University of St Andrews

April 20, 2012

CS4099 - Major Software Project

Final Report

Oleksandr Stasyk

Supervisors: Dr. Ian Miguel, Prof. Ian Gent

Declaration

I declare that the work submitted in this document is all my own work except where the credit has been given. The work was carried out in the academic year unless otherwise stated.

The main text of this document is 10779 words long.

In submitting this document to the University of St Andrews I give permission for it to be viewed in accordance to the University Library regulations. I also give permission for the title and abstract to be published and supplied at any cost to any bona fide library or research worker, and to be made available on the World Wide Web.

I retain the copyright in this work.

Oleksandr Stasyk: _____

Abstract

The terrain used in video games is difficult to completely generate procedurally. This results in limitation of immersing experience when re-playing the game. This project aims to allow for quality terrain generation and creation of a different random worlds. The project has lead to the discovery that the terrain generator can be used beyond video games, while also providing realistic and natural looking output.

Contents

Chapter 1

Introduction

1.1 The Problem

Role playing video games is a popular computer game genre, having the player controlling a single character or party of multiple adventurers. The game takes place in a world where the character or party must complete quests to eventually complete the final goal. As the player progresses, the quests become more challenging, presenting the player with more powerful enemies to battle using various combat systems (variations being turn-based/real-time, taking place in the game world or have the player teleported to a battle field). To balance the difficulty, the player's character or party will level up, gain more powerful abilities and items as rewards for kills or completing quests. By far one of the most important factors in making a successful role playing game is its setting. Hence the environment that surrounds the player is key to the experience.

Throughout history, most role playing video games handcrafted, relying on a set world created by a level designer. Some earlier, text based, games used procedurally generated worlds and scenarios. This provided for feel of never having the same adventure twice. However once more graphically advanced games appeared, it became more challenging to design and implement such worlds. This resulted in game setting variety taking a lower priority [1].

The foundation of the setting is the physical world the player exists in. A common approach to creating the terrain, for the player to traverse, is the use of fractal algorithms. This project will go beyond basic fractal generation to add extra features, such as refining the quality of the produced terrain as well as addition of rivers, forests and settlements.

Furthermore, the developed generator will be used in two ways: to have the initial terrain generated be changed dynamically overtime (e.g. due to erosion) or to have the generator creating the terrain concurrently while the game is running. The latter would be used for games that do not require great detail and regard the terrain as a backdrop.

1.2 Project Baseline

The project was implemented from scratch and the nature of the software to be created provides for a lack of any basis to start from. The tools and algorithms are to be written by myself apart from only the most basic functionality provided by the development libraries, such as parsing, and pixel based rendering.

1.3 Project Objectives

The aims of the project include creating an interface for gaming software, which should be utilized as middleware for the user software. The generator, will be able to handle a query, to create terrain according to specified attributes, and in turn should output said terrain represented in a format convenient for the user

software. Additionally, the generator will create overlaying features, such as forests, vegetation, rivers and settlements. These features will be stored as decoupled data from the rest of the terrain to give the user software freedom to ignore them if they are not required. The generator should also provide 'on the fly' terrain generation, for example after the user software has requested some initial terrain, while still running, additional terrain data might be requested which should seamlessly merge with the previously created terrain.

1.4 The Requirements

The requirements outlined above are refined into a much more detailed list, logically split into groups by their significance to the overall success of the project.

1.4.1 Primary Requirements

Primary requirements are those which are important and core to the success of the project.

1. The generator should be able to output a two dimensional matrix of values. The matrix should be generated using a fractal approach, such as the square-diamond algorithm.[2]
2. The generator should take the matrix size properties as an input from the user.
3. To add a layer of realism to the resulting output height map and erosion algorithm should be applied.
4. To simulate further realism, in this case, formation of gorges and specific peaks and an impression of tectonic plate presence, a Voronoi diagram can be generated and interpolated together with the existing matrix.
5. The generator should be able to output the matrix into a file representing the data in a defined, standard format.
6. The generator should output the data into the file in another, non-standard format, such as, for example a representation using XML.
7. The generator should determine a set of points to be river sources according to specified constraints.
8. The generator should determine a set of points to be the initial vegetation seeds according to specified constraints.

1.4.2 Secondary Requirements

Secondary requirements are those which are important, but are not core to the success of the project. They may be formal requirements from the users or self-imposed requirements.

1. To aid the development and testing a tool should be developed to allow for rendering of the terrain.
2. A concept of scale should be introduced, regarding the amount of information represented by the single element of the height matrix.
3. The generator should allow for a specific manipulation of the properties of the terrain, i.e.:
   (a) The seeded height values and the maximum difference between any two elements should be used as input for the generator.
   (b) A value should be provided as input to determine the 'roughness' of the terrain. For example low values would create a smoother looking terrain, while higher values would result in an increased number of peaks.

Note: These values have been chosen due the specific nature of the algorithm used.

4. A configuration file should be made available for the user to specify the input parameters.
5. A set of points should be determined to represent the central locations of settlement according to specified constraints.
6. An algorithm should be implemented to iterate the produced set of river sources and extend into full rivers. These should be represented using a standard XML format and output in a file.
7. An algorithm should be developed to iterate over the produced set of vegetation seed points and grown them into vegetation groups. The user should be able to specify the number of generations available for the algorithm to develop the groups. The resulting data should be represented using a standardised XML format and stored in a file.
8. An algorithm should be implemented to iterate the produced set of settlement points and procedurally generate the containing buildings. The as a whole, the buildings should be arranged in a realistic pattern. Alike the previous requirements, the data should be output and stored in an XML file.
9. The generator should be able to apply further erosion effects on the terrain data, after the initial height matrix has been generated.

1.4.3 Tertiary Requirements

Tertiary requirements are those which are neither core to the success of the project, nor particularly important as features in the final deliverable. Just one of these would however be an asset to the project and provide distinguished quality.

1. The generator should be able to create adjacent terrain with respect to already existing terrain. The border between the two data sets should be merged to the point of seamless terrain transition.
2. Concurrent generation of the terrain should be able to provide real time performance of the generator while creating realistic terrain.
3. With the aim of further developing the project, the generator can be put in context of computer role playing games by overlaying the terrain with objects the player can interact with. For example, the player should be able to cut down a tree resulting in updating the vegetation state. Furthermore, a quest system could be developed relating to the terrain and the interactive objects present.

1.5 Tools Used

For the implementation of the project the Eclipse C Development Toolkit IDE was used together with VIM text editor. Git version control software was used to manage the source repository. The majority of the project is implemented in C++ along with Python tools and helper Bash scripts. The various development libraries used are described in the Design and Implementation chapter, section 2.1.2.

Chapter 2

Design and Implementation

2.1 Generator Software

2.1.1 Choice of programming language

The language chosen for the implementation of the core generator is C++. The primary reason for this was performance. This is related to one of the specified requirements being the ability of the software to produce terrain in real time (see section 6.1.3). The compiled executables should be able to provide a superior execution time compared to an interpreted script. The original choice for implementation was C, however not very far into the development stage it became clear that more sophisticated data structures were essential for the algorithm implementations. One such example is the `std::vector` data structure's ability to hold a dynamically sized list of elements. This choice also presented the convenience of Object Oriented approach to particular problems. The details of these solutions will be discussed later in this chapter.

2.1.2 Development libraries used

The primary part of the functionality has been implemented by myself, however there are two helper libraries that have been used.

getopt.h This provides functionality for easily parsing the command line arguments. Since the generator software relies heavily on user input, a lot of input values can be passed using the command line arguments.

libyaml.h Similarly this library is also related to input handling. However this particular library provides functionality to parse and extract data from `.yaml` files. Such is the type of the configuration file the generator uses to receive input from the user.

Note: Every other library/function used in the project is part of the standard C/C++ library.

2.1.3 General Architecture

As a whole the generator software takes on a simplistic architecture when viewed from an abstracted level. The user provides an input in a form of a configuration file or command line arguments. If input is not provided, the generator uses inbuilt default values. The software then runs a set of algorithms and produces an output. The output takes a form of the terrain map file along with data files representing the generated rivers, settlements, vegetation, etc..

2.1.3.1 Source modules

The generator is split into a set of source modules, each representing a major part of functionality. All of these modules include the same header file and share values throughout the software.

terrain_generator.cpp This module is the main wrapper for the software, containing the main method. The module requires both of the libraries mentioned in section 2.1.2 due to being the first to encounter user input, with respect to the data flow throughout the software. Having implemented the main method, the module is required to deal with the command line arguments. The main method is used to set up the known arguments and create an option struct for the use of the option command line argument parsing library. However the method requires to parse and set values from the configuration file next, since the argument values should override the configuration values. Hence the method utilises the yaml file parser to set values from the configuration file prior to reading the values of the command line arguments. And hence after this two stage set up of values, the main method calls the generate procedure. Which in its turn makes use of other source modules to produce an output.

midpoint_displacement.cpp This module is the backbone of the generator. It implements the square-diamond algorithm which is the basis for the generated terrain. For a detailed description of the algorithm see section 2.1.4.1. The module also implements algorithms for creating and interpolating vornoi diagrams as well as eroding the resultant terrain. The constraint checking methods related to the altitude of the terrain are also implemented here. And finally the module contains output methods, allowing for outputting the generated matrix in different format given a stream. The stream type given is usually an open file to be written to. The functions in this module have to be executed prior to the functions contained in other modules due to heavy dependence upon the generated data of the algorithms contained in the latter functions.

rivers.cpp This module is responsible for providing the functionality to create and develop rivers. It relies on the data generated by the `midpoint_displacement` module. The majority of the module is the implementation of the seeding of river source points along with the river growth algorithm. The module also provides output functionality by allowing to write the river representations to a file.

settlements.cpp Contained within this module is primarily an algorithm for placing points to represent centre location of settlements on the map. The choice is based on the pre-generated terrain data. The module also allows for outputting the created features in a file.

vegetation.cpp This module is responsible for seeding the initial vegetation points. It also implements algorithms to grow groups of forests using those seeds as starting points. The algorithms in use are also dependant on the terrain data produced. Finally, alike the above modules, this module is able to output the generated settlement data in a file.

contour_format.cpp This module focuses on providing functionality to convert the terrain data into a contour like format. This included multiple algorithms for iterating the terrain matrix and transforming it into a specific contour format. The functionality in the module, however, has been disabled due to only partial implementation. Alike the other modules, this also has the functionality to output the produced data to a file. The output formats included a visual representation using ASCII characters and an XML file to be read by a game engine. The main purpose of the module was the first step into allowing for integration with user software. The module could be the output interface for a game that could specify the format of map data it uses. A dictionary of format mapping could be stored in the module to accommodate the game's requirements.

