Random recursive map generation with hexagons

Edward John
79916470

MEng. Computer Science

Third year project for the University of Manchester, Computer Science

Supervisor, Jim Garside

Abstract

This document reports the work of experiments that investigate the concept of procedurally generating fractal landscapes on a grid of tessellated hexagons. The experiments aim to prove whether procedurally generated landscapes created from subdividing hexagons produce a more visually appealing or realistic image. Traditionally, squares or triangles have been used to represent landscapes in games. Hexagons have occasionally been used, but there are few examples of procedural generation. Hexagons occur in nature and have different properties that could enhance the procedural generation of content for the use in cinema and gaming. This project explores some aspects of their use.

Three experiments are developed: procedural generation through subdivision of uniform squares; rendering hexagons using Windows Presentation Foundation framework; and rendering hexagons in 3D using OpenGL. The first two help prototype a Windows 8 application which demonstrates procedurally generating a hexagonal terrain based on height values which are then used to derive a colour such as blue for water and green for high land. The application also proves the ability to be able to zoom into one particular part of the map and have that part of the map subdivided again to reveal more detail. This process is theoretically infinite, only limited by the memory constraints of the underlying computer. The third experiment assesses the visual impact and feasibility of creating a three dimensional world using hexagons.

The report discusses the different phases of development of the application and how individual parts of the system were refined using trial and error to improve the rendered output. The images included show the results of these improvements and the potential of hexagons in procedural generation.

Acknowledgements

The following people are appreciated for their help and support:

Jim Garside for project inspiration, hexagon subdivision and supervision of project.

Graham John for report review and support.

James Robertson for motivation, testing and feedback.

Table of contents

Table of figures

1 Introduction

Over the past twenty years, films, television and video games have required increasingly realistic content generated or enhanced by computer. Requirements include landscapes, cityscapes, material textures, particles, water, and supernatural phenomena. Originally, content was hand made using full-scale sets, models or drawings, however producers needed to reduce costs and speed up production while improving the appearance and realism. Once good algorithms have been devised, procedural generation provides a rapid and repeatable mechanism to generate varied but lifelike results. Another advantage is the reduction of storage as content can be generated on demand.

Hexagons have properties, such as the ability to tessellate, which make them very interesting shapes to divide content such as landscape up into. They occur as crystalline shapes in the natural world and have been popular in the design of board games and video games.

The aim of this project was to procedurally generate a landscape made only out of hexagons that exhibit fractal-like properties so that a map can be enlarged indefinitely to reveal more detail (in the spirit of a Mandelbrot set [1], see section 2.2.1).

The report summarises research undertaken on procedural generation and hexagons, the experiments and prototypes developed, the design decisions made to optimise results, and provides illustrations of the output achieved.

1.1 Inspiration

Procedural generation is commonly defined as algorithmically generating content as opposed to pre-preparing the data. For example, applications can generate textures that look like water or clouds, or terrain with contours that you might find naturally. It may be used to reduce storage requirements – particularly where it is uncertain which data will eventually be needed.

Procedural generation involves algorithms, often mathematical or logical. The creator does not necessarily have to be an artistic or graphics designer, but needs to be good at devising algorithms that can create realistic content. This can result in alienation of artists who are challenged by the automation of content production, however scientists and mathematicians are rarely trained in aesthetics, so a better approach is to train artists to use procedural generation as a tool to improve their productivity rather than as a replacement of their creativity [2]. Procedural generation can provide a significant advantage when repetition is required, such as the armies animated in The Lord of the Rings film trilogy.

The study of real life to help design procedural algorithms can be fascinating. Many plants, trees and other organic objects often sustain a complex pattern, which can be replicated using simple formulae.

1.1 Uses

Procedural generation of content is useful for various subjects. A common one is games where there is a need for generated content that looks realistic without needing to be handmade. Examples are clouds, terrain maps, particle effects and water. These examples don't just apply to games; they are now frequently used in TV shows and films where Computer Generated Imagery (CGI) allows procedurally generated content to be used. Any new research or experiments could enhance, improve or give a different perspective to how procedural content is currently generated.

Whilst being used to reduce artist time and increase realism, it is also used to reduce storage requirements. Particularly textures used for clouds, smoke and fire, can be generated at runtime or in real-time for games so that they do not take so long to load up or to reduce the storage needed for the game when distributed on media of limited capacity such as discs or cartridges.

Whilst films typically use procedural generation for artificial creations such as large city backdrops, they might often use algorithms to generate landscapes particularly in fantasy films or science fiction films where not only is there not a suitable location in real life to film but also to produce nature that looks realistic but is not found in real life – for example floating islands, large frozen terrain or giant plants.

