original height and detail multiplier.	*float*	0.0	1.0
		Blend Strength	Gaussian Blur control parameter	*integer*	1	500
Result Normalization		Result Bounds	Values between which the final result will be normalized.	*integer*	0	255

Table 3-1 Process Parameters

4 SOFTWARE ARCHITECTURE

- Write small introduction about 4+1 architectural view model

4.1 Use Case View

The developed application enables the user to create a detailed terrain from a deterministic base surface. To achieve this goal several features, that aid the user in this process, were implemented. This features are specified in Table 4-1 in the form of user stories. Given the nature of this application only one actor, called User, is needed.

Code	Name	Description
US-001	Load Base Surface	As a User I want to load a surface from an image so that I can create a detailed version of it.
US-002	Export Result	As a User I want to export the result so that I can use it in other applications.
US-003	Add Detail	As a User I want to add detail to a previously loaded base surface.
US-004	See Result	As a User I want to see my result as I change the parameters so that I can adjust them more easily.
US-005	History	As a User I want to be able to consult a list of previously edited terrains so that I can track my work.
US-006	History Parameters	As a User I want to have access to the parameters used in previously edited terrains.

Table 4-1 User Stories

4.1.1 User Interface

The application was developed using Google's material design [18] for the user interface style. The editor is the main page of the application and it is shown in Figure 4-1. It contains a menu bar (1) where the user has access to the available functions, a details panel (2), which has the parameter controllers, a list with the previously results (3), a panel where the details of the selected previous result

are displayed (4) and a canvas where the preview of the current current terrain is rendered (5).

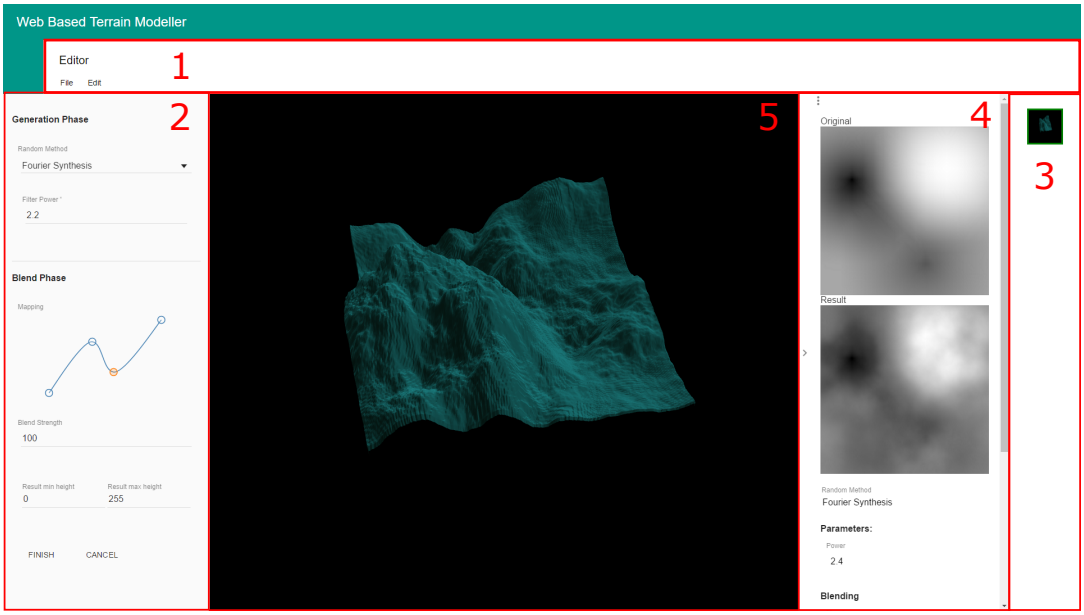

Figure 4-1 Editor Page

4.1.2 Input and Output Formats

In order to allow the user to load and save his terrains the application implements an import and export feature. The user can either import a Grayscale PNG image, which will be set as the base surface, or import a previously exported Zip file, which will add an entry to the history list and render the result contained in the file. In terms of export formats the user can: obtain: a 4-channel 8-bit per channel PNG containing the height map of the result; a Zip file that can be later imported; and a Grayscale 1-channel 16-bit per channel PNG image of the result height map, which can be import as a landscape in Unreal Engine 4.

4.2 Physical Architecture

The system was developed as a client-side single page web application and thus all the server-side content is static. Given this properties the application works as a static web site and has a simple deployment architecture which consists on a HTTP web server and a web browser, as shown in Figure 4-2.

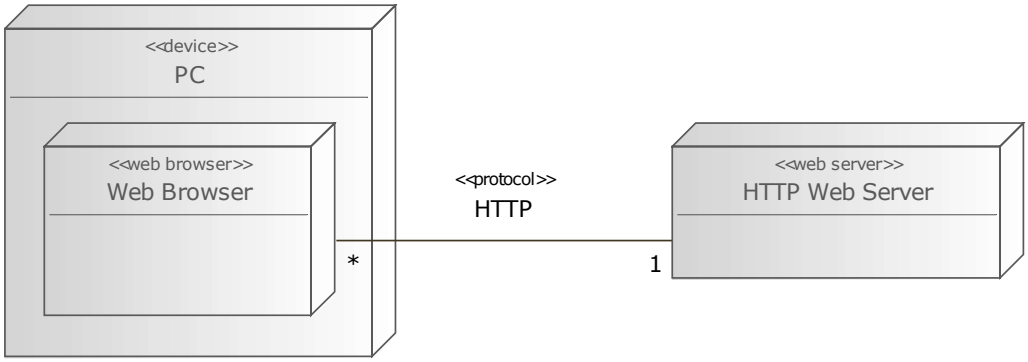

Figure 4-2 Deployment Diagram

4.3 Implementation Architecture

The application is composed by three submodules (Figure 4-3). The *components* submodule is responsible for configuring the page routing and, as such, contains, as submodules, the editor and the benchmarks pages. The *common* module contains services and directives that are generic to the application. The *imgproc* module contains the image processing utilities.