2.1.3.2 Software input

The generator allows for input via command line arguments as well as using a configuration file. The command line arguments allow for:

- A help prompt display
- A filename of a custom configuration files to be used
- A verbose flag for generator's standard output to be enabled
- A height and width integers to be used
- A 'roughness' value used for the square-diamond algorithm, determining the shape of the terrain, see section 2.1.4.1
- A seed value used for the square-diamond algorithm, determining the average height of the terrain, see section 2.1.4.1
- An offset value to be used for the square-diamond algorithm, determining the shape of the terrain, see section 2.1.4.1
- A plate value, used as the interpolation fraction of the Voronoi diagram, see section 2.1.4.2
- Erosion value, used for the number of erosion iterations, see section 2.1.4.3
- A negative values flag, allowing for negative values in the terrain map
- A standard flag, forcing the standard height map format in the output file
- An XML flag, forcing the XML height map format in the output file

The configuration file allows for the following input:

- Specified output format, either standard or XML
- Output filename for the terrain height map
- Height and width values
- Scale value, allowing for different interpretations of the terrain
- A seed value used as in the command line arguments
- An offset value used as in the command line arguments
- A 'roughness' value used as in the command line arguments
- A normalise flag, specifying whether the output should be normalised
- A normalisation minimum boundary
- A normalisation maximum boundary
- An integer to represent the sea level
- An integer to represent the sand level
- An integer to represent the snow-top level
- An integer to represent a minimum height difference for defining cliff features

- An integer amount of rivers sources to be created, see section 2.1.4.4
- An integer amount of maximum allowed branches per river, see section 2.1.4.4
- An integer amount of settlements to be generated, see section 2.1.4.5
- A value of minimum separation between the settlements, see section 2.1.4.5
- An integer amount of the initial vegetation seed objects to be placed, see section 2.1.4.6
- An integer value for the radius of the vegetation roots, see section 2.1.4.6
- The number of generations to be ran to develop the vegetation population, see section 2.1.4.6

2.1.3.3 Software output

The generator provides output in a form of files containing data expressed in multiple formats. The generated files include:

- A file containing the terrain height map.
- A file containing the generated river data.
- A file containing the generated settlement data.
- A file containing the generated vegetation data.
- A file containing an alternative contour based representation of the terrain.

There are two available formats of the terrain height map file. The first is the standard format, which consists of a series of integers. The first two integers represent width and height respectively, while the rest of the integers are the contents of the height map, with all the rows appended to the same line.

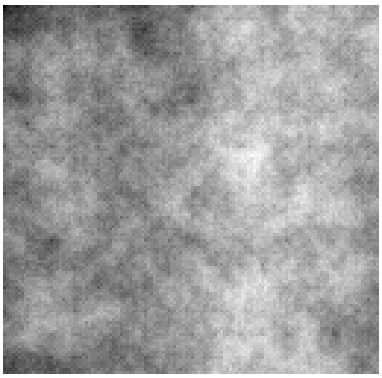
The second format is the height map representation using XML. This allows for a more informative output. Alike the standard format, the XML output includes the size of the height map, however every tile of the map has additional information such as tile coordinates and the tile type. The implementation allows for five different tiles types:

- Water tile, a tile below the specified sea level
- Sand tile, a tile below the specified sand level
- Grass tile, a tile between the sand level and snow-top level
- Cliff tile, a tile which has a pair of neighbours whose value difference is above a specified value. This tile overrides any global height constraints, meaning a cliff tile can be located any where above sea level, regardless of the actual tile's own height value.

2.1.4 Algorithm Approaches

This section describes the algorithms used to generate the terrain. To display the output in a meaningful way, a developed visualisation tool was used described in section 2.2.3.

Figure 2.1: A height map produced by a basic square-diamond algorithm. The two definitional output matrix is mapped onto a canvas by rendering each pixel according to a value of an element in the matrix. Brighter pixels correspond to higher values.



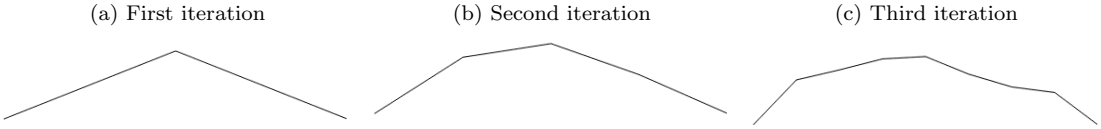
2.1.4.1 Square-diamond

This algorithm is one of the most common methods to create a height map. When visualised, the resultant image can be described as a fractal, since, when the generated shapes seem to replicate each other.

The concept of midpoint displacement technique, which is the basis of the algorithm, is best introduced in one dimension.

```
Given a straight line, find the middle point on said line
Offset it by a random amount bounded by a range R while keeping the end points in place
  This produces two lines joined at the offset midpoint
Reduce range R
Repeat the above, for each new line, for a set number of iterations
```

Figure 2.2: Visual representation of the midpoint displacement algorithm using a line.[2]



From Figure **??** it is possible to deduce that every iteration of the algorithm adds detail to the overall picture, and even by the third iteration the image is starting to look like a mountain or a hill silhouette. Given enough iterations a detailed image of a mountain range can be produced.

However to create the equivalent in two dimensions a more advanced approach is required. Since the one dimensional approach can only fill a single row of a matrix, simply applying the same algorithm across all the columns will not give a cohesive terrain. The adjacent columns will not have any information taken into account about their neighbours. Hence the output terrain will have unnatural stripes due to the height differences between the elements in adjacent columns. To produce a cohesive matrix of values, the two dimensions need to be addressed at the same time. Nevertheless the same recursive divide and conquer approach can still be applied.

The main idea of the approach is to select the outermost four values in a square formation, calculate their average, add a random offset and assign it to the element in the centre of the four elements. The size of the square is then divided by 2, meaning the distance between the elements belonging to the square is

halved. The resultant new element is set a value using the same function as before. The square of elements is then shifted along the matrix by the its size and the new centre values is applied again. This is repeated until the square has been shifter to across its possible rows and columns, each time moving by its size, so when the square is halved for the first time there are four possible positions. Then the size of the square is halved and the above procedure is repeated until the size of the square is 1.

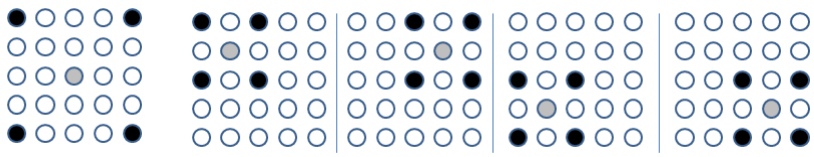
Listing 2.1: Pseudocode describing the algorithm

```
Given a size S [S having to be a (power of 2) + 1],
  and an offset range R
Create a 2D matrix of size S
Set the corner elements of the matrix to desired or random seed values

let s = S
while s  > 1

  middle = s/2
  i = m

  while i < S -1
    j = m
    while j < S -1
      element[i][j] = average + random values within R
      j += s
    i += s
    loop
  loop
  reduce R
  s = s/2
loop
```

Figure 2.3: Illustrated is the procedure to assign values to the matrix, assuming the corner seed values are already present. The black circles show the four elements used for the average, while the grey circle is the element to which a value is being assigned.

(a) First iteration, s=S

(b) Second iteration, s=S/2



The resultant matrix produces a fractal height map. However due to the fact that only the surrounding square is taken into consideration, some values required for the calculation of the centre element are still unassigned a value. This creates an effect of the terrain consisting of superimposed squares. To balance this, an augmentation to the algorithm is required. After assigning a value using a square formation of elements, a diamond formation can be used to fill in the missing values as well as remove the visual effect of superimposed squares. And hence now the algorithm includes a 'square' and 'diamond' step.

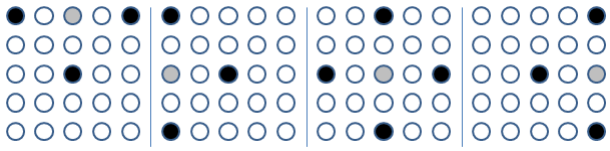
Listing 2.2: Pseudocode describing the algorithm with added diamond step, the same algorithm used to generate the height map in Figure 2.1

```
Given a size S [S having to be a (power of 2) + 1],
  and an offset range R
Create a 2D matrix of size S
```

```
Set the corner elements of the matrix to desired or random seed values

let s = S
while s  > 1

  middle = s/2
  i = m
  #square step
  while i < S -1
    j = m
    while j < S -1
      element[i][j] = square average + random values within R
      j += s
    i += s
    loop
  loop

  #diamond step
  odd = false;
  i = 0;
  while i <= S - 1
    if !odd
      j = m
    else
      j = 0
    while j <= S - 1
      element[i][j] = diamond average + random values within R
      j +=s;
    loop
    i += m;
    flip odd
  loop
  reduce R
  s = s/2
loop
```

Figure 2.4: Illustrated is the procedure to assign values to the matrix, during the diamond step phase (only the first iteration is shown). As previously, the black circles show the four or three elements used for the average, while the grey circle is the element to which a value is being assigned.

(a) First iteration, s=S



Close attention must be paid to the reduction of the offset range R mentioned in Figures **??** and **??**. The size of R determines the maximum offset from the average value. If R was set to 0, essentially setting the element's value to the average, the produced terrain would be completely flat, sloping smoothly between the seeded values in the corners. As R is set to a high value, the terrain is filled with random, extreme, sharp peaks and trenches. So, as the algorithm fills the values there is a required decrease in R, resulting in a mixed variation between the offset values. If the R is reduced slowly (default reduction value used in implementation is $R = R \times 0.8$), the terrain has a moderate set of peak features while still remaining smooth, over giving them most natural looking output. Of course the reduction of R can be modified to produce features of more rough cliff based terrain as well as more eroded, desert-like features, if desired.[2]

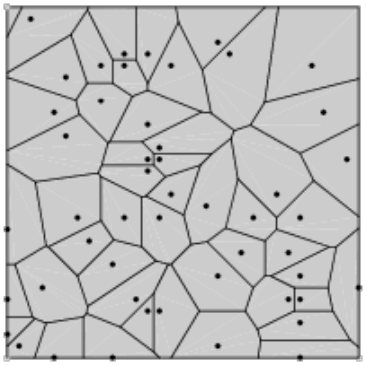
Overall the output of the algorithm provides a very natural looking basis for terrain generation. The

resultant matrix however is further manipulated to enhance the realism of the output.

2.1.4.2 Vornoi diagrams

Voronoi diagrams can be represented by a set of points. These points are hence used to decompose the space they are located. This is very useful in terms of graphical representation and is used widely in science and engineering. In the context of this project, these diagrams can represent various features depending on interpretation as well as separate the height map into areas. With respect to the scope of the project, a Voronoi diagram is used for interpolation with the height map produced by the square-diamond algorithm to create crevasses and mountain ranges. The overall effect gives a visual impression of land separation due to tectonic plate movements [3].

Figure 2.5: A representation of a Voronoi diagram, the dots represent the points, while the lines show separation of the areas defined by those points. [**?**]



The algorithm use to generate the diagram is implemented in two stages. The first stage involves simply filling a set of points with random values. The second phase actually alters the height map values in the actual matrix, meaning no external data structures are not used to save the generator's memory usage. The representation of the diagram in this context is as follows:

- The diagram is represented as a set of points
- Each element takes on an integer value
- The value of a given element depends on its distance to the closes points
- These values are the used for interpolation with the height map

Once the basic height map is created, the generator proceeds to evaluate the interpolate Voronoi values. This includes iterating over the given matrix, for each value two point identifiers are defined: one represents the closes Voronoi point, the other represents second closest. These are then calculated by iterating through all the points, using euclidean distance to the element position and storing the two lowest distances. Then, the value is assigned to the element using the formula $value = D_2 - D_1$, where D_2 and D_1 are the second closest and closest distances receptively. The reason for the specific value calculation is the resulting shape of the diagram.