1.2 Objective

The objectives of the project are to experiment and investigate the concept of recursive procedural landscape generation using hexagonal tiles; and develop an application that demonstrates the results of the research, including the ability to zoom into a particular hexagon where more detail is generated recursively and infinitely in the spirit of the Mandelbrot set.

1.3 Method

The first task was to undertake some research on the topic, gathering information about procedural generation and hexagons and existing developments combing both. The next task was to produce two prototypes, one to experiment with procedural generation recursively through fractals (see section 3.3.1) and another to experiment with rendering hexagons (see section 3.3.2 and section 0). After these prototypes were completed, the application was designed based upon the lessons learned from the prototypes. Unit tests were developed as the application was written to allow incremental testing. Once the application was completed, more thorough testing was undertaken before preparing a release for an alpha trial where a small selection of users were asked to try the application. Any bugs identified were fixed and any suggestions for improvement were assessed and the best implemented into the final version, which was then used for the evaluation and drawing of conclusions.

2 Background

2.1 Hexagons

Regular hexagons are six sided polygons where each angle between sides is equal. They were popular once in old video games as the geometry of the playing map. Hexagons look closer to circles than square tiles without being too complex.

A notable property of hexagons is their ability to tessellate (fit together with no overlaps or gaps). With the exception of squares and rotated triangles, hexagons are the only other regular polygons which can tessellate.

Hexagons are seen in nature. Due to the attributes of hexagons such as its six point structure, it serves as a very useful material for building structures as it tiles an area with minimal surface area [3]. Honeycombs built by bees take hexagonal form and so does the chemical structure of graphine, with its six sided hexagonal lattice of carbon bonds.


[1]

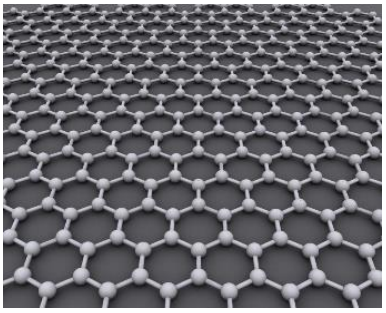

[2]

Figure 2.1 Examples of tessellated hexagons seen in nature.

Hexagons make an appearance in board games. With six sides, each tile has six neighbours as opposed to a more conventional square tile board where each tile only has four neighbours. This requires more sophisticated strategies to play the game as the number of possible options / choices is larger.


[3]

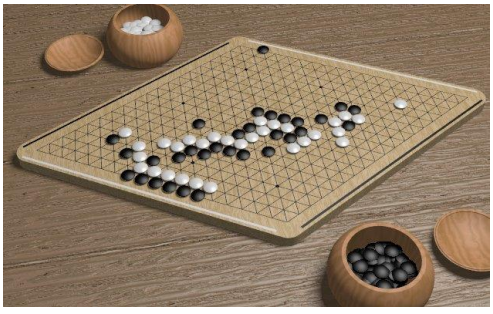

[4]

Figure 2.2 Hexagonal chess and Hex, examples of hexagonal board games.

With hexagons being used for board games, it is inevitable that they make an appearance in video games. Puzzle games such as Hexic and Entanglement are turn based puzzle games. They again use the concept of having six neighbours for each hexagonal tile as a logic element to the game. Sid Meier's

[1] http://www.bienenwabe.eu
[2] http://en.wikipedia.org/wiki/File:Graphen.jpg
[3] http://www.chessforums.org/general-chess-discussion/3911-my-new-toy.html
[4] http://en.wikipedia.org/wiki/File:Hexposition02.jpg

Civilization V is a turn based strategy game where the game world is constructed on a hexagonal grid, but unlike this project does not use recursion to provide greater detail when zooming in.

[5]


[6]

Figure 2.3 Examples of video games (Hexic and Sid Meier's Civilization V) that exploit hexagons.

The above examples show a clear progression from the existence of hexagons in the real world to the use of them in video games, and this project develops this idea further by using them in procedurally generated landscapes.

2.2 Procedural generation

Procedural generation is the concept of algorithmically generating a natural feature such as landscape or living nature automatically. It is commonly used in modelling real world simulation, video games or films.

Fractals are known as a shape which exhibits self-similarity. It is common for fractal like shapes to recurse indefinitely by a repetitive or iterative phenomenon. Fractals occur naturally, such as coast lines and in plants.

2.2.1 Mandelbrot set

The Mandelbrot set is an example where iterating the set of values in a complex quadratic polynomial can produce a detailed and infinite sequence of self-similar patterns [1]. Figure 2.4 shows the full set zoomed out. The overall pattern can be seen in several different places, sizes and orientations.