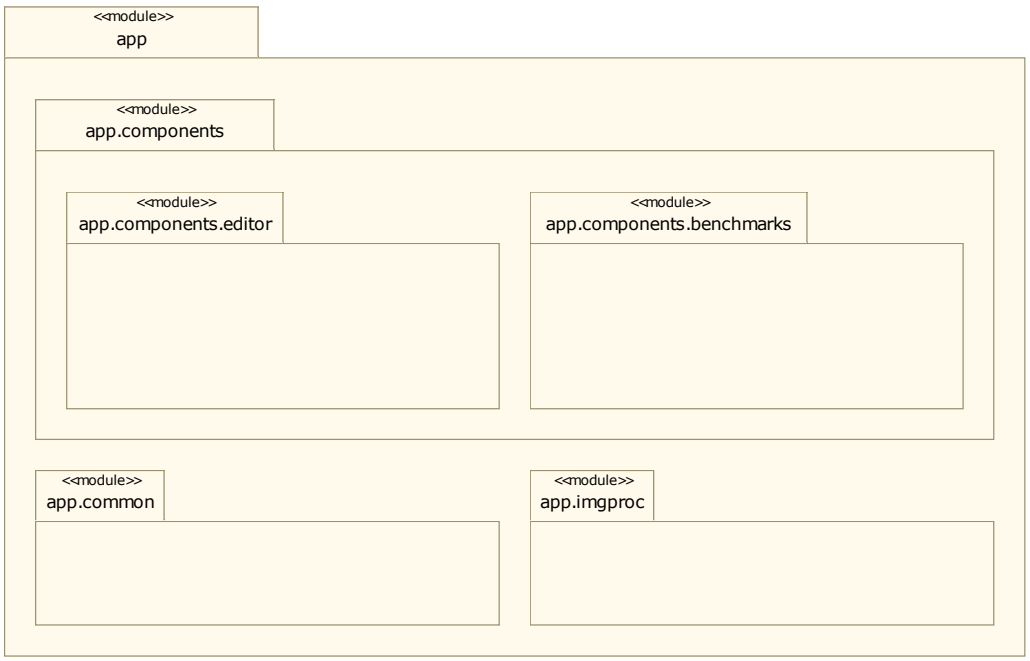

Figure 4-3 Component Diagram

- Fill Component Diagram modules with classes.

4.3.1 Technologies

Given the web-based requirement the application needed to be in Javascript. For convenience the Babel transpiler was used so that the development could be done in ECMAScript 6, which is a newer standard of Javascript. Additionally, to simplify the management of the required dependencies, JSPM and System.js were used, which, integrating with Babel, enable better support for ECMAScript 6 modules.

To aid the development of the application, Angular.js was used. This framework implements the MVC pattern, which was used to better modularize the source code.

The terrain viewer was implemented using three.js, which is a 3D rendering Javascript library that allows the developer to create 3D Scenes in a straight forward way.

In order to parallelize some operations, WebGL 2.0 was used. This implementation will be detailed in section 4.5.1

4.4 Logical View

The logical architecture of the developed application was influenced by the usage of Angular.js. Due to this in Figure 4-4, where the UML class diagram for the application is shown, two prototypes are attributed to the classes: controller and service; the controller classes are responsible by the dynamic behaviour of the element to which they are attributed; the service classes are lazily instantiated singletons that are injected in the application in the required modules.

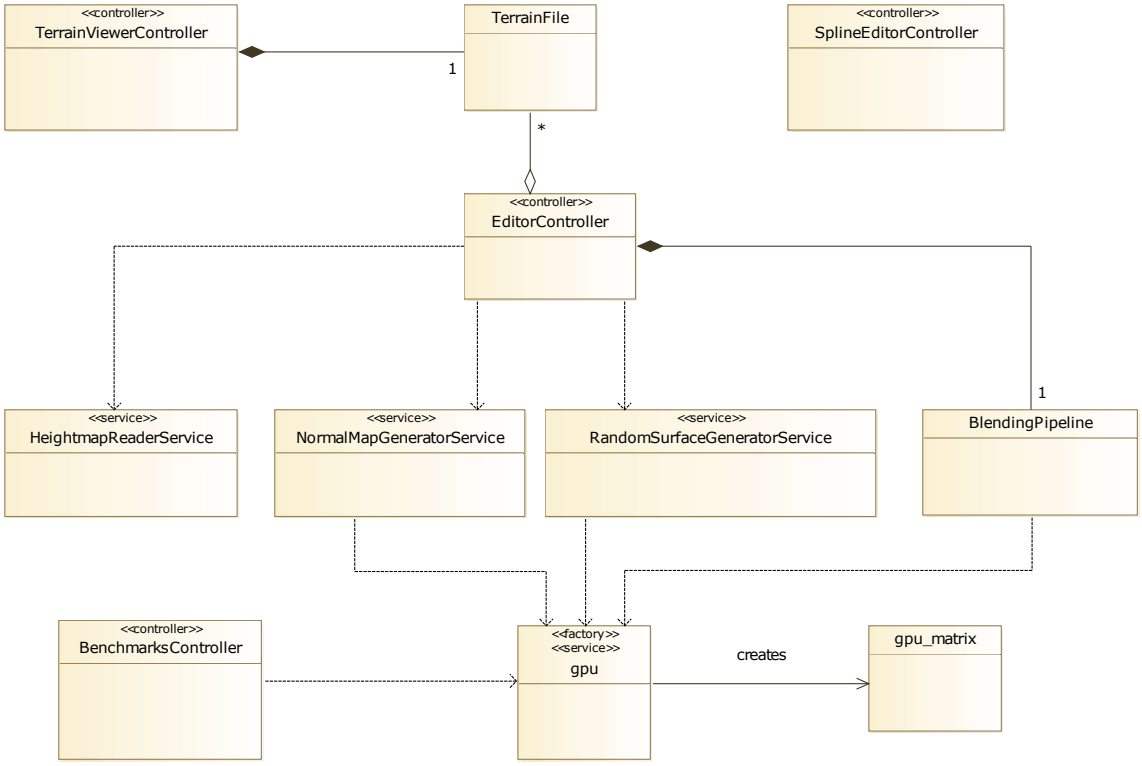

Figure 4-4 Class Diagram

- Fill Class Diagram with main class methods

4.5 Process View

4.5.1 GPU Computation Framework

In order to optimize some operations performed by the application a GPU computation framework was implemented. This framework uses WebGL 2 to perform pixel-wise operations on images. WebGL 2 is a 3D rendering API for the web derived from OpenGL ES 3.0, which features a programmable rendering pipeline that gives the developer the ability to manipulate the rendering data in two different stages: the vertex stage and the fragment stage. The vertex stage executes the vertex shader using as input the vertex data and it outputs the final vertex position as a 4-component floating-point vector in homogeneous coordinates (x, y, z, w). In the fragment stage the fragment shader is executed once per

fragment and it's output depends on the active render target, which can be either the HTML canvas or a texture. When the render target is the HTML canvas the output is 4-component floating-point vector that represents the color of the fragment. When the render target is a texture the output depends on the texture format. For a texture to be a render target it's format needs to be *color-renderable* and, in WebGL 1, as in OpenGL ES 2, only 3 or 4 component integer textures are *color-renderable*. This limitation is surpassed in WebGL 2 and in OpenGL ES 3, where, with the *EXT_color_buffer_float* extension, floating-point textures with 1, 2, 3 or 4 components are considered *color-renderable*.