Listing 2.3: Pseudocode for interpolating the vornoi values with the produced height map

```
give an value A, an interpolation ratio
```

```
set d1 = size of map
set d2 = size of map

for each element e in the height map
  for each Voronoi point p in the Voronoi set
    let D = distance between e and p
    if D < d2
      d2 = D
    if D < d1
      d2 = d1
      d1 = D
  loop
  val = d2 - d1

  set value of e = (1-A)*value of e + A*val

  reset d1 and d2
loop
```

Figure 2.6: A visualisation of the generated diagrams, high values are represented by light pixels and vice versa.

(a) With interpolation ratio set to 1 (b) With interpolation ratio set to 0.33

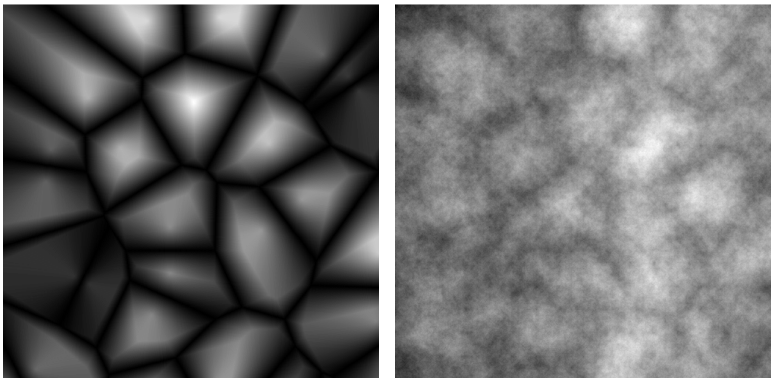


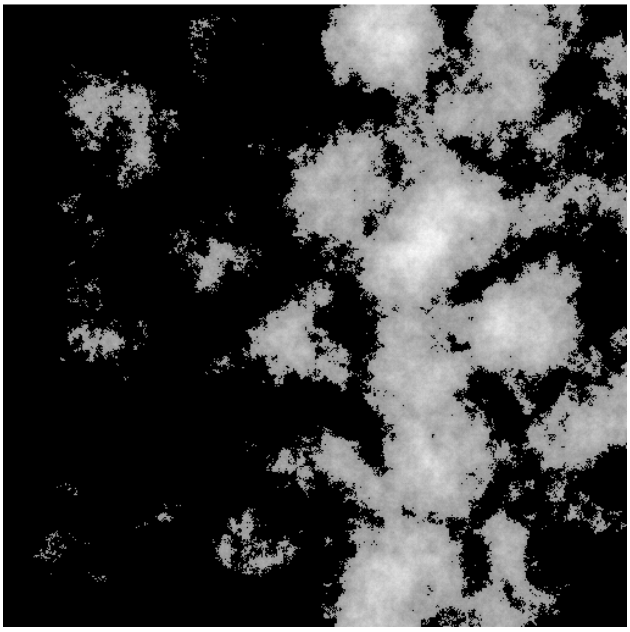
Figure 2.6 shows the produced height maps with varying interpolation value. The implementation uses a value of 0.33 because it has been found to give the best visual results in terms of producing realistic terrain. Tools described in section 2.2.3 were used to determine the optimal value.

2.1.4.3 Thermal erosion

The next stage of terrain creation deals with a further level of realism. Visualisation tools, section **??**, provide a crucial insight into the texture of the output terrain. If a threshold cut off is used to visualise the height maps from 2.6, for example to imitate water level in the terrain, it can be seen that the produced terrain has a subtle flaw, see Figure 2.7. The primary source of the problem is the use of the diamond-square algorithm itself. The 'roughness' property, alternatively how smooth the terrain is, can be change by editing the reduction of the offset range, however, due to the noise-like nature of the data produced, the terrain will always have scattered values. These scattered points are highlighted when a visual threshold applied, for example, only elements containing a value above a threshold value are rendered. This hinders the natural look of the terrain and requires an erosion algorithm is required to smooth these values.

Thermal erosion happens due to material from higher ground falling to lower. Hence an approach for an algorithm would follow the same idea. The higher value elements would share out a fraction of their

Figure 2.7: A height map rendered with a cut off threshold to show the scattering of individual pixels, making the terrain look more like a cloud.



values to surrounding elements. Hence all of the elements in the terrain matrix need to be traversed, for each element, all eight neighbouring elements are traversed to find the lowest neighbour. A design decision has been made that the current element will transfer a fraction of its value to its lowest neighbour, given that the neighbours value is lower than that of the current element. This of course can happen more than once, in turn producing a more eroded terrain.

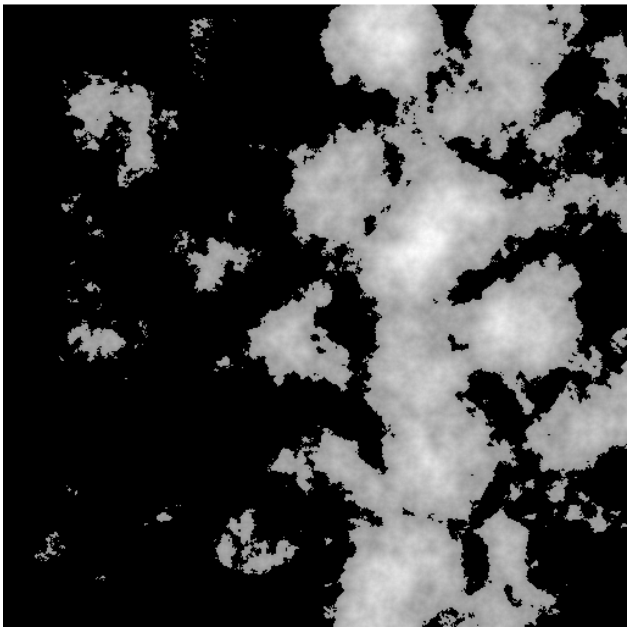
Listing 2.4: Pseudocode depicting the thermal erosion algorithm

```
for number of erosion iteration
  for each element e in the map
    populate set N with neighbouring elements
    traverse the set N
    find minimum element value min_v of element min_e

    calculate a fraction depending on value of e and minv to be moved m
    e value -= m
    min_e value += m
  loop
loop
```

Figure 2.8 shows the same height map as figure 2.7, however after it has been eroded using the algorithm featured in Figure 2.4. There is a visual difference between the two outputs. The pre-eroded terrain clearly shows scattered single points giving a very fuzzy appearance, while in comparison the eroded terrain his been visibly smoothed. This smoothness property is essential for generating rivers later.

As far as the implementation is concerned, this concludes the terrain foundation. The resulting terrain map is therefore stored in a file to be used by user software.

Figure 2.8: Terrain height map after being eroded.



2.1.4.4 River generation

In the context of the generator, rivers are defined as having a source points, and a list of fresh water tiles related to that source. The rivers are considered to have their source at high ground and either flow into the main sea waters, flow into other rivers or form lakes. The generation of rivers consists of two phases. The first includes creating the river sources, while the second deals with growing the rivers starting at their sources.

To create a set of river source points source the generator first make a set of viable candidate points according to set constraints. A design decision has been made to constrain the river sources to only be located above a certain height, for example in the mountain peaks. The generator then chooses and removes candidate locations at random from the candidate set. Each chosen candidate is then set as a river source point.

The growing phase involves iterating the source points and running the growing algorithm. The growing algorithm, in turn, involves placing a water tile element on the river source. Then all eight neighbours of this river tile are traversed to find the one with the lowest value. That is then chosen as the next water tile. The grown algorithm is then recursively applied to the next tile. A few conditions apply to choosing a neighbouring tile.

- If the lowest neighbour is part of the same river, the next lowest block is chosen.
- If there is a neighbour which is a part of a different river, the grow algorithm is stopped.
- If the lowest neighbour value is below or equal to the specified sea level, the grow algorithm is stopped.
- If there are no suitable neighbours, the algorithm is stopped.

Listing 2.5: Pseudocode depicting generation of rivers

```
given total number of rivers Rn
```

```
#candidate selection stage
for each element e in the map
  if e fits candidate constraints
    append e to candidate set C
loop

#source selection stage
for i ranging 0 to Rn
  pop random element from C
  append to river source set S with id i
loop

#growing stage
for each element s in S
  grow using river source s
```

Listing 2.6: Pseudocode depicting the grow algorithm

```
Given a river point R

create a set of available neighbours N

for each element n in N
  if R is located at a water tile with the same river id
    skip current element
  if R is located at a water tile with a different river id
    force procedure return

  find lowest neighbour n

if n has not been set
  force procedure return

create a new water tile T element using n
call grow procedure using T
```

The generator also implements river branching. This can be triggered by a number of factors. One of the conditions is if neighbour elements have the same value, mimicking the fiver creating a fork. Another condition is dependant on the inverse of the distance from sea level. Meaning the closer the river tile is to sea level, the more likely it is to branch out. This is aimed to imitate river deltas, increased branching of the river as it approaches the main body of water. The branching itself consists of choosing a second neighbour and creating a new river tile and running the grow algorithm on it.

Figure 2.9 shows the terrain with included rivers, created using the algorithm depicted in figures 2.5 and 2.7. The motivation behind the rivers came from the observation of water flowing downhill due to gravity. Hence this suggested that the use of inverse hill climbing should be the basis. This however created the problem of local minima. Yet again a real life example was used for inspiration, once a flowing body of water reaches a blow without any clear path to flow downwards, providing a constant flow, the bowl would be overfilled and new paths of flow would be exposed. This is mimicked in the algorithm by ignoring the height difference between the current element and its neighbours. Also the rivers approach of 'turning away' from its own waters, creates and illusion of the water spreading out. This in turn produces major lake-like features as clearly seen in figure 2.9.