Mandelbrot sets are often shown as an animated clip of zooming into a particular area where more detail is revealed and the process repeated to, what seems a cycle that lasts forever. This is achievable as the storage complexity for zooming in is $O(1)$, i.e. calculations can be repeated on the same data. Old data can be discarded and the new generated data set is no larger than the previous set. This concept is the basis for the repeated zooming in of a hexagonal landscape, where the memory usage needs to be kept balanced.

The Mandelbrot set is an example of a non-random fractal, it is completely derived from a formula. Procedural generation tends to exhibit pseudo-random properties so that a variety of different shapes and patterns can be produced.

[5] http://www.techknight.com/blog/2005/12/29/hexic-hd-for-xbox-360-works-on-the-pc.html
[6] http://www.gamersgate.co.uk/DD-CIV5/sid-meiers-civilization-v

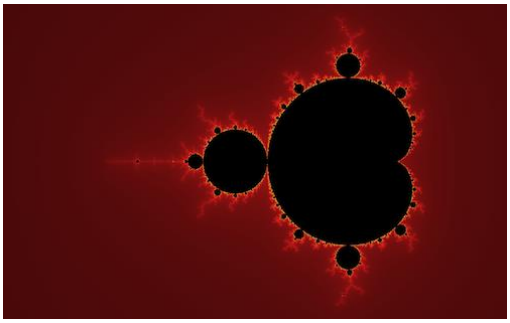


Figure 2.4 Mandelbrot set, zoomed out [4].

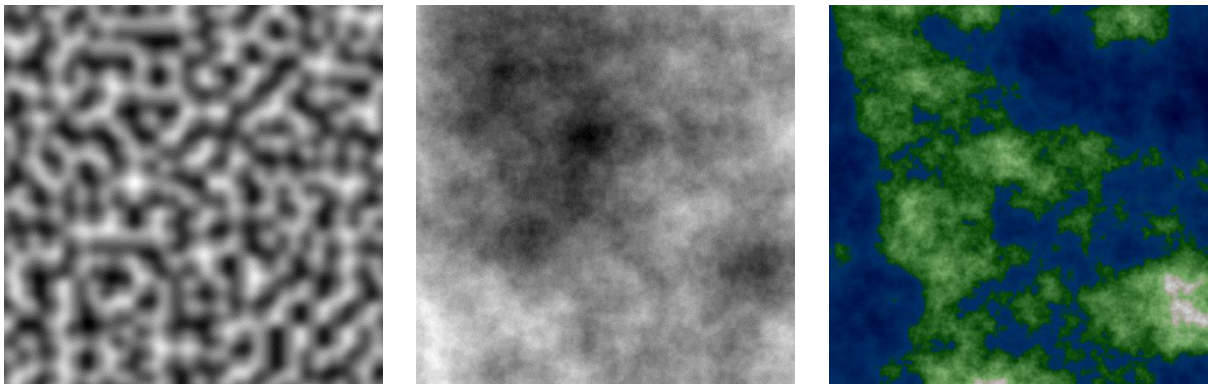
2.2.2 Perlin noise

Perlin noise is a computer-generated visual gradient noise effect for producing random variation which exhibits a natural appearance. Since its introduction in 1984, the algorithm has been used widely in graphics [5], particularly where available memory is limited. While pseudo random number generation can produce adequate unpredictability, it lacks natural realism as random number generators tend to produce harsh changes in value [6].

Perlin noise can work in one or more dimensions. In a two dimensional case, a grid of vectors can be created using a pseudo random number generator. These vectors can represent gradients of each grid point. The dot product for each grid cell can be calculated using the gradient vectors and then interpolated together using either a linear or spline interpolation function. An example of a generated two dimensional Perlin noise texture can be seen in Figure 2.5a.

To create a fractal-like appearance, the Perlin noise process can be repeated multiple times using different wavelength and amplitude values also known as the persistence. These different instances can then be added together and thus creating a smooth fractal noise. This process is called Fractional Brownian Motion [7] and can be used with any noise function. An example of this process applied to two dimensional Perlin noise can be seen in Figure 2.5b. Figure 2.5c shows another example with landscape colouring applied.

Perlin noise is a $O(2^n)$ function [8] and thus inefficient if a large amount of noise data needs to be created. Ken Perlin, designer of the classic Perlin noise function later improved the algorithm in 2001 to overcome two deficiencies (second order interpolation discontinuity and unoptimal gradient) [5]. This is known now as Simplex noise and has a time complexity of $O(n^2)$ [8].