In order to use WebGL for computations the framework uses a texture as the render target and renders a square on a viewport of the same size as the texture, resulting in a 1:1 fragment to pixel mapping, and uses a specific fragment shader depending on the algorithm that is being executed.

4.5.1.1 Implementation Details

To implement this framework two classes were created: the gpu class and the gpu_matrix class (Figure 4-4). The gpu class works as an Angular.js service which is injected wherever it is needed. At the same time this class is also factory for gpu_matrix class instances, as the latter need an initialized gpu instance to be created, this is noted in the class diagram by the usage of the *factory* prototype and the *creates* relation between the two classes.

The gpu_matrix class represents a matrix that was uploaded to the GPU. Due to restrictions imposed by WebGL the textures dimensions need to be powers of 2 and their format is limited to 1, 2 or 4 components per pixel and 8 bits unsigned integer, 32 bits signed integer or 32 bits floating-point components. This class handles the allocation and reallocation of the textures and framebuffers, as well as, the upload and download of the matrix to and from the GPU device.

The gpu class also contains the methods to execute the algorithms. Although this methods have different parameters depending on the algorithm that they execute,

they all have as last parameter the return variable, which is also returned by the function. This rule enables the framework to reuse gpu_matrix instances when one is passed into a function, otherwise, that is, when the last parameter is *undefined*, a new gpu_matrix is created and returned. Additionally, given that each algorithm requires different shader programs, a lazy initialization for the shader programs was implemented. This feature results in a smaller initialization time as it only compiles the shader programs when they are needed, on the first time a certain function is called.

4.5.1.2 Implemented algorithms

The implemented algorithms can be divided in 3 categories:

- Fast Fourier Transform
- Element-wise Operations
- Matrix Normalization

The Fast Fourier Transform was implemented using the radix-2 Stockham FFT algorithm used in [19]. This algorithm avoids the bit reversal phase of the FFT by reordering the dataflow and, consequently, cannot be performed in-place, which is not a problem for this implementation taking into account that textures in WebGL cannot be read from and written to at the same time. Additionally, as noted in [19], in order to compute a 2D FFT, the matrix is usually transposed to optimize the traversal of an array stored linearly in memory. This issue is also surpassed as, on a GPU, textures are swizzle, thus, no transposes are performed. To implement the Inverse Fast Fourier Transform the FFT implementation was used with the real and imaginary parts swapped as shown in [20, p.450].

The element-wise operations implemented are addition, subtraction, multiplication and division. There are two version of this operations: the binary version, where two matrices are combined using the given operator, and the immediate version, where the operator is applied to matrix and to a given value. Additionally

some image processing algorithms were implemented: two frequency domain filters: the gaussian blur and the $f^{-\beta}$, that were implemented by multiplying the fourier transform of the matrix by the fourier transform of the filter, which was computed using the analytical formulas; and a spatial domain sobel filter, both in x and y, which was implemented by performing a pixel-wise convolution of the matrix.

In order to implement matrix normalization the process was divided in two phases. The first phase consists in finding the minimum and the maximum value of the matrix. The second is the element-wise execution of equation 4-1. To implement the first phase in parallel a reduction method was used. This method consists in first reducing each row to it's minimum and maximum, resulting in a vector of two values, and then reducing the resultant vector two the global minimum and maximum of the matrix.

$$M_{Normalized}(x,y) = \frac{M(x,y) - minimum_M}{maximum_M - minimum_M} \tag{4-1}$$

4.5.2 Blending Process Pipeline

In order to improve the application's performance a pipeline for the blending process was created. This pipeline saves the intermediate matrices of the blending process and reuses them when a parameter changes. This process is illustrated in the activity diagram shown in Figure 4-5, where the initial nodes represent the starting point when the associated parameter is modified. When this process reaches the final node a callback is invoked to trigger an update in the user interface.