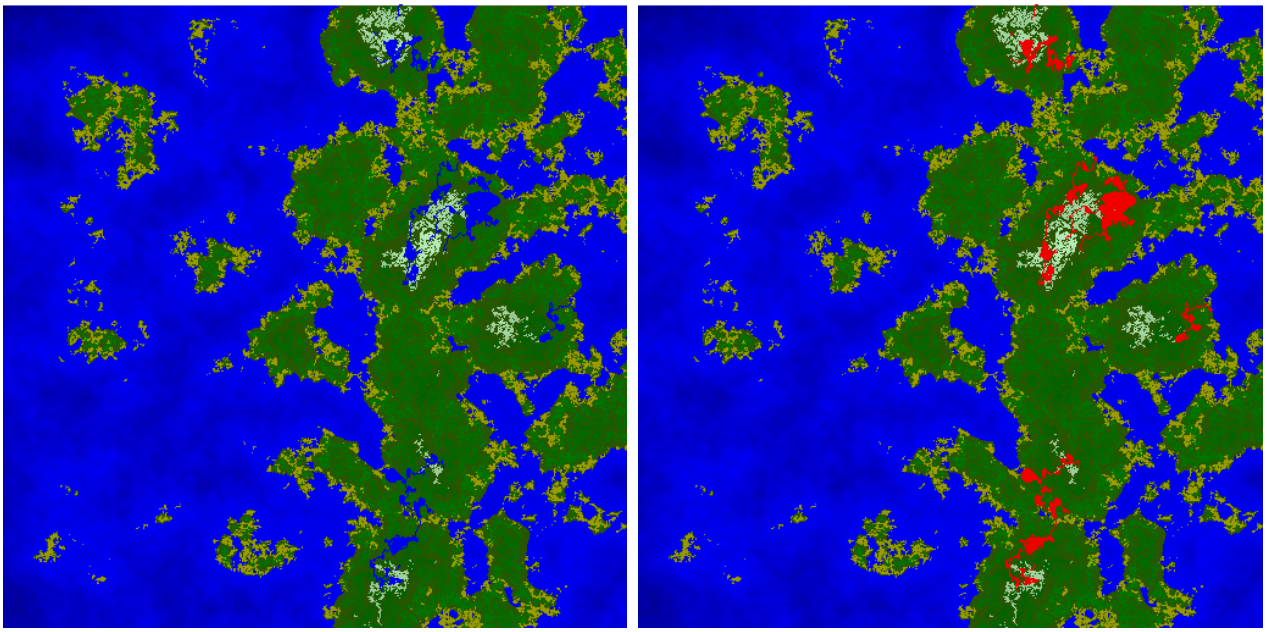
2.1.4.5 Settlement generation

The generator is able to output a simplistic representation of settlements that may be present on the generated height map. In the context of this implementation the settlements are represented by a name and a general location.

Figure 2.9: A visualisation of the generated rivers, using full coloured map.

(a) Default rendering

(b) Rivers highlighted in red



The location of the settlements are chosen according to a set of simple constraints. Alike the river source generation, a set of candidate is used as well as the procedure of randomly selecting and removing from this set. The settlement constraints are as follows:

- A settlement must be located above sea level
- A settlement must be located below snow-top level
- A settlement must be located a specified distance away from all other settlements

The placement algorithm requires two stages of placing appropriate locations. First the candidate set must be created. All of the elements in the map are traversed. If the element value satisfies the former two conditions mentioned above, the element is appended to the candidate set. Once all of the elements are traversed, being either ignored or added to the candidate set, the second stage takes place. A random element is chosen and removed from the candidate set, if the element satisfies the condition of being located far enough from any other settlements, the location is suitable for a settlement.

Listing 2.7: Pseudocode depicting the location selection of settlements

```
for each element e in the map
  if e value > sea level and e value < snow level
    append e to candidate set C
loop

create empty set of settlement locations S
for range 0 to number of settlements
  pop random element e from C
  while e location difference with every other element in S is < allowed minimum
    if C is empty force return
    pop a random element e from C
```

```
    loop
    append e to S
  loop
```

Once a suitable location for a settlement is chosen, it is assigned a generated name. The name generation has been inspired by location names through the United Kingdom. It has been observed that most of the these names are composed of pairs of name elements, holding meaning related to the surrounding features. The identification of said features is outside of the scope of this project and will require implementation of complex algorithms, hence a random naming convention has been chosen.

The naming algorithm is given a prefix and suffix list of naming elements. When a name is required, the algorithm picks a random prefix and appends a random suffix, thus creating a complete name. The resulting output visualisation is portrayed in figure 2.10.

2.1.4.6 Vegetation generation

The core inspiration for vegetation generation is taken from John Conway's Game of Life[5], where elements of a matrix are highlighted according to special rules. The overall emergent behaviour gives an impression of a biological system. When put in continuous animation, the patterns of highlighting elements give impression of moving artifacts, and hence something alive. This applies to real life forests on a very slow scale, assuming elements in the Game of Life are equivalent to trees, it can approximated that real life trees follow similar rules. These rule set allows for object multiplication given certain conditions are true, as well as object remove from the space when other conditions are true. There is also possibilities for objects to remain uncharged. Here objects are equivalent to highlighted cells in Game of Life.

The alike previously, the vegetation generation algorithm uses two phases, one for setting up candidate locations, the next runs the grow vegetation procedure. The location candidates are chosen to satisfy similar constraints to the settlement location constrains. Meaning a vegetation object, such as for example a tree, can be placed above the sea level and below the snow-top level. The overall outcome of the candidate selection phase will create a specified number of vegetation objects on the terrain.

The next, growing, phases involves iterating through every available vegetation point and applying the following rules, to resolve an action to be taken. The neighbour hood of a single vegetation object is determined by a specified root radius, the higher the root radius value, the higher the neighbourhood of the object.

- If the neighbourhood contains no water tiles (either belonging to a river or tiles below sea level) and contains 3 or more other vegetation objects, remove the current vegetation object.
- If the neighbourhood contains 6 or greater, regardless of surrounding water, remove the object.
- In every other case a new vegetation object is created.

The reason for two conditions for removing objects is based on the output terrain to used primarily for games. If the constraints allowed for less object removal, the vegetation population becomes very dense. This may be the natural case, however it greatly hinders the mobility of the character and therefore hinders game-play. The current rules provide for grouping of objects yet mostly avoiding heavy, gap-less, clustering.

Listing 2.8: Pseudocode depicting the vegetation generation algorithm

```
#candidate stage
for each elemente in the map
  if e value is > sea level and < snow-top level
    append e to candidate set C
loop

for range 0 to number of initial seed objects
  pop a random element e from C and set it as a vegetation location
  append e to veg object set V
```

```
#vegetation growing stage
for range 0 to number of generations to be ran
  for each vegetation object v in V
    create neighbourhood set N
    for all elements n within the specified radius
      append n to N
      if n is a water tile
        set water neighbour to true
      if n is another vegetation object
        increment neighbour veg count
    loop

    if water neighbour is false and neighbour veg count is >= 3
      remove object v
    else if
```

Figure 2.11 shows the vegetation clearly showing the desired effects of clustering, while still containing gaps for a player character to traverse through. The vegetation also shows very strong preference for water, evident by large clusters surrounding coastlines.

2.2 External Tools

2.2.1 Choice of programming language

Any tools developed to aid the project were required to be decoupled from the main generator software. Hence to allow for faster development, a scripting languages were chosen. Providing with extensive previous experience Python was the choice for the majority of the tools. Additionally, scripts dealing with automated generation were implemented using Bash.

2.2.2 Development libraries used

The output data format involves XML hence processing will require XML parsing. A common XML parsing python library, called python-lxml[6], was used. Furthermore, once the terrain data was parsed, a graphical rendering library was required to display the data as a rendered map. For this, a python library called PIL[7] was used.

2.2.3 Visualising the data

The visualising tool consist of a series of scripts, each specialising in rendering a specific feature of the terrain, such as rivers, settlement, etc.. Due to being developed alongside the generator and being decoupled the scripts share a lot of functionality that helped further develop the generator. An example would be the concept of rendering different heights with different colours lead to development of generating different tile types within the generator itself. However it also meant that the tool would lack compatibility with the generator in the long term. Thus an external configuration file was needed to synchronise reference points of both the generator software and the visualisation tools.

The main visualising script renders the generated map with all of the generated features. The procedure used first reads the terrain data file and iterates over the containing data. It should be noted the script expects the terrain data to be in the standard format. For every height value a corresponding pixel is coloured depending on the value. The visualisation tool colours tiles below the sea level threshold with varying shades of blue, depending on the height, in this case depth, value. A thin layer above the sea level is coloured yellow to represent sand. Above the sand is a varying green layer showing the grassland plains. Darker green represents more elevated areas. The highest points of the terrain are considered to be completely covered by snow and are rendered white. Through the landmass, dark green pixels are displayed to represent areas of with steep gradients.

Next the script reads and parses the contents of the river data file. Then the river tile elements are coloured blue on the map. The vegetation elements are rendered as four pixels of slightly varying green pixel, aiming to rendered a miniature tree. Finally the script reads and parses the settlement data. The locations are highlighted as black squares and the related name is rendered using a true type font beside the location. The resulting output is shown figure 2.12.

This tool has played an essential part in the development of the project. The tool is able to portray every feature of the generator and hence was used as a primary method of debugging.

Figure 2.10: Terrain with generated settlements.

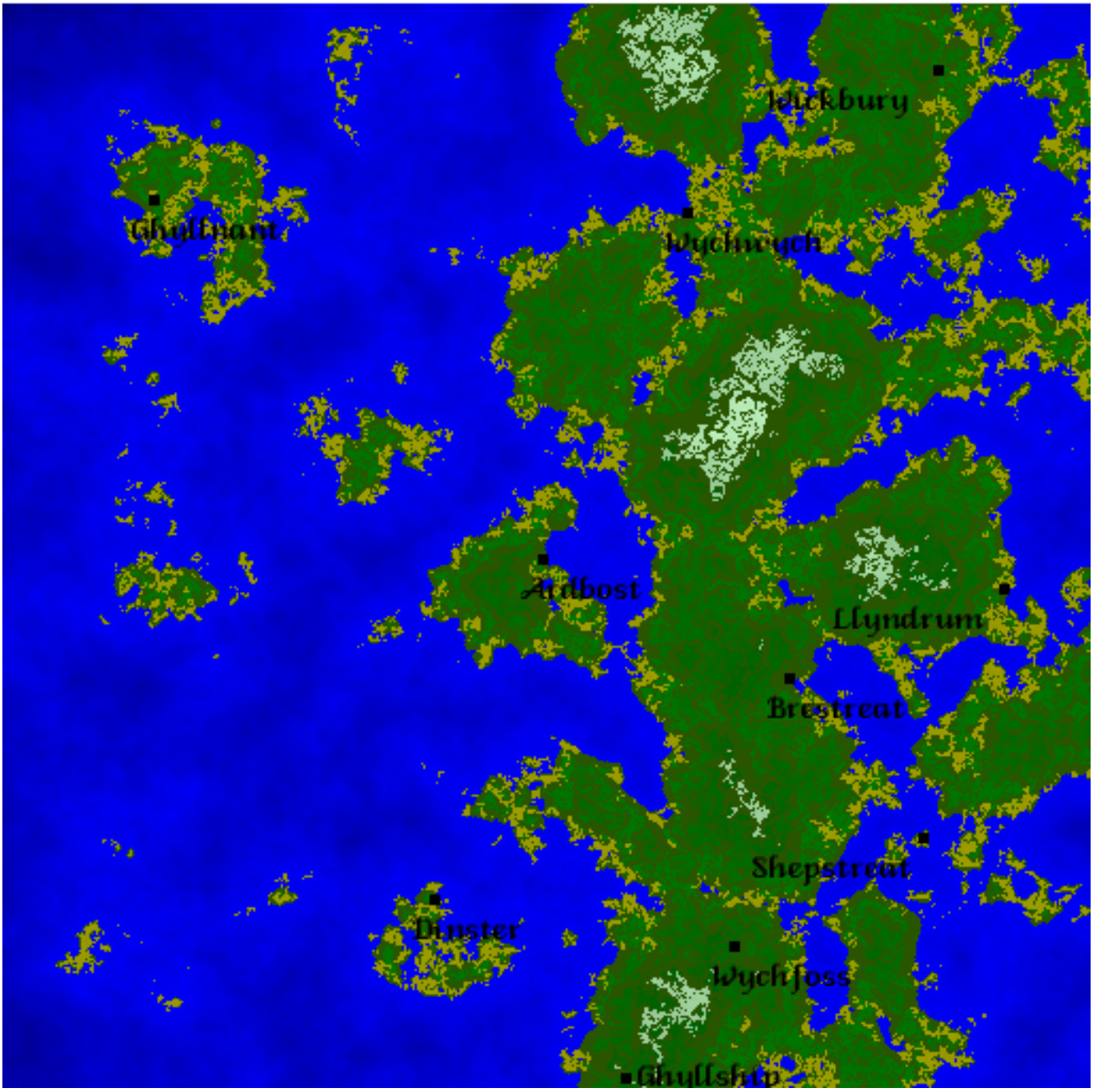
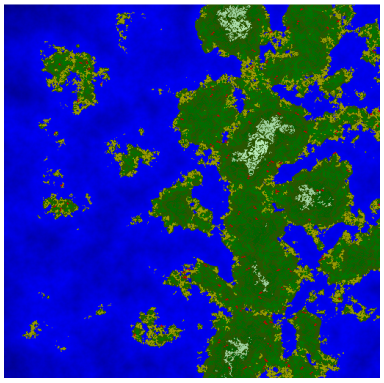
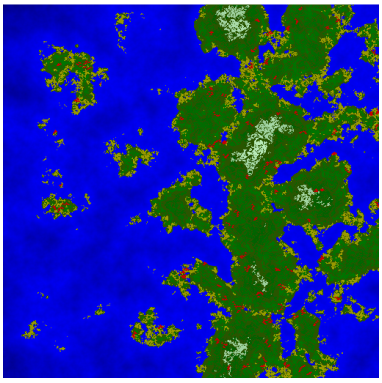