(a) (b) (c)

Figure 2.5 Perlin noise [7]

2.2.3 Midpoint sub-division

Another common method for procedural generation is midpoint subdivision. This works well with mountains and coast lines where the variation of height or distance can change. It provides easy control over the level of detail as detail is gained by repeating the process as many times as necessary. This method has the advantage of producing a simple shape initially and then adding more detail to it when required such as when zoomed in on. This is useful in games where far away objects can be rendered with less detail to improve the rendering time performance but rendered in more detail if close up.

The algorithm works by generating a basic three point shape first such as a large triangle. Each section between two points can then be split into two by creating a new point centred between the two points either side. This point can then be offset by a random amount a particular axis to form a more complicated shape. Repeating this process will add double the amount of detail each time, without changing the basic shape of the overall pattern too much providing the offset range is limited to at most half of the difference between the two neighbouring points. An example is shown in Figure 2.6.

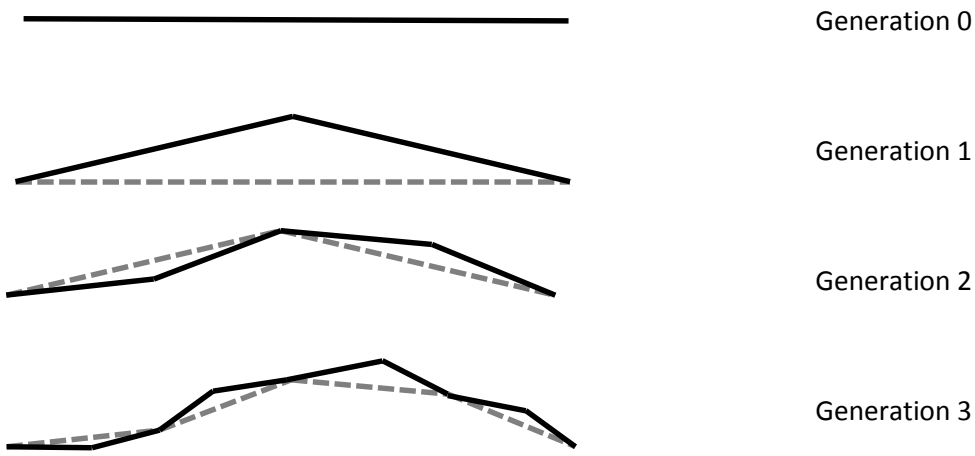


Figure 2.6 Sub-dividing a mountain to produce extra detail.

2.3 Hexagons and procedural generation

One of the issues with using procedural landscapes on square grids is that the results tend to contain unwanted artefacts. Mandlebrot described these artefacts as an unwanted texture that looked like folded and crumpled paper, and suggested the use of triangular or hexagonal lattices to overcome this [9].

While hexagons can be tessellated quite easily, sub-dividing them into smaller hexagons is not so easy to achieve. To subdivide a hexagon into smaller ones, the smaller hexagons will have to overlap some of the edges of the parent hexagon whilst leaving gaps in the other edges. If these gaps are on opposite side of the hexagons that overlap an edge, the neighbouring cell's hexagons can fill that gap with their hexagons that overlap their edges. Figure 2.7 Subdividing hexagons shows an example where one hexagon is split up into three hexagons.

Another possible solution is to allow child hexagons to overlap each other so that they can be placed in such a way that it fills the entire parent hexagon without any hexagons overlapping the parent hexagon's edges or leaving any gaps inside the parent hexagon. If the child hexagons were slightly transparent, then their colours could blend together which might produce a soft transition between two terrains.

Because hexagons have a more rounded outline than squares, with less sharp changes in direction at points, the edges of the hexagons could be used to represent thin terrain. Rivers and roads are good examples where you might have a thin blue or grey line flowing down the edges of hexagons and also forking into two different paths which would be at 120° angles from each other.