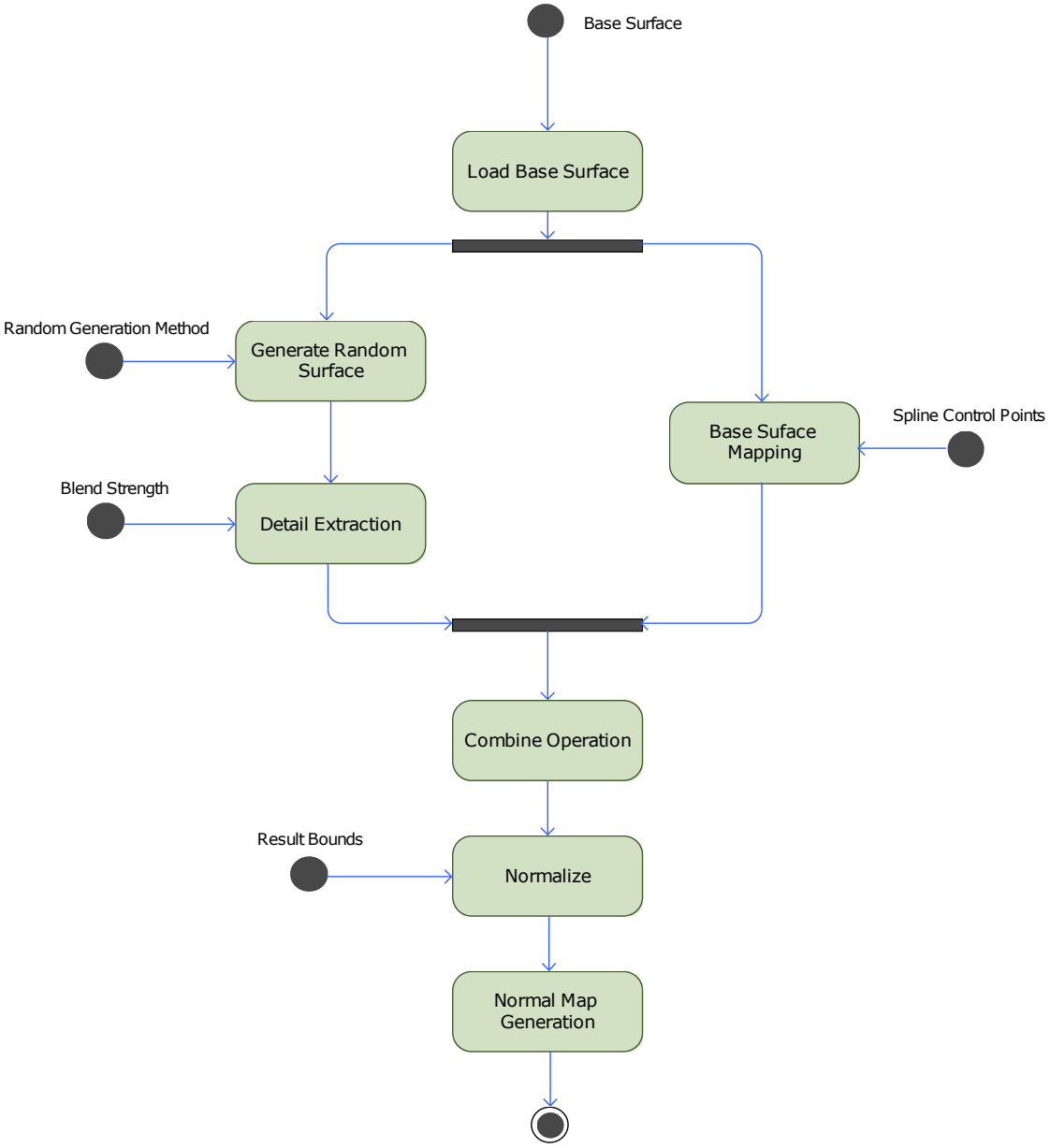

Figure 4-5 Activity diagram for the blending process

5 RESULTS

- Write chapter introduction

5.1 Terrains

- Show some comparison between base surfaces and results with different parameters.