Figure 2.11: A visualisation of the generated vegetation with varying number of generations. The vegetation is highlighted in red

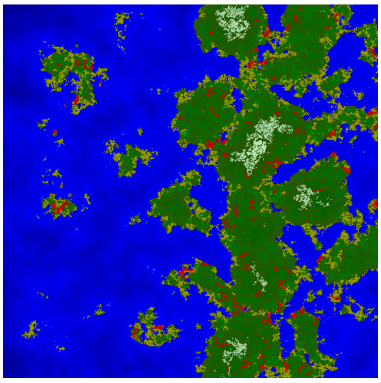
(a) One generation



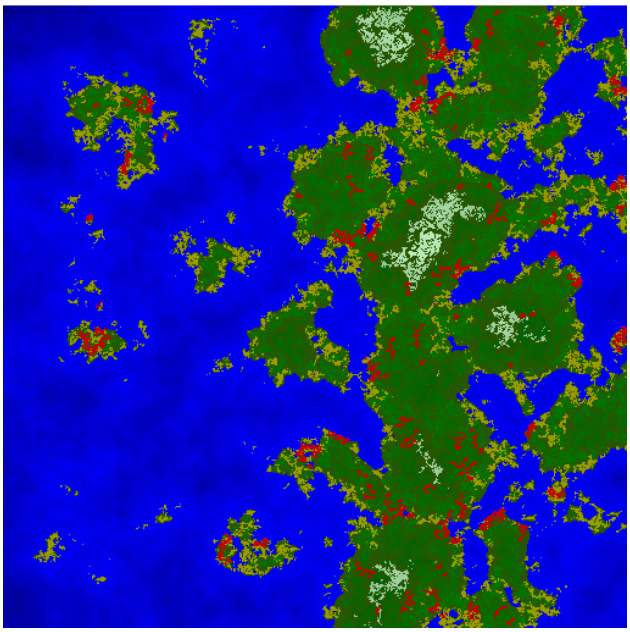
(b) Three generations



(c) Six generations



(d) Ten generations



(e) One hundred generations

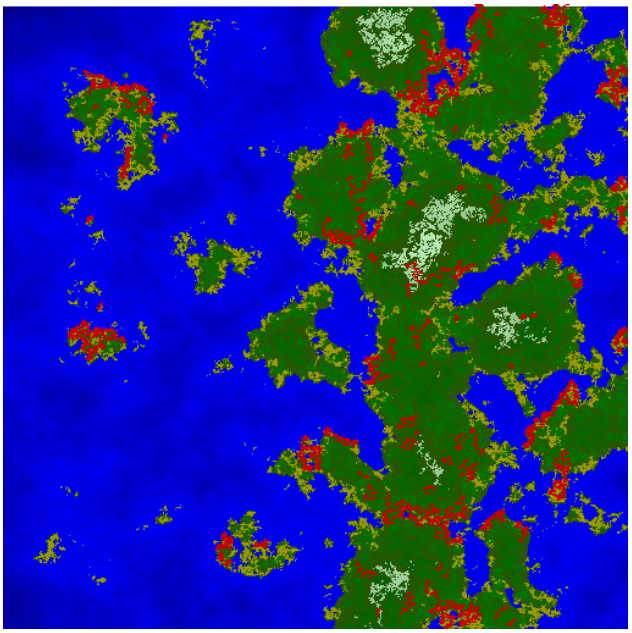
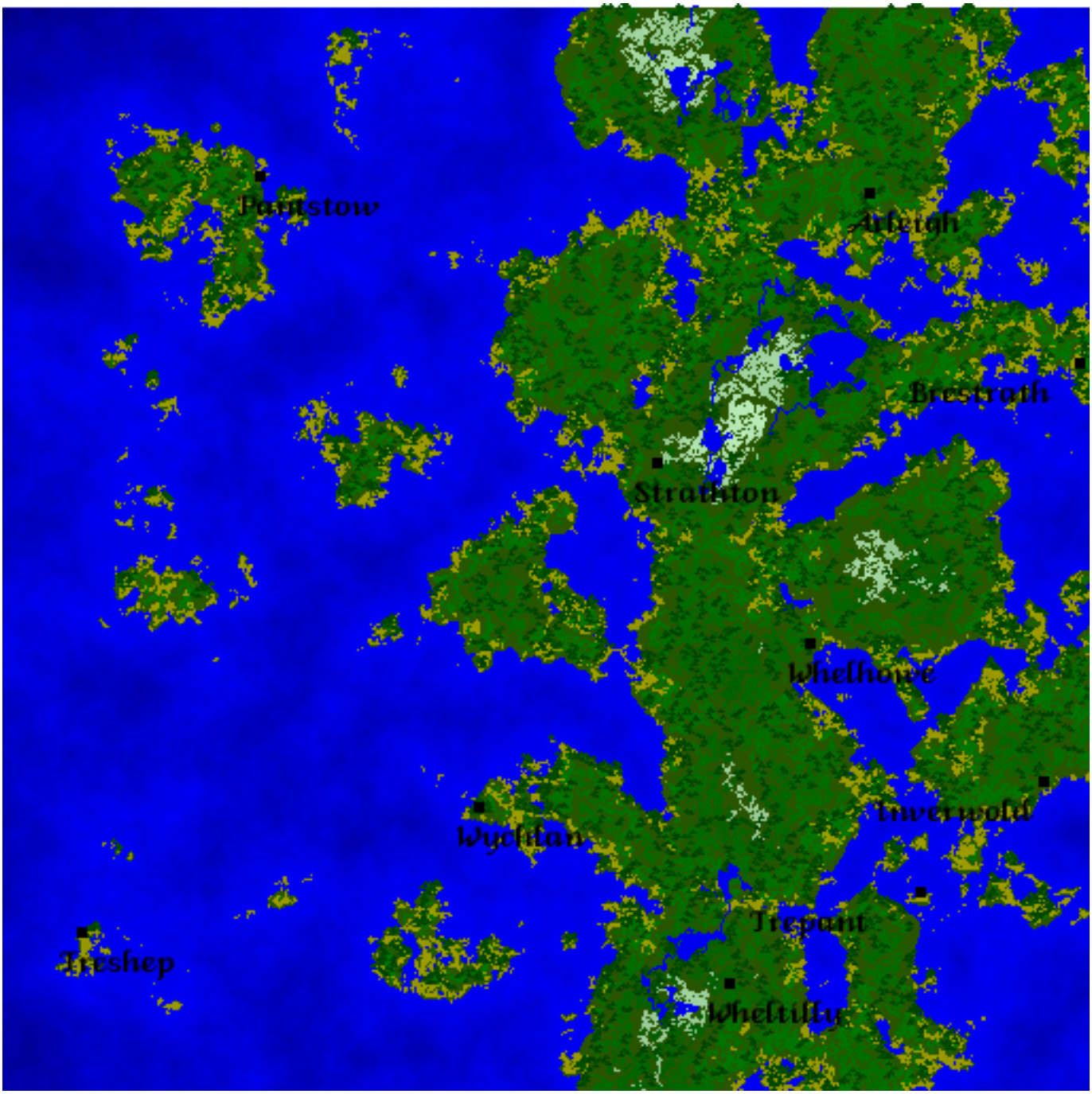


Figure 2.12: Output of the visualisation script, rendering all of the generated features together.



Chapter 3

Testing and Analysis

3.1 Testing

Due to the nature of the software output, the project testing was primarily based around visual feed back. Hence development of visualisation tools was heavily focused on, and less time was spent of developing testing frameworks and unit tests. The development-testing cycle included the following:

- Implementation of an algorithm.
- Make sure the output is valid, this includes checks ranging from, existence of a generated file to the correct ranges of values were produced.
- Implementation of rendering algorithm
- Visual check, the produced terrain makes visual sense and has the desired properties.

3.1.1 Height map testing

To check whether a height map was generated correctly, immediate check could be made by observing the generated height map. If the height map output stored in the file has obvious defects alterations to the algorithm must be made. However if the defects were present in the rendered representation, the problem was most likely present in the input give to the generator.

The visualisation tool also allowed to view the terrain at different stages of erosion and Vornoi interpolation values. Hence it was possible to render different images with successive values of the mentioned values and produce an animated development of the terrain.

3.1.2 River generation testing

Unlike the height map generation, it is very difficult to distinguish correct and faulty output by examining the generated output file. This is due to the output data being represented using XML format. Therefore, the visualisation tool always had to be used for testing and debugging the river creation algorithms. The visual features which required close attention were the presence of river branching, river merging and emergent river meandering.

3.1.3 Settlement location testing

Alike the river generation, the visualisation was relied upon to make sure the location of settlements are assigned to sensible locations.

3.1.4 Vegetation testing

Similar to settlement location, the seed placement could be checked using the visualisation tool. Similarly to the height map testing, different number of maximum generations can be rendered in increasing order, the growth of the forests can also be animated.