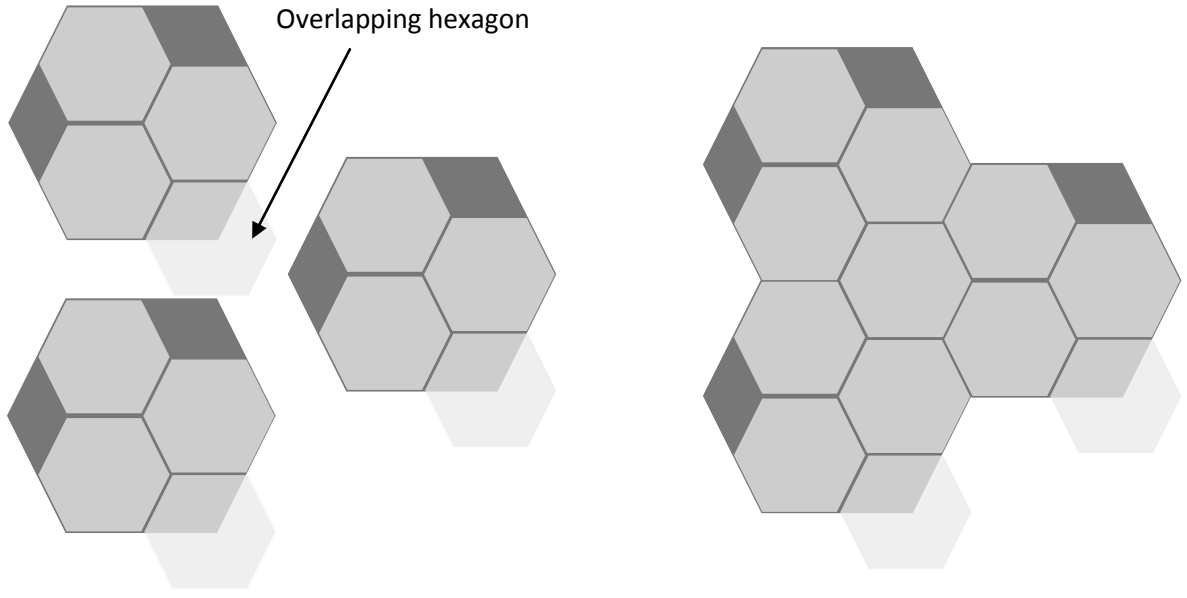


Figure 2.7 Subdividing hexagons

Another example of hexagons used in procedural generation is seen in the article Polygonal Map Generation for Games [10]. It explains that a grid of irregular polygons can be created by perturbing a regular hexagonal grid. However Voronoi polygons were used in favour of this.

3 Design

3.1 Programming language and library

Before commencing further design and development, a suitable programming language and suitable libraries needed to be chosen. Due to the nature of the project, an object oriented language was selected to break up the functionality into constituent classes and methods, for example to represent hexagons as objects and rendering as a method. The application needed to display multiple polygons in real-time. For maximum performance, hardware acceleration was used rather than software rendering. Possible APIs compatible with Windows 8 are DirectX and OpenGL [11], and the former was used as it is built into the framework libraries.

As this is a proof of concept application, it was not important for the application to be able to run on many platforms, which would have greatly increased the complexity of the implementation. So that more time could be spent on the core functionality, a Rapid Application Development (RAD) methodology was used.

A Windows Store application, also known as a Windows 8 app [12], was the chosen architecture as it provides a simple tactile interface on touch screen devices and can be written in C# on top of Microsoft's layout mark-up, Extensible Application Markup Language (XAML), which is hidden from the developer and uses DirectX to render the controls. This uses the .NET Framework [13] which is a large library with many useful utility classes and methods, helping reduce development time.

3.2 Tools

Visual Studio 2013 Ultimate is an integrated development environment which supports creation of C# *Windows Store* applications by providing a project workspace, syntax highlighted text editor with IntelliSense (auto complete) and debugger [14]. It also includes an editor for designing the user interface for Windows Presentation Foundation (WPF) applications and Windows 8 apps. It is widely used in the business environment for writing large scale systems and includes various tools to help with this. One of the useful tools is a unit test framework, allowing code to be written which tests the application logic automatically with the use of assertions to check that the output data from method matches the expected output data. The tests can be run automatically at any time and be organised into groups. The debugger includes a simulator that allows an application to be tested for touch screen devices when not on a touch screen environment.

LINQPad [15] is a Windows application written by Joseph Albahari which allows one to write C# (and other .NET languages) expressions, statement blocks or small programs in a scratchpad environment which is compiled and executed quickly, outputting rich formatted results. This was found to be very useful for testing small blocks of code without recompiling the whole project again. It was helpful prior to starting the final implementation solution, for prototyping and experimenting with concepts.

Git was used for source control, as it commits changes to a local repository providing a history of code revisions [16]. **GitHub** is a web-based hosting service for Git repositories and provides the server storing a central repository and data backup [17]. **GitHub for Windows** is a graphical user interface for Git and automatically provides the synchronisation operation for pushing and pulling change sets from the central repository on the GitHub servers [18]. Git provides branch control which allows work to start on new features without disrupting the main codebase, and the ability to switch back and forth between branches when necessary. Branches can be merged so that programmers can keep a primary stable branch of the codebase and include working change sets from other branches when they are ready. Figure 3.1 shows several commits on different branches and how branches can be merged back into the original branch.