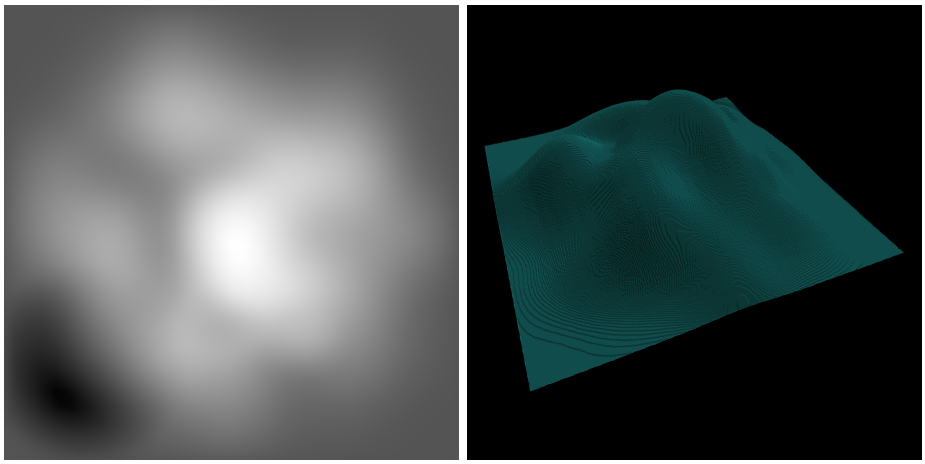

Figure 5-1 Base surface used in the examples

5.1.1 Fourier Filtering Generation

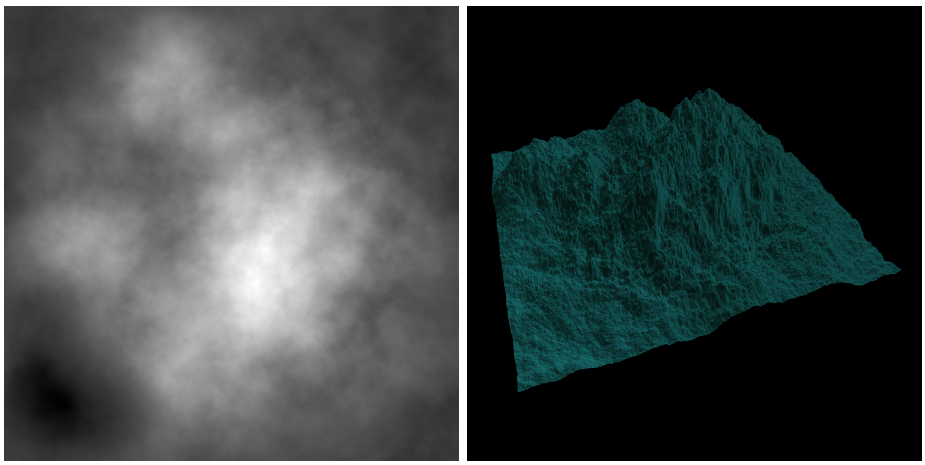

Figure 5-2 $\beta = 1.8$

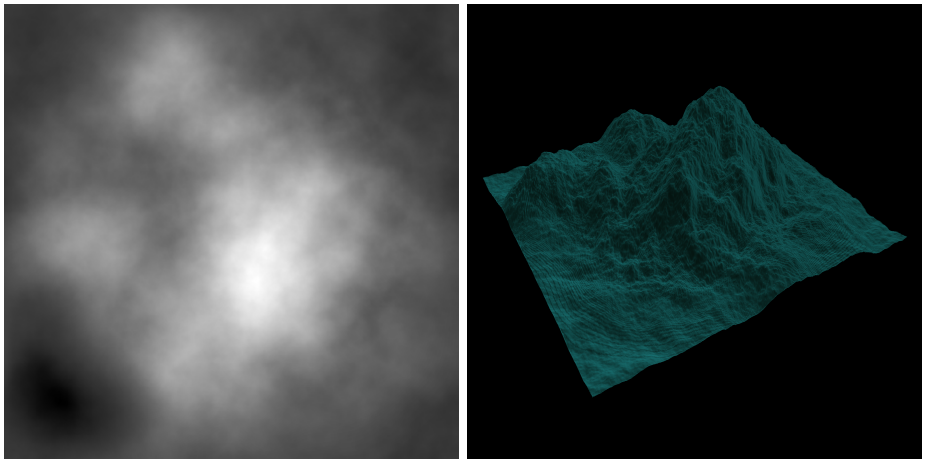

Figure 5-3 $\beta = 2.0$

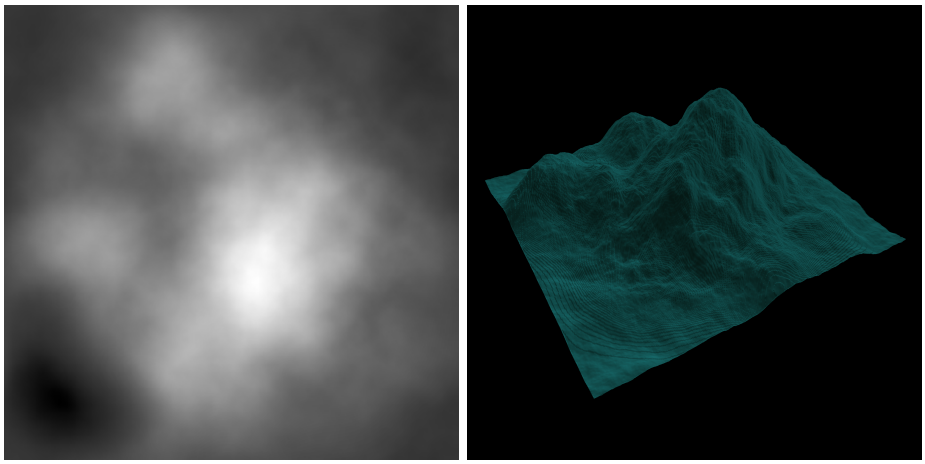

Figure 5-4 $\beta = 2.2$

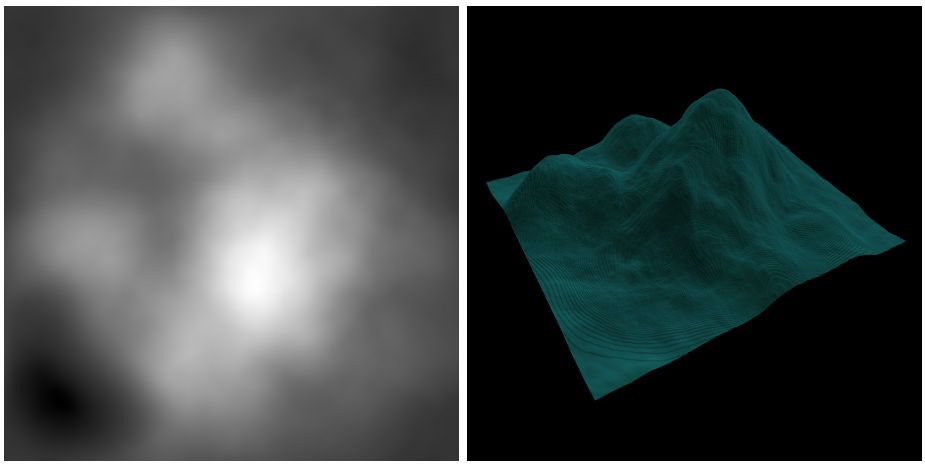

Figure 5-5 $\beta = 2.4$

5.1.2 Noise Synthesis

5.1.2.1 Perlin Noise

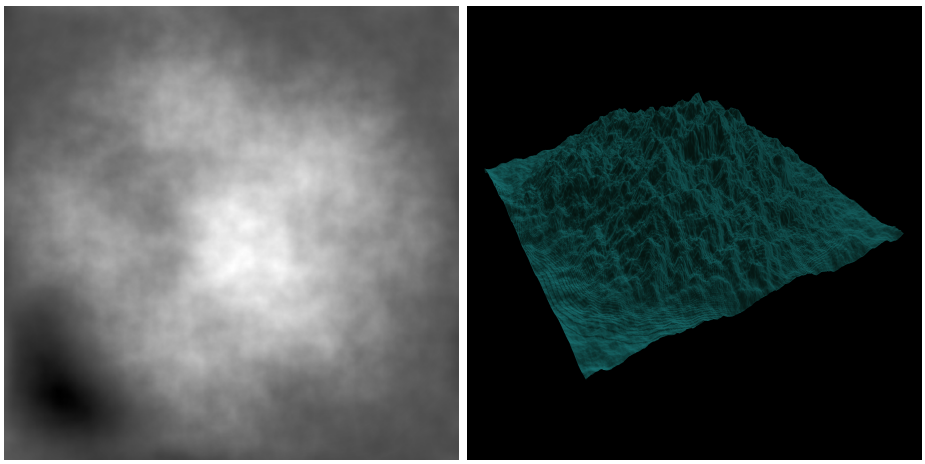

Figure 5-6 $frequency = 120$

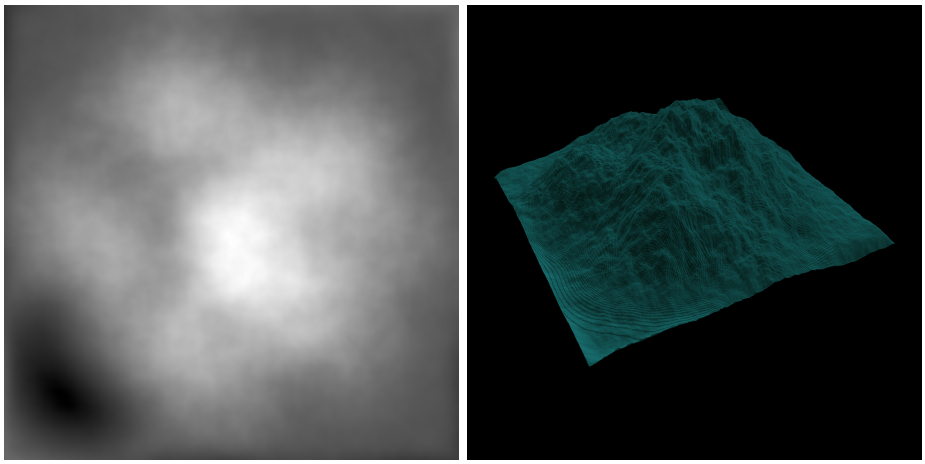

Figure 5-7 $frequency = 240$

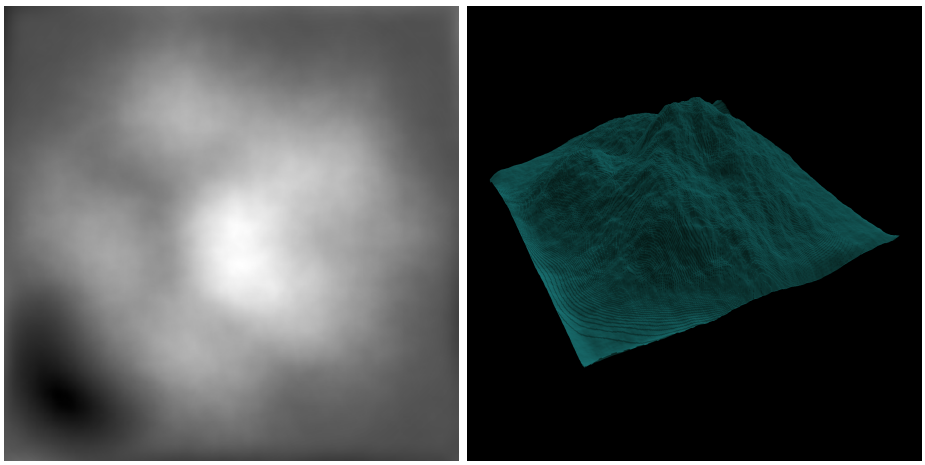

Figure 5-8 $frequency = 360$

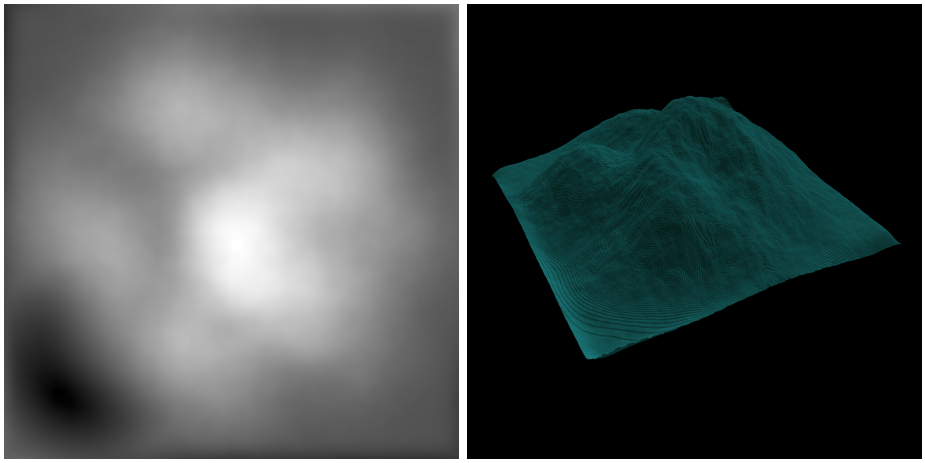

Figure 5-9 $frequency = 480$

5.2 Random Generation Methods Performance

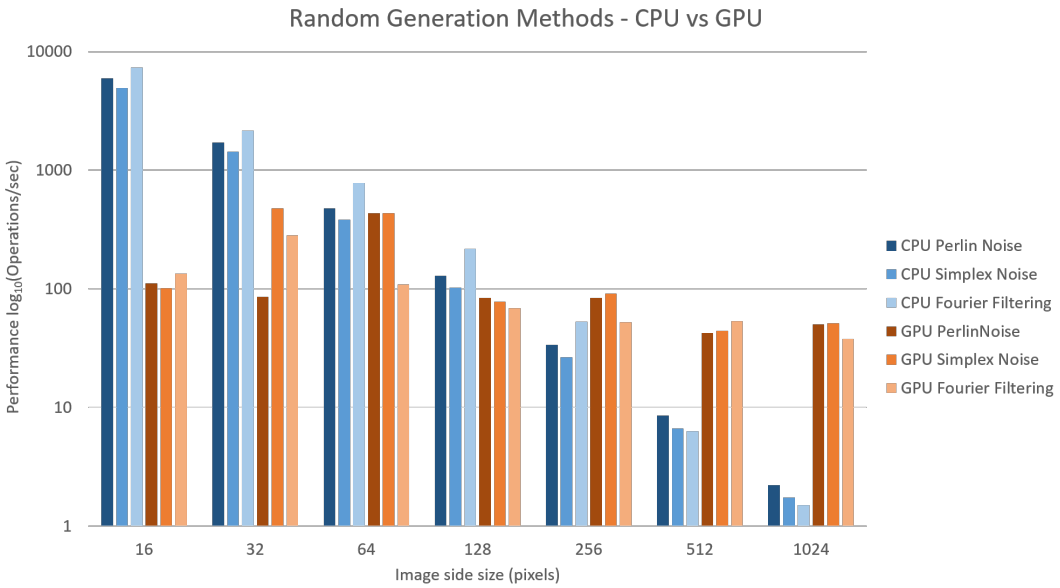

Figure 5-10 Random Generation Methods Performance

With the objective of evaluating the performance from the random generation methods, both when executed in the CPU and in the GPU, some benchmarks were performed. The results from these tests are shown in Figure 5-10 where it can be seen that when the generation size is greater or equal to 256 the GPU implementation is faster. Additionally it is also possible to note that the noise synthesis methods execute faster than the fourier filtering.

5.3 GPU vs CPU Computations Benchmarks

- Should I talk about the big-O for each solution?

In order to evaluate the GPU Computations' performance some benchmarks were implemented and executed. In these benchmarks, the number of operations per second of the GPU implementation is compared with a reference CPU implementation. To be noted that all the measures assume that the required parameters are already in the testing device and the result is to remain in the same device, that is, the upload/download time is not measured. Additionally, due to the lazy initialization of the GPU computations framework, all the operations are initialized before the tests are run.

All the computations were performed using an Intel® Core™ i7-6700HQ CPU with 16 GB of RAM, with a NVIDIA® GeForce® GTX 950M GPU with 2 GB of VRAM, running Microsoft® Windows® 10 build 10586 64 bits, NVIDIA Driver version 365.10 and Google Chrome Version 51[1].

5.3.1 Fast Fourier Transform Benchmark

This test measures the performance of computing a Fast Fourier Transform. The results of this benchmark are shown in Figure 5-11, where for each size of a square matrix the base 10 logarithm of the number of operations per second is plotted, from which it can be seen that for square matrices of size up to 32 pixels, the CPU implementation is faster, while for larger matrices the GPU implementation prevails.

[1] Additionally Google Chrome had the flags *#enable-webgl-draft-extensions* and *#enable-unsafe-es3-apis* active in order for WebGL 2.0 to be available.