3.1.5 User based testing

Further development was aided by close cooperation with user software. This gave major insights on the weak points of the generator as well as ideas for additional functionality, such as meta data files about the available data tiles.

3.2 Analysis

3.2.1 Terrain analysis script

To easily extract statistical information about the generated height map, a script was developed. The script is able to output the following information:

- Minimum height
- Maximum height
- Average height
- Standard deviation of heights
- Average height and standard deviation ratio

This allowed for quick evaluation of the terrain statistics at a glance.

3.2.2 Vegetation growth

After implementing the vegetation growth algorithm, the emergent behaviour showed interesting patterns in the population. Figure 3.1 is illustrating the relation ship between the generation number and the population. It can be seen that the population sizes converge at a point relative to the starting number of seed vegetation objects. This behaviour is expected due to the nature of the algorithm. The initial seeds are the starting points of the developing clusters of forests. The population levels off because of the vegetation removal rules that are applied, which prevent complete exponential growth of the population. The different converging levels show that it happens due to the application of the rules as opposed to the physical space limitation, such as the inability of the vegetation to grow below sea level or above snow to level.

3.2.3 Performance

Unfortunately, real time performance has not been achieved for large maps. The overlaying algorithms, provide far from optimal performance and have major potential for improvement.

Figure 3.2 shows the performance relation with the map size requested. There is a noticeable jump at 1024 due to the next tier of the map size being used. The square-diamond algorithm must use a map size of power of 2 plus 1, in order to be able to apply division appropriately. This means that the map size must be generated using quantized tiers of sizes. Therefore the generation will provide for a longer processing time every time a new tier is reached.

Figure 3.1: A graph showing the relation of the number of generations versus the population size. Please note the highest seed data set has only been ran using 20 maximum generation due to the exponential performance.

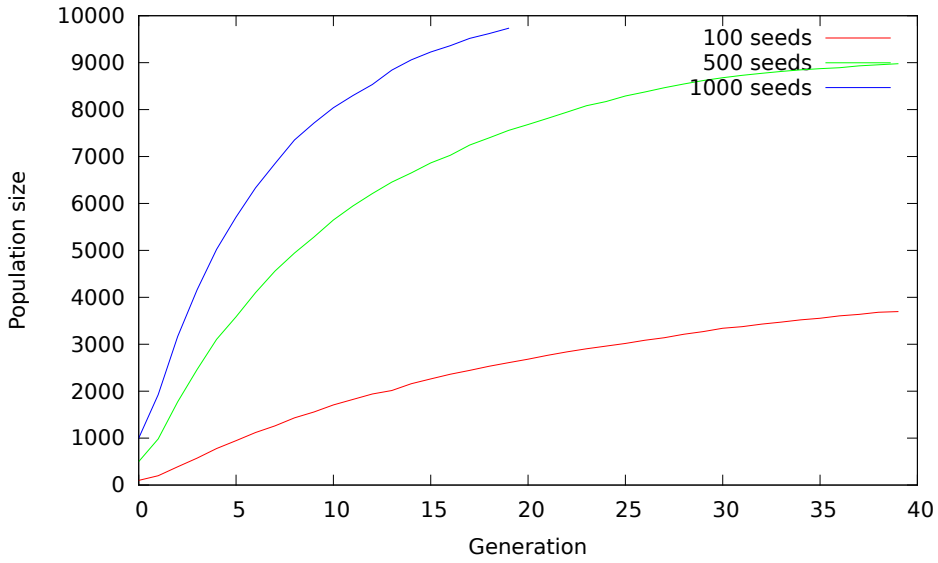
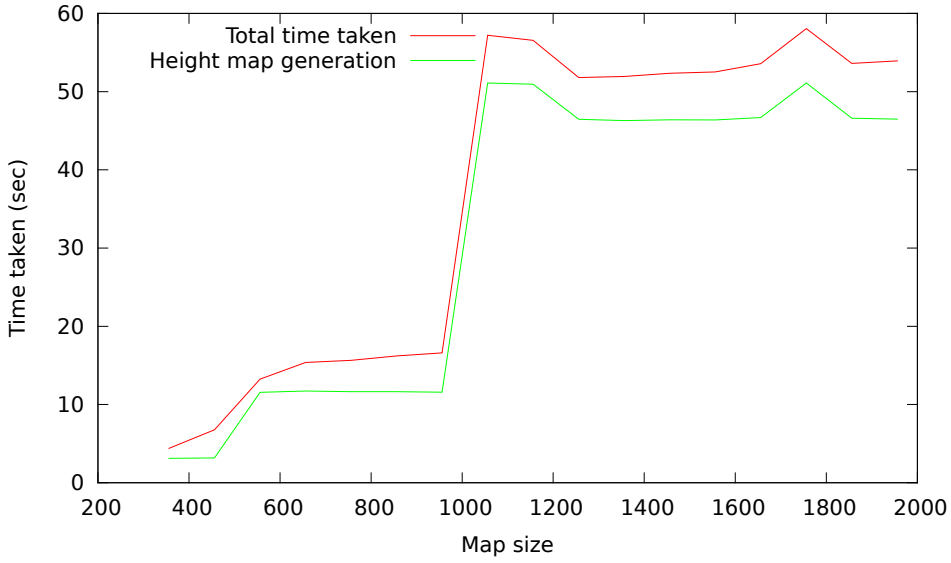


Figure 3.2: A graph showing the time take to compete the generation of terrain of various size. The attributes of the overlaying features are set, the aim is to demonstrate the performance of the terrain generation itself



Chapter 4

Project Management and Software Engineering Practices

4.1 Development Techniques

4.1.1 Overall Development Approach

From the requirements it is assumed that the project allowed for both agile development techniques and water fall like development model. Therefore a hybrid approach was taken, primarily focusing on incrementally adding features to the software, while later allowing the user to utilize features available in the developed prototypes.

4.1.1.1 Incremental Approach

Define system deliverables. I defined the complete generator system with all the primary objectives as my main deliverable.

Design system architecture. My system was designed with an incremental approach in mind, allowing us to easily add an increment to the existing functionality.

Specify system increment. One example of this could be to add the ability to create river source points and output them to a file.

Build system increment. The actual implementation of the river source module.

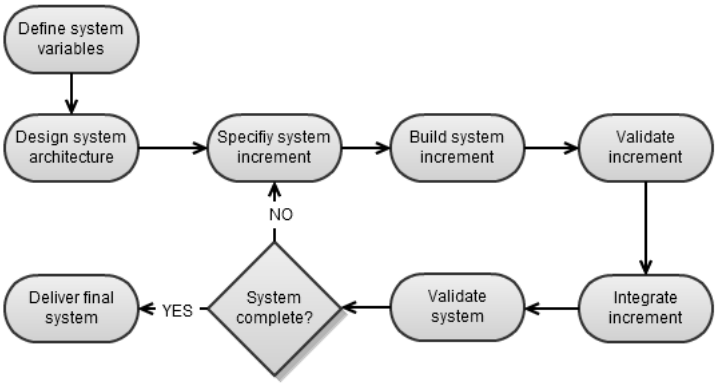
Validate increment. Testing that it works correctly.

Integrate increment. Adding the functionality to the overall system.

Validate system. Check that the rest of the system still works.

System complete. If not complete, continue incremental approach. Otherwise deliver final system. Each of the systems at this stage could potentially be seen as a prototype.

Figure 4.1: Incremental development approach. Adapted from pg. 394 of [8].



4.1.1.2 Rapid prototyping

The incremental approach had the additional benefit that it allowed me to have a working prototype for most of the development phase. If a new feature, such as river source creation, had to be added, it could be tested almost instantly, without significantly affecting the rest of the system.

4.1.2 User driven development

Through the later parts of the development, feedback from the developers of the user software became available. This lead to implementation of additional features, which greatly aided the interface available for the user. The cooperation involved providing the user with the generator software, the user then creatiing terrain to and fianlly giving feedback. The feedback was then used to either implement new features or correct existing bugs.

Chapter 5

Usability Survey

5.1 User feedback

To ensure the quality of the software produced, testing and development was carried out with cooperation with user software. Two of my classmates were working on software which involved rendering terrain, hence they were perfect candidates for testing the usability of the generator. The user's were asked the following questions after using the terrain generator.

5.1.1 Questions asked

The users were presented with a series of questions and were asked to respond with brief answers.

1. Did the generator output the required files to be used?
2. Which format did you request?
3. Was the file format as expected?
4. Did you have to make changes to the file format? If so what were they?
5. Once rendered, did the terrain contain the expected features?
6. If there were any drawbacks, what were they?
7. If you would like any extra features implemented, what are they?
8. Would you consider using the terrain generator for your software, if the above mentioned functionality was implemented?

5.1.2 Responses

5.1.2.1 080008164

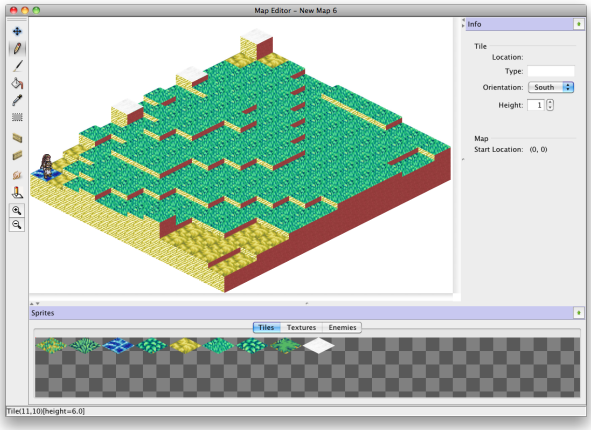
This project involved developing an Tactical Role Playing Game Engine. This involved implementing an underlaying rendering engine which was perfect as user software for my project. Bilal's implementation rendered the terrain in an isometric perspective which provided a different angle for viewing the generated terrain. The engine was also able to utilize the tile type and hence render tiles differently depending on it, giving the terrain a more meaningful representation. The results are shown in Figure 5.1.

The software required maps in XML format and the terrain generator was able to supply the height map in the correct format. There were a couple of minor issues involving inverted dimensions of the map and a

prerequisite library. However, otherwise the generator software proved reliable in satisfying the user software requirements. Additional features requested included generation of roads between settlements. Overall Bilal would be happy to use my generator for his project.

The answers given are available in the appendix section A.1.

Figure 5.1: Terrain rendered using the Tactical Role Playing Game engine.



5.1.2.2 050007653

This project was not video game based, which was a good indication that the scope of usage of this project is much large than originally intended. Jamie's project involved developing a phone app which allowed the user to view the surrounding terrain features. The height maps in use contained real data about the surrounding physical terrain. This terrain was then rendered from the user's point of view, slitting the height map into layers according to the distance from the user's coordinates. The results are shown in figure 5.2.

The application required very specific input, of a series of segmented maps, thus the same height map replicated in order to satisfy this. Nevertheless the generated maps were successfully rendered using the application. The feedback from 050007653 indicated that minor alterations of the terrain format were necessary. 050007653 also requested an alternative format for representation of settlements, so that it will also be possible to show their location relative to the application user's position. Finally Jamie suggested the generator would be a great testing tool for his application.