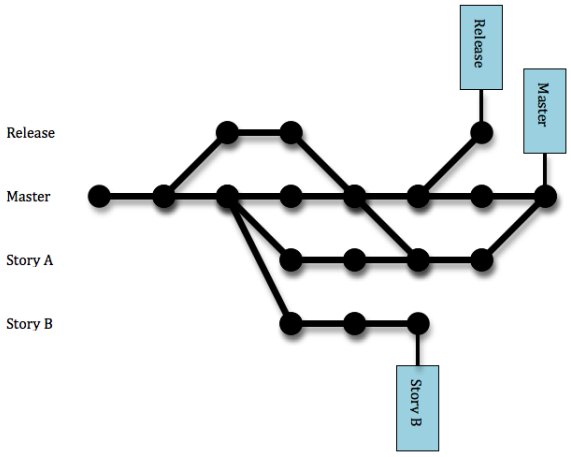


Figure 3.1 A typical state of branches for a GIT repository [19].

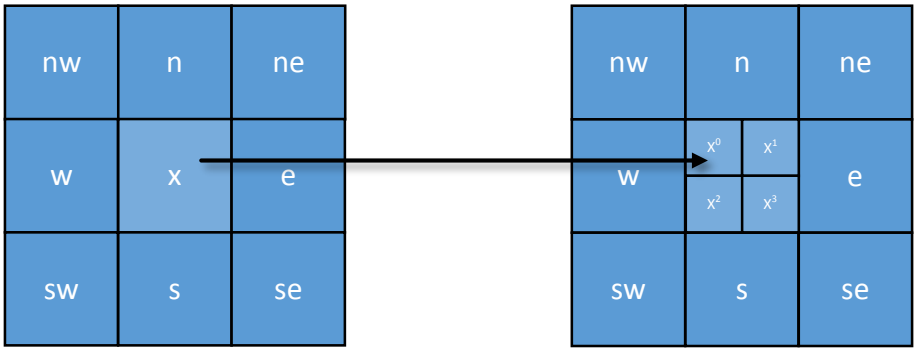
3.3 Experiments

Before making the application, two concepts were experimented with so that the better design could be chosen, reducing the possibility of redesign during implementation. As fractal generation and hexagon rendering are individually complicated in their own right, experimentation and prototyping of these parts were done separately.

There were three experiments in total. The first, investigated procedurally generating terrain using fractal recursion of square nodes. The second tested the capabilities of Windows Presentation Foundation (WPF) [20] by rendering 2D hexagons and tessellating them. The third was an attempt at rendering hexagons in 3D using OpenGL, although this was not a core objective.

3.3.1 Experiment A – Square fractal recursion

To experiment with fractal recursion, mid-point subdivision was used with squares, a 2D implementation of a similar process seen in section 2.2.3. In this experiment, each square is referred to as a grid cell. A grid cell has a height (a value between 0 and 1, where a value of less than 0.25 represents depth of water, and 0.25 and over is height of land). A grid cell can be subdivided into four grid cells which are half the width of the original. This in turn means that four height values (one per subdivided cell) are to be derived from one.



When dealing with one dimensional subdivision as shown in section 2.2.3, the value of the new point is calculated as the average (midpoint) of the two neighbouring points which is then offset by a random amount approximately in a range of half the difference between the two neighbouring point values. In this situation, there is a centre parent height value and eight neighbouring cells around it. The average could be based on all eight neighbours and the original parent cell, or just four neighbouring cells (north, west, south, and east). It was decided that the former method was more appropriate. Without any further changes to the average height, this would ultimately smooth the map if this was applied to all the cells. This is good and bad, as in many cases smoothing the map increases realism but reduces variation. It would also not produce any variation at all if starting subdivision with a one by one cell.

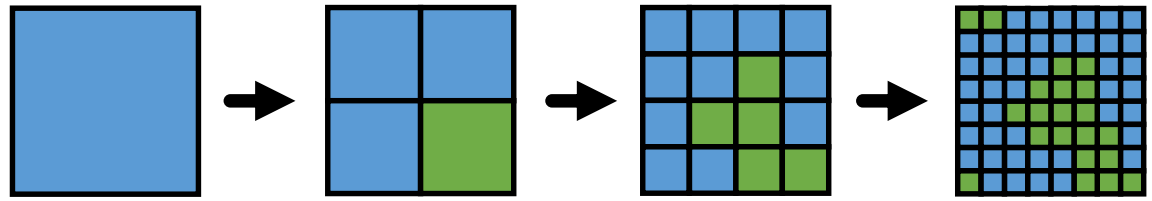
To increase variation randomly, a technique was tried out where, instead of offsetting the average height value by a random amount, there would be a random change of a flip in height i.e. a random chance of water becoming land and land becoming water for each new cell.