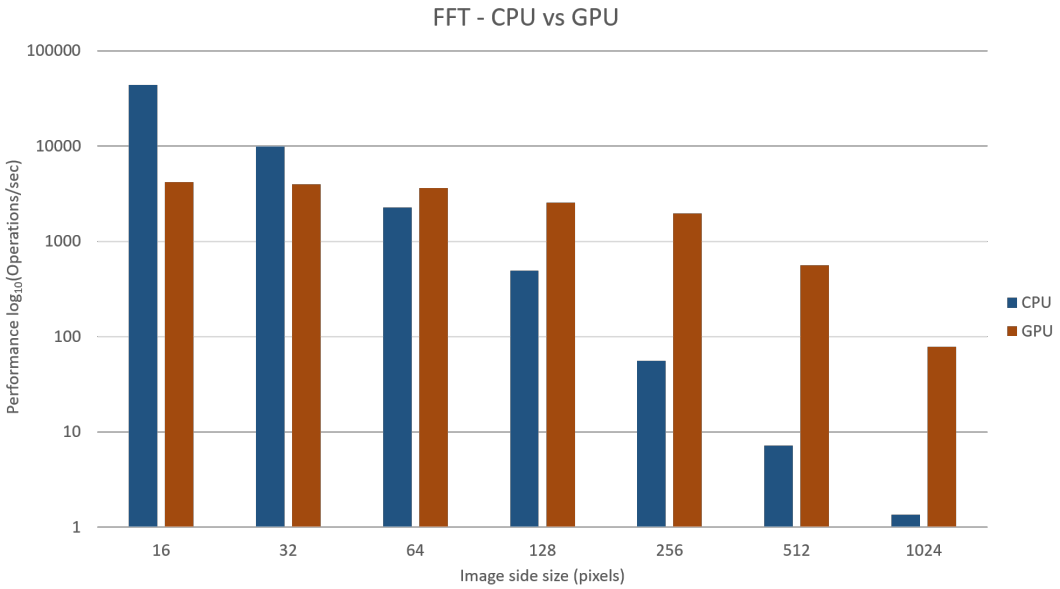

Figure 5-11 FFT benchmarks GPU vs CPU comparison

5.3.2 Element-wise Operations Benchmark

This test measures the performance of the element-wise operations: addition, subtraction and multiplication, when applied to two matrices. The results obtained from these computations can be seen in Figure 5-12, where a plot is shown mapping each size of a square matrix to the base 10 logarithm of the number of operations per second. It can be noted that for square matrices up to 64 pixels the CPU operations are faster or perform the same as their GPU counterparts, while for bigger matrices the GPU implementation is better. Additionally it can also be seen that, in the CPU implementation, the addition operation is faster than the subtraction and multiplication operations.

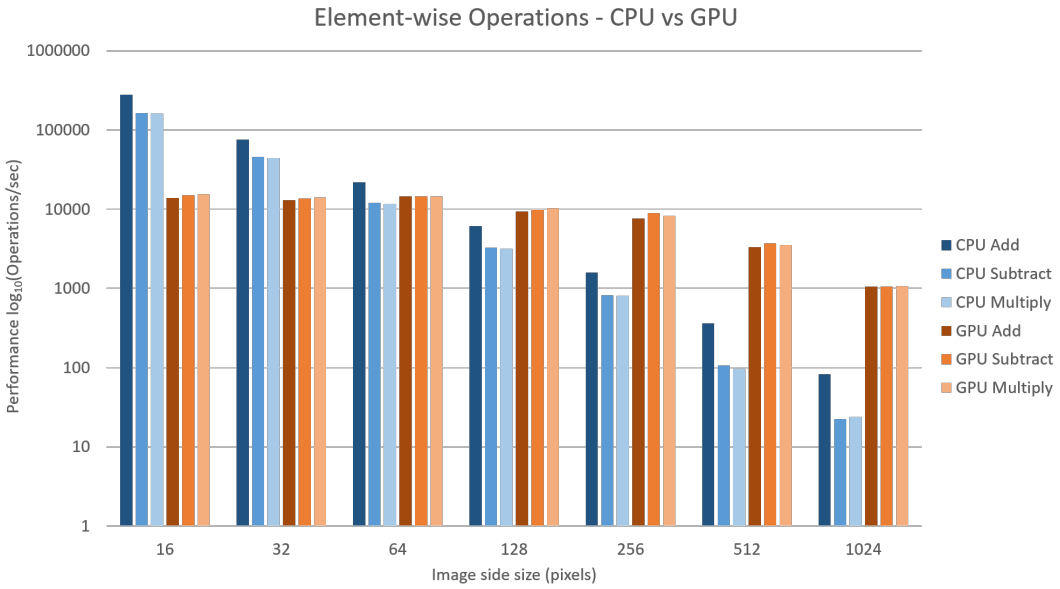

Figure 5-12 Element-wise operations benchmarks GPU vs CPU comparison

5.3.3 Normalization Benchmark

This benchmark aims to measure the performance of the normalization operation. In Figure 5-13 a plot representing the results in logarithmic scale is shown. From this plot it can be seen that the GPU implementation surpasses the CPU for square matrices with size 512 or 1024, while the CPU implementation is faster for smaller matrices.

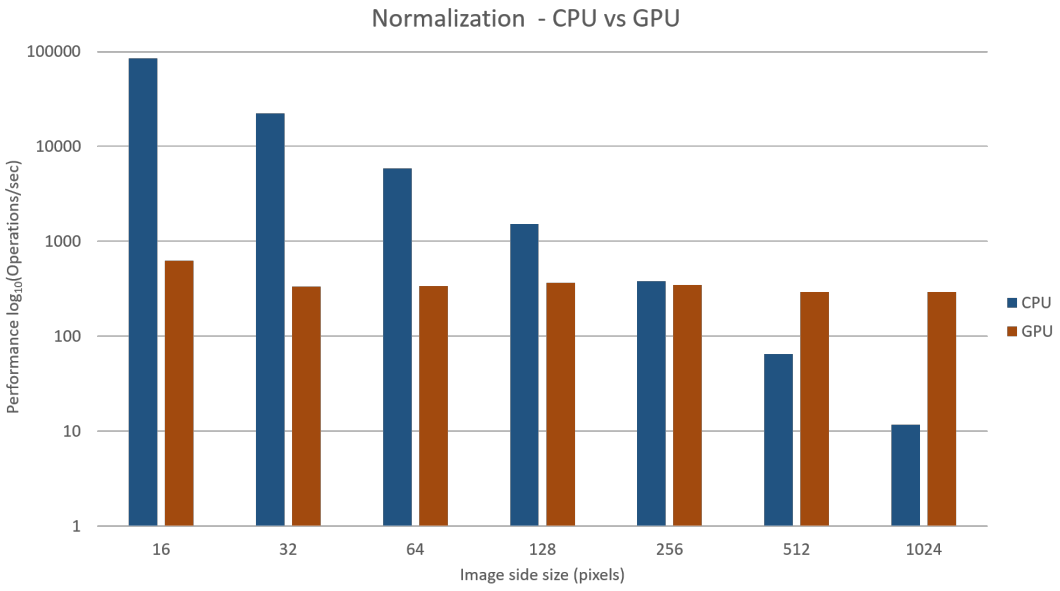

Figure 5-13 Normalization benchmarks GPU vs CPU comparison

5.3.4 Benchmark Summary

In summary, for square matrices with size greater or equal to 256 the GPU implementations perform better than it's CPU counterparts. This fact can be noted in Figure 5-14 where the ratio of the GPU over the CPU performances is shown, in base 10 logarithmic scale, for each size and for each benchmark executed.