The answers given are available in the appendix section A.2.

5.2 Data quality analysis

To asses the quality of the maps produced, in terms of how natural it appears to a human viewer, a questionnaire web form has been developed. The form presents the user with multiple map examples and asks the viewer to answer questions with numerical values, mostly related with the natural appearance of the map and its features. The aim of the survey is to focus on how the terrain appears to a human eye, meaning the visualisation tool plays an equal part. Maps with both large and small scale attributes are presented. The user has to answer a series of questions by selecting a value between 0 and 9. The form presented to the user is described in the following section 5.2.1.

Figure 5.2: Terrain rendered using the terrain viewing application.

5.2.1 Map survey

'The following images show computer generated maps, primarily intended to be used in video games. The blue areas represent water (sea,rivers), yellow areas are sandy beaches, green areas are grassy plains, white are snowed in areas. Trees are drawn as little drank green triangles throughout the landmasses. The map also shows locations of settlements shown by a black marker and a name.

Please answer the below question using a choice between 0 and 9, 0 representing complete disagreement and 9 representing complete agreement with the statement. Once all of the values have been selected please click submit. If you need a closer look at the map please click to zoom in, or if required, right click and save as to view using software of your own choice.

The data gathered is completely anonymous and will be used for scientific research.'

The user is the presented with two terrain images, one made specifically large and primarily made up of scattered islands. This map features many spread out settlements and rivers. The second map presented is a small map with one continent and a few separated islands. This map contains a few rivers, a few sparse forests and several settlements. The main difference being stressed is one maps is large and split up while the other is mostly continuous and much smaller. For the images used please refer to section B.1.

Each map had a related set of statements, where the user should assign a value, 0 being complete disagreement and 9 being complete agreement:

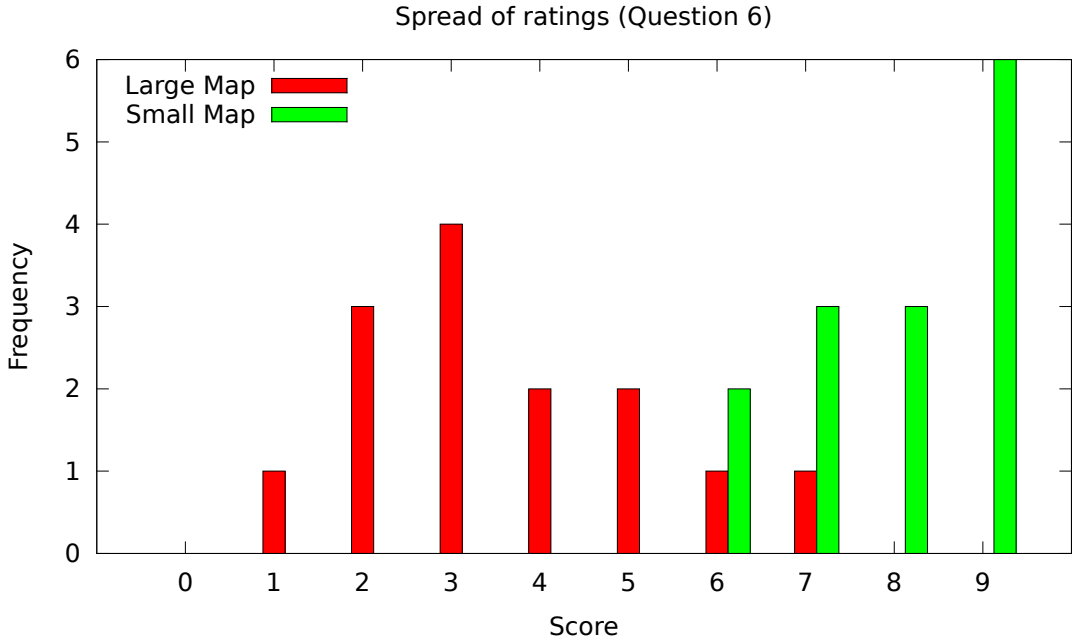
1. The shape of the terrain itself looks natural
2. The appearance of the rivers looks natural
3. The appearance of the forests/vegetation looks natural
4. The locations and names of settlements make sense
5. The map reminds you of a real location in the world
6. The map as a whole looks natural and makes sense
7. 'I cannot wait to start an adventure and begin exploring this world!'

The most relevant questions in terms of the objectives of the project are 6 and 7. These indirectly represent how immersing the environment will be for the player, as well as the motivation for exploring the world. The outcome of the survey is seen in figure 5.3.

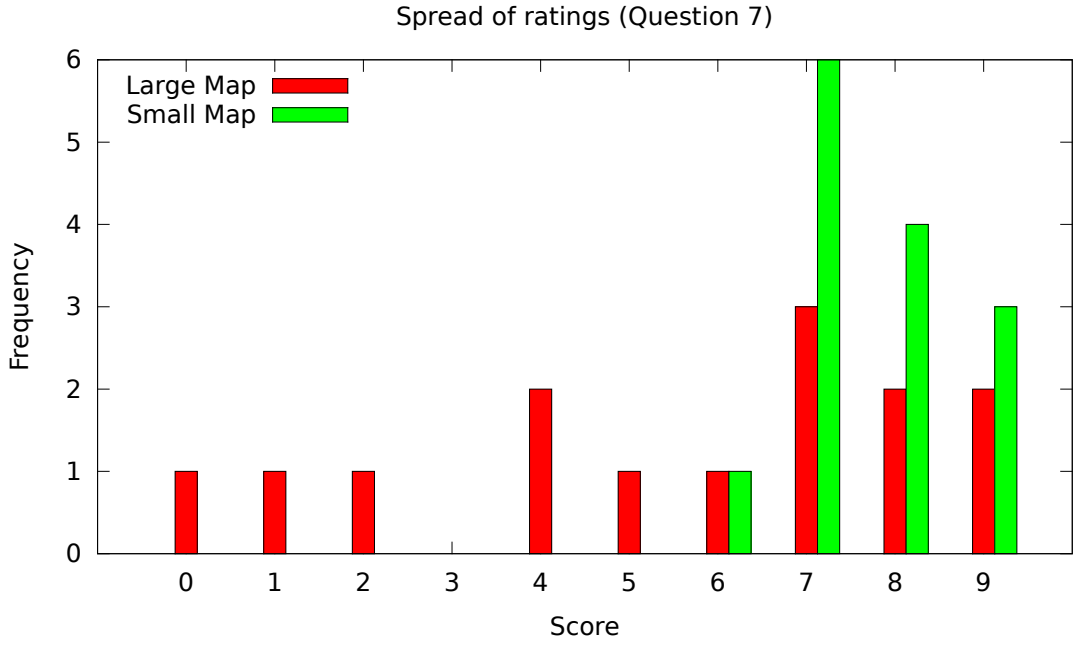
From the graphs shown in figure 5.3, it can been seen that the small map took the preference of the viewers. It both, provides for a more immersing appearance by having a more natural looking terrain, and promises a more fun experience by not overloading the view with features hence making it easier to relate

Figure 5.3: Graphs depicting the spread of scores in the terrain quality survey

(a) Graph showing the spread of scores for the question related to the appearance of the map as a whole.



(b) Graph showing the spread of scores for the question asking whether the user would like to enter the game world as a character in the game.



to the features already present. This suggests that a smaller map would be more preferable by everyone for usage in a game.

The large map shows a wider spread of scores. This implies that the map preference is down to opinion of the users. Some users might enjoy gigantic worlds while others find them too intimidating or requiring too much attention to enjoy.

It can be concluded that in terms of human preference, the less is more approach will provide for a more reliable immersing experience. Creating gigantic worlds and presenting them to the user in the entirety might intimidate them or make them loose their focus, however the survey proved that there are still users that enjoy exploring such a large world.

Chapter 6

Critical Review and Conclusion

6.1 Requirements

Below is the revised list from section 1.4. This list states whether the required functionality has been implemented succesfully, marked by $\surd$.

6.1.1 Primary Requirements

1. $\surd$ The generator is able to output a two dimensional matrix of values. The matrix is generated using a fractal approach, such as the square-diamond algorithm.[2]
2. $\surd$ The generator takes the matrix size properties as an input from the user.
3. $\surd$ To add a layer of realism to the resulting output, height map and erosion algorithm are applied.
4. $\surd$ To simulate further realism, in this case, formation of gorges and specific peaks and an impression of tectonic plate presence, a Voronoi diagram can be generated and interpolated together with the existing matrix.
5. $\surd$ The generator should be able to output the matrix into a file representing the date in a defined, standard format
6. $\surd$ The generator should output the data into the file in another, non-standard format, such as a representation using XML
7. $\surd$ The generator should determine a set of points to be river sources according to specified constraints.
8. $\surd$ The generator should determine a set of points to be the initial vegetation seeds according to specified constraints.

6.1.2 Secondary Requirements

1. $\surd$ To aid the development and testing a tool has been developed to allow for rendering of the terrain.
2. $\surd$ A concept of scale has be introduced, regarding the amount of information represented by the single element of the height matrix.
3. $\surd$ The generator allows for a specific manipulation of the properties of the terrain, i.e.:
   (a) The seeded height values and the maximum difference between any two elements should be used as input for the generator.

(b) A value should be provided as input to determine the 'roughness' of the terrain. For example low values would create a smoother looking terrain, while higher values would result in an increased number of peaks.

4. $\surd$ A configuration file has been made available for the user to specify the input parameters.
5. $\surd$ A set of points has be determined to represent the central locations of settlement according to specified constraints.
6. $\surd$ An algorithm has been implemented to iterate the produced set of river sources and extend into full rivers. These are represented using a standard XML format and output in a file.
7. $\surd$ An algorithm has been developed to iterate over the produced set of vegetation seed points and grown them into vegetation groups. The user is able to specify the number of generations the algorithm should have available to develop the groups. The resulting data is represented using a standardised XML format and stored in a file.
8. An algorithm has not been implemented to iterate the produced set of settlement points and procedurally generate the containing buildings.
9. The generator is not able to apply further erosion effects on the terrain data, after the initial height matrix has been generated.

6.1.3 Tertiary Requirements

1. The generator is not able to create adjacent terrain with respect to already existing terrain. The border between the two data sets should be merged to the point of seamless terrain transition.
2. Concurrent generation of the terrain should be able to provide real time performance of the generator while creating realistic terrain, but that heavily depends on the input from the user.

6.2 Emergent Possibilities

During the development of the software, certain functionality has emerged which was not mentioned in the requirements. This extra functionality primarily provided interesting simulation based possibilities.