The random chance of this happening had to be set fairly low so that land changes were not too dramatic. Interestingly, a lower chance of water becoming land compared with land becoming water gave more attractive results where there was a good balance of water and land in the final 256x256 map. This might be because only a quarter of the height scale represented water. The following algorithm for calculating the value of a child node is as follows:

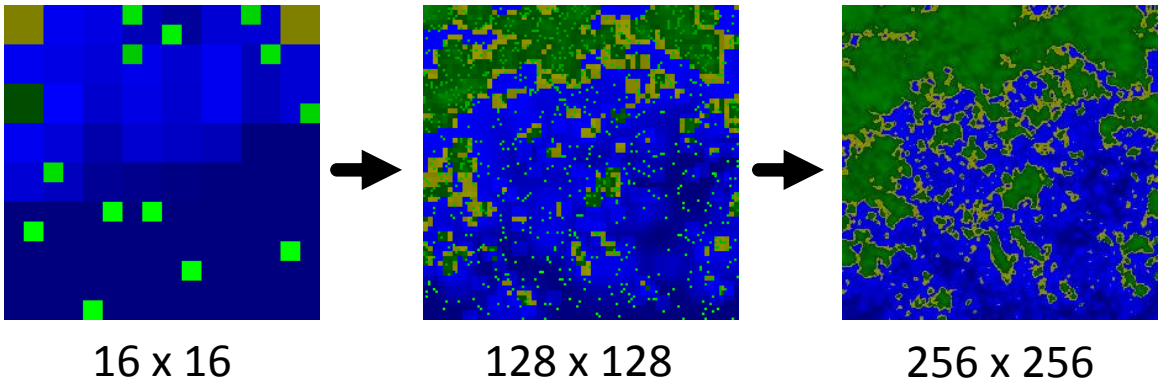
```
average ← sum(surrounding_nodes) / count(surrounding_nodes)
if average < 0.25 then
      if random(24) = 0 then
            return 1.0 – average
else
      if random(8) = 0 then
            return 1.0 – average
```

Figure 3.2 Pseudo-code for calculating the terrain height of a sub node.

The experiment was written in C# using LINQPad. Code was written to take a two dimensional array of height values and subdivide each one into four additional cells creating a new two dimensional array twice the size as the original. Figure 3.3a shows the step by step process of subdividing a 1x1 cell four times to give an 8x8 grid with a procedurally generated area of land and Figure 3.3b shows the real results generated by the application.



(a)



(b)

Figure 3.3 Subdivision of squares

From these results, it was shown that a low resolution grid initially began unvaried and unsmooth. But as the grid began to subdivide recursively, the detail began to appear with a very pleasing appearance. The 256x256 grid shows an obvious land mass with several detailed islands and peninsulas and overall a fractal like image demonstrating that this works very well with uniform squares.

3.3.2 Experiment B – WPF rendering of hexagons

The second experiment's goal was to render two dimensional hexagons neatly together with one another in a tessellated fashion. Windows Presentation Foundation is a graphical user interface library for .NET framework applications that uses DirectX to render shapes and controls. Windows 8 apps use a framework based on the WPF library; they both use XAML for storing the user interface and thus very similar in how they are used.

To render a hexagon, WPF provides a polygon shape class with properties such as a position, a collection of points, a fill colour, a stroke thickness and a stroke colour. A method was written to create an instance of a polygon which automatically calculates the appropriate points relative to the centre position of the polygon to form a flat top hexagon. The points were calculated using the fractions seen in Appendix 9.2 and the formula in Appendix 9.1 using an input size. The code for this can be seen in Figure 3.4.

```
Polygon GetHexagon(double hexSize, double cx, double cy) {
    double hexWidth = 2.0 * hexSize;
    double hexHeight = Math.Sqrt(3) * hexSize;

    Point[] hexPointOffsets = new Point[] {
        new Point(-0.25, -0.50),
        new Point(+0.25, -0.50),
        new Point(+0.50,  0.00),
        new Point(+0.25, +0.50),
        new Point(-0.25, +0.50),
        new Point(-0.50,  0.00)
    };
    ...
}
```

Figure 3.4 Hexagon offset calculation

Once this method was written, it simplified the task of rendering hexagons to calling this method with the centre position and size of the hexagon. The results can be seen in Figure 3.5.

Figure 3.5 WPF tesselated hexagons

3.3.3 Experiment C – 3D hexagons using OpenGL

Code to create a three dimensional landscape was developed using hexagons to see the visual impact and to investigate the feasibility of utilising this type of model in the application. A terrain was rendered using 3D hexagons by writing a C# application which utilised OpenTK, a C# wrapper around OpenGL.

A height map was loaded into a two dimensional array representing heights between 0 and 1024 which allowed a single value of zero to represent water and 1024 values to represent land height which are divided by four to fit the 256 possible shades of green. These heights would be used to position the three dimensional vertices which define the hexagonal polygons. The colours of these vertices were set to a shade of green based on the height value.

Hexagons were made from six triangular faces which were drawn by specifying three vertices generated using the two dimensional array of height values. The graphics card interpolates the colours of the vertices to give a varying face colour. This led to a smooth map with no visual evidence of hexagons. To remedy this, instead of setting each vertex to a colour based on that vertex's height, all vertices used to render a single hexagon were set to a colour based on the average height of all the vertex height values. This led to sloping hexagons with (solid flat) colours, as seen in Figure 3.6.

Going further, the application was modified so that not only was the hexagon's colour fixed for each hexagon, but the height too. As each hexagon is rendered, the vertices' height values were set to an average of all the vertex height values used in the hexagon being rendered to produce a flat base hexagon. This gave interesting results, however gaps between the hexagons could be seen where the height changed between neighbouring hexagons, so extra vertical faces were drawn around each cell to fill the gaps. This technique resulted in lots of hexagonal towers of different heights tessellated together, as seen in Figure 3.7. Surprisingly, this type of landscape formation occurs naturally - the Giant's Causeway in Northern Ireland being an example.

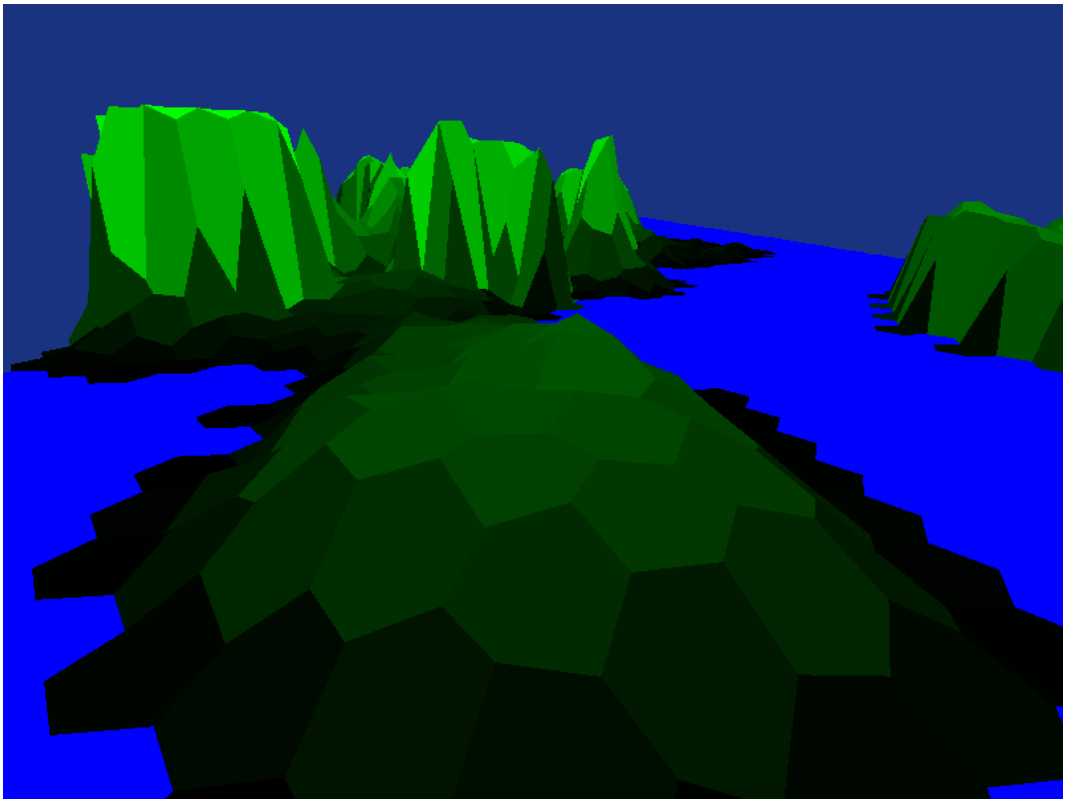


Figure 3.6 3D hexagons