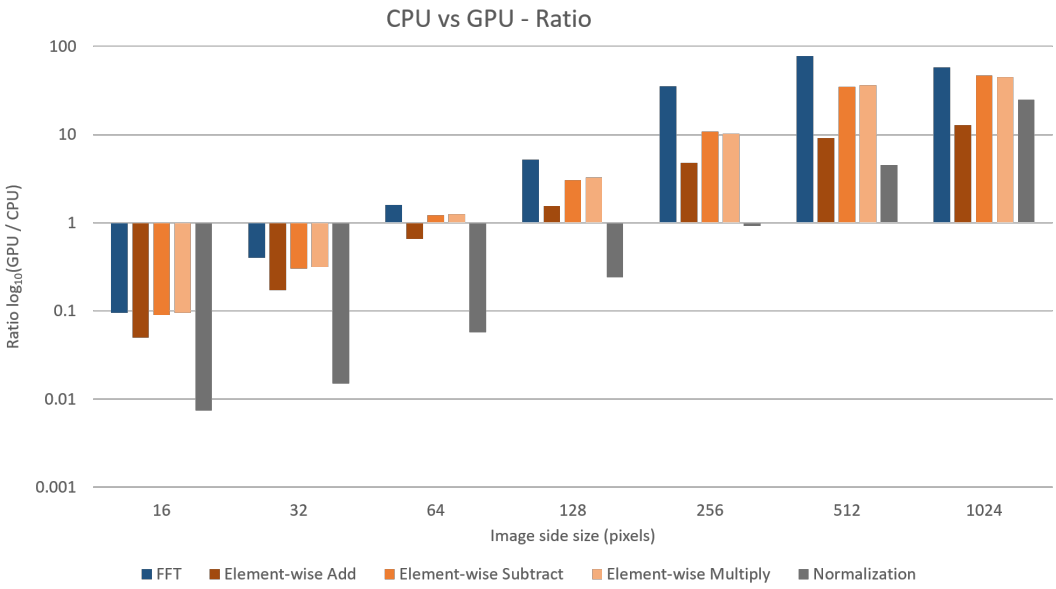

Figure 5-14 GPU vs CPU ratios

6 CONCLUSIONS

In this thesis a procedural terrain modelling method was proposed in the form of a hybrid process, in which, the 3d modeller creates a simplified base terrain comprised of the major features, and the developed system adds the high level details by generating a random surface, extracting it's details and adding them to the base terrain.

6.1 Implemented Features

The system was implemented as a client-side single page web application using *Angular.JS* and *WebGL*. The surface enhancement process was developed, as well as, optimized using WebGL 2.0 as a platform to perform some computations and, as a result, some calculations are now performed between 10 and 20 times faster than with the CPU implementation. In order to preview the resultant terrain the *three.js* library was used to render the terrain in an HTML 5 canvas. Furthermore the results can be exported as height maps or in a zip format supported by the implemented system which can be reimported and re-edited.

6.2 Further Work

Although the implemented system is capable of performing all the required operations some additional improvements could be researched, such as, other detail extraction methods, for example using different filters; usage of the base surface gradient in the blending process, more specifically, in the base surface mapping step in conjunction with the base surface height; additionally some more tools could be given to the user in order to provide more control over where to add details in the surface, as well as, to enable the usage of different method in distinct regions of the terrain.

REFERENCES

1. Kruchten P. Architectural BlueprintsThe 4+1 View Model of Software Architecture. IEEE Software. 1995; 12(6): 42–50.

2. Mandelbrot BB. The Fractal Geometry of Nature. 1983.

3. Mandelbrot BB, Ness JV. Fractional Brownian motions, fractional noises and applications. SIAM review. 1968; Available at: http://epubs.siam.org/doi/abs/10.1137/1010093

4. Musgrave FK. Methods for Realistic Landscape Imaging. Thesis. Yale University; 1993. pp. 1–276. Available at: http://www.kenmusgrave.com/dissertation.pdf

5. Ebert DS, Musgrave FK, Peachey D, Perlin K, Worley S, Mark WR, et al. Texturing and Modeling. Texturing and Modeling. Elsevier; 2003. 9 p. Available at: DOI:10.1016/B978-155860848-1/50050-4

6. Musgrave FK, Kolb CE, Mace RS. The synthesis and rendering of eroded fractal terrains. ACM SIGGRAPH Computer Graphics. New York, New York, USA: ACM Press; 1989; 23(3): 41–50. Available at: DOI:10.1145/74334.74337

7. Mandelbrot BB. Fractal landscapes without creases and with rivers. The science of fractal images. 1988. pp. 243–260. Available at: http://dl.acm.org/citation.cfm?id=61160 http://portal.acm.org/citation.cfm?id=61160

8. Fournier A, Fussell D, Carpenter L. Computer rendering of stochastic models. Communications of the ACM. ACM; June 1982; 25(6): 371–384. Available at: DOI:10.1145/358523.358553

9. Lewis JP. Generalized stochastic subdivision. ACM Transactions on Graphics. 1987; 6(3): 167–190. Available at: DOI:10.1145/35068.35069

10. Miller GSP. The definition and rendering of terrain maps. ACM SIGGRAPH Computer Graphics. 1986; 20(4): 39–48. Available at: DOI:10.1145/15886.15890

11. Saupe D. Algorithms for random fractals. The Science of Fractal Images. New York, NY: Springer New York; 1988. pp. 71–136. Available at: DOI:10.1007/978-1-4612-3784-6_2

12. Voss RF. Random Fractal Forgeries. Fundamental Algorithms for Computer Graphics. Berlin, Heidelberg: Springer Berlin Heidelberg; 1985. pp. 805–835. Available at: DOI:10.1007/978-3-642-84574-1_34

13. Perlin K. An image synthesizer. ACM SIGGRAPH Computer Graphics. 1985; 19(3): 287–296. Available at: DOI:10.1145/325165.325247

14. Perlin K. Improving noise. ACM Transactions on Graphics. New York, New York, USA: ACM Press; 2002; 21(3): 2–3. Available at: DOI:10.1145/566654.566636

15. Spencer K. OpenSimplexNoise.java. 2015. Available at: https://gist.github.com/KdotJPG/b1270127455a94ac5d19

16. Musgrave FK. 2 Procedural Fractal Terrains. Texturing and Modelling. A Procedural approach. 1994; 2.

17. Evertsz CJG, Mandelbrot BB. Multifractal measures. Chaos and Fractals. 1992. pp. 921–953. Available at: http://lipenreferences.google-code.com/svn/trunk/Papers/Multifractals/1992Multifractal Measures.pdf

18. Google. Material design. Google design guidelines. 2016. Available at: https://material.google.com

19. Lloyd DB, Boyd C, Govindaraju N. Fast Computation of General Fourier Transforms on GPUs. Microsoft Research; April 2008 p. 7. Available

at: https://www.microsoft.com/en-us/research/publication/fast-computation-of-general-fourier-transforms-on-gpus/

20. Lyons RG. Understanding digital signal processing. 2nd edn. Upper Saddle River, New Jersey: Prentice Hall; 2004.

APPENDICES

Appendix A User Manual

- Explain each UI panel in detail
- Insert activity diagram showing the main scenario
- Possibly one diagram per scenario (See User Stories in Table 4-1)

Appendix B Additional Results

- Tables with benchmark values
- Additional Terrains