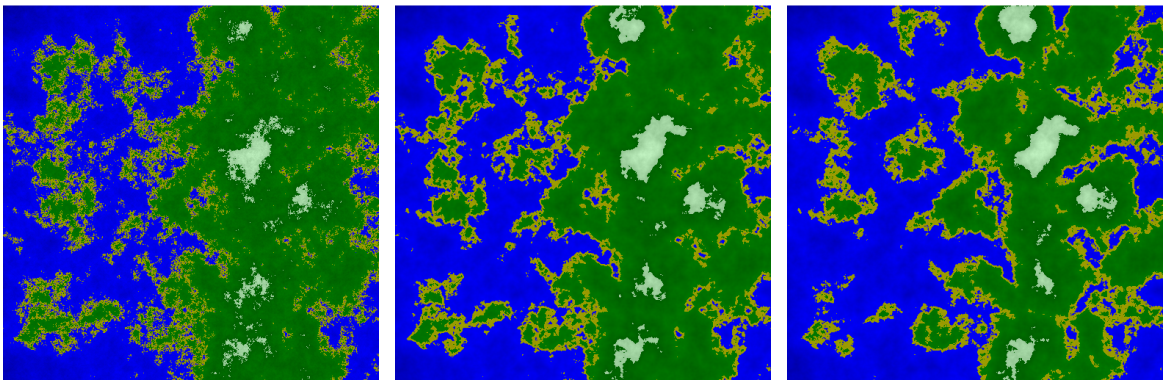
An example simulation would would be based on the ability to specify the number of erosion iteration coupled with specification of the vornoi diagram interpolation value. When maps are created with an increasing value of the mentioned attributes, an emergent effect of developing terrain is produced. When shown in rapid succession, an animation of terrain eroding is created. A few of the key frames are shown in the figure 6.1.

A similar concept can be applied to the growth of vegetation. The frames of animation can be seen in figure 2.11.

6.3 Possible Future Work

There is a great variety in the options for future development. These can be primarily divided into developing the current features and implementing new functionality.

Figure 6.1: Frames of erosion animation



6.3.1 Existing Features

The height map generation would be one of the first place for further development. In particular, during the initial stage of generation, the vornoi diagram points are randomly placed on the map. The configuration file can be appended with a list of specified points for the user to specify. In addition, the generator should be able to move these points in groups to allow for simulation of tectonic plate movement.

Further more concerning height map generation, the erosion algorithm can be refined to give a more realistic effect, by spreading the fallen material to multiple neighbours. In addition a different erosion algorithm can also be implemented, dealing with the effects of hydraulic erosion. This will involve simulating water flow throughout the terrain, and shifting material accordingly.

Next, given a newly eroded terrain, the river generation will have a different appearance. This will allow for additional rules added to the river flow, such as forcing the river to meander and even form ox-bow lakes.

Settlement generation can have additional constraints applied, resulting in a more realistic positioning. Also, the name generation can be enhanced. By assigning properties to certain prefixes and suffixes of the names, it is possible to have the settlement name related to the surrounding features. As suggested by the users, additional information can be attached to the settlement data to allow for easier rendering.

6.3.2 New Features

Starting with suggestions from the user's, the following additional features are possible improvements for the project. These improvement do not take into account the tertiary requirements.

A new generator, or perhaps and extension of the settlement generation, can be developed, producing roads between settlements.

Once the procedural generation of the internal structures of the settlements has been implemented, simulation of the inhabitant population has be simulated.

6.3.3 Architecture Refinement

To allow for a more modular use, the generator structure can be rearranged for easier inclusion into other source files.

6.3.4 Tools

Additional tools can be created for a different rendering of the terrain. For example an withered effect can be added to the map, to give an appearance of it being drawn by hand. Also a 3D view can be implemented for an even closer analysis of the generated features.

6.4 Lessons Learned

One of the first flaws noticed with the generated terrain became apparent during the implementation of the erosion algorithm and then creating rivers. The terrain seemed to be very rough and forcing the rivers to form a lot of lakes due to poor flow paths. This is supported by observing figure 5.2. The surface of the terrain has almost a fluffy property.

The main reason for this is the use of integers for storing the height data. Initially chosen to gain a performance advantage, the integer type has proven to hinder the quality of the terrain. After being eroded, the integer differences between adjacent blocks were too crude to express subtle differences between heights, hence procuring the rough surface effect.

Another helpful addition to the generator would be a set of simple unit tests. This came apparent while testing the software with a user. Discovery of multiple minor bugs could be easily fixed by running a test suite.

Finally, the initial requirements for the generator have been proven very ambitious. The time constraints given did not allow for completion of some of the requirements.

6.5 Overall Conclusion

Over all I have learned a lot about developing software for a user. The prototype-based software engineering approach greatly aided the development of the project.

The initial requirements have proven to be overambitious, due to the time constraints, however the overall project did not suffer.

The availability of the users has proven essential for testing of the project and has given me a advantage. The generator working with different software is crucial evidence of the generator working reliably.

I have delivered a reliable and robust product which usable by software developed by others. It can be safely concluded that the generator does not have to be restricted just to the Role Playing Game genre or even games as a whole. The project has proven to successfully not only generate terrain but provide insight on behaviour of its overlaying features as well as the player's preference of the game's world.

Appendix A

User Survey Answers

A.1 080008164

1. Did the generator output the required files to be used?
   Yes.

2. Which format did you request?
   XML.

3. Was the file format as expected?
   Yes.

4. Did you have to make changes to the file format? If so what were they?
   I had to invert the width and height of the xml, since you had them the wrong way round.

5. Once rendered, did the terrain contain the expected features?
   Yes.

6. If there were any drawbacks, what were they?
   You said you statically linked libyaml when you did not, so I had to change the compilations settings to statically link the library, for embedding.

7. If you would like any extra features implemented, what are they?
   Add 'roads' between 'cities' on the world map generator.

8. Would you consider using the terrain generator for your software, if the above mentioned functionality was implemented?
   Yes.

A.2 050007653

1. Did the generator output the required files to be used?
   Yes.

2. Which format did you request?
   Standard 401 by 401.

3. Was the file format as expected?
   Yes.

4. Did you have to make changes to the file format? If so what were they?
   Carriage return after file size specifier.

5. Once rendered, did the terrain contain the expected features?
   Yes.

6. If there were any drawbacks, what were they?
   None.

7. If you would like any extra features implemented, what are they?
   Named features on the national grid coordinate system so i can name them on the map (my code is inflexible on that).

8. Would you consider using the terrain generator for your software, if the above mentioned functionality was implemented?
   Yes, for testing that random terrain can be generated.

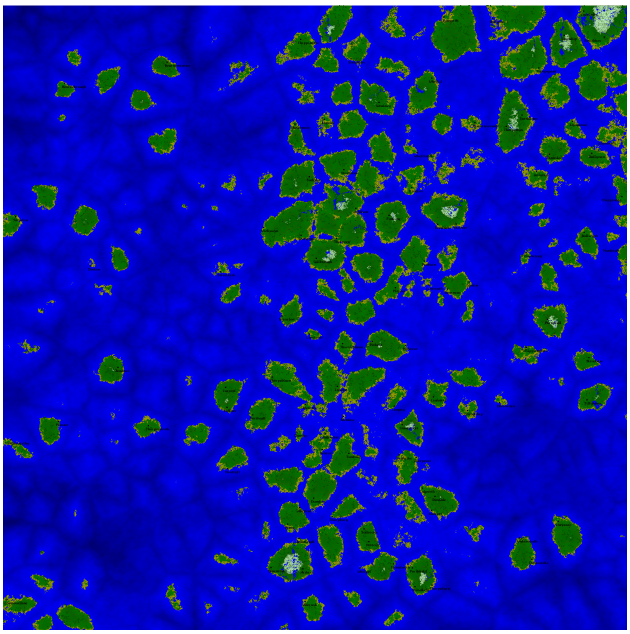
Appendix B

Quality Survey

B.1 Answer data

B.1.1 Large Map

Figure B.1: The large map presented to the user.



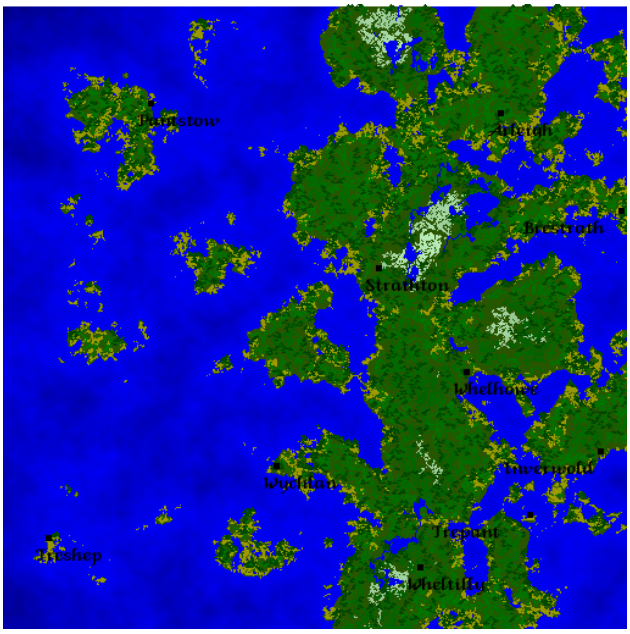
Listing B.1: Data gathered from the survey, each row represents a user. Please refer to section 5.2.1 for the questions.

```
7 4 6 0 9 5 0
1 5 5 6 2 4 2
4 5 4 7 4 4 6
1 1 7 3 0 2 1
6 2 1 5 1 2 7
7 6 4 5 2 3 4
```

```
2 2 2 6 2 2 7
5 4 4 6 2 3 9
7 4 7 8 1 7 7
8 6 7 6 4 3 5
7 7 6 7 2 6 9
0 1 3 4 4 1 8
```

B.1.2 Small Map

Figure B.2: The small map presented to the user.



Listing B.2: Data gathered from the survey, each row represents a user. Please refer to section 5.2.1 for the questions.

```
9 6 6 4 9 7 8
6 3 6 7 5 6 7
8 6 6 7 7 9 9
8 7 7 6 6 8 6
5 3 8 7 8 9 8
8 4 6 3 6 9 7
6 7 6 6 6 6 7
6 6 7 8 9 9 9
8 6 5 7 6 7 7
8 8 7 7 7 8 7
8 6 6 6 7 8 7
7 7 8 4 7 9 8
8 6 3 5 7 7 8
```

Appendix C

User Manual

C.1 Perquisites

The project requires the following libraries:

- `libyaml.h`
- PIL (Python Imaging Library)
- `python-lxml`

C.2 Compiling

The source code along with the makefile is located in the `Default/` directory. Before compiling the directory needs to be cleaned using the `make clean` command. Then the program can be compiled using the `make` command.

C.3 Running

The generator can be ran using the `./terrain_generator` command. A `-h` flag can be added for a help display. The generator uses `config.yaml` as the default configuration file.

Bibliography

[1] Matt. Barton, *Dungeons & Desktops*, A K Peters, 2008.

[2] Paul. Martz, "Generating random fractal terrain", 1997.

[3] Jacob. Oslen, "Realtime procedural terrain generation", October 31 2004.

[4] Eric W Weisstein, "Voronoi diagram".

[5] Paul. Callahan, "What is the game of life?", 2002.

[6] "Python Lxml Library", Apr. 2011.

[7] "Python Imaging Library (PIL)".

[8] Ian. Sommerville, *Software Engineering*, Addison-Wesley, 8th edition, 2007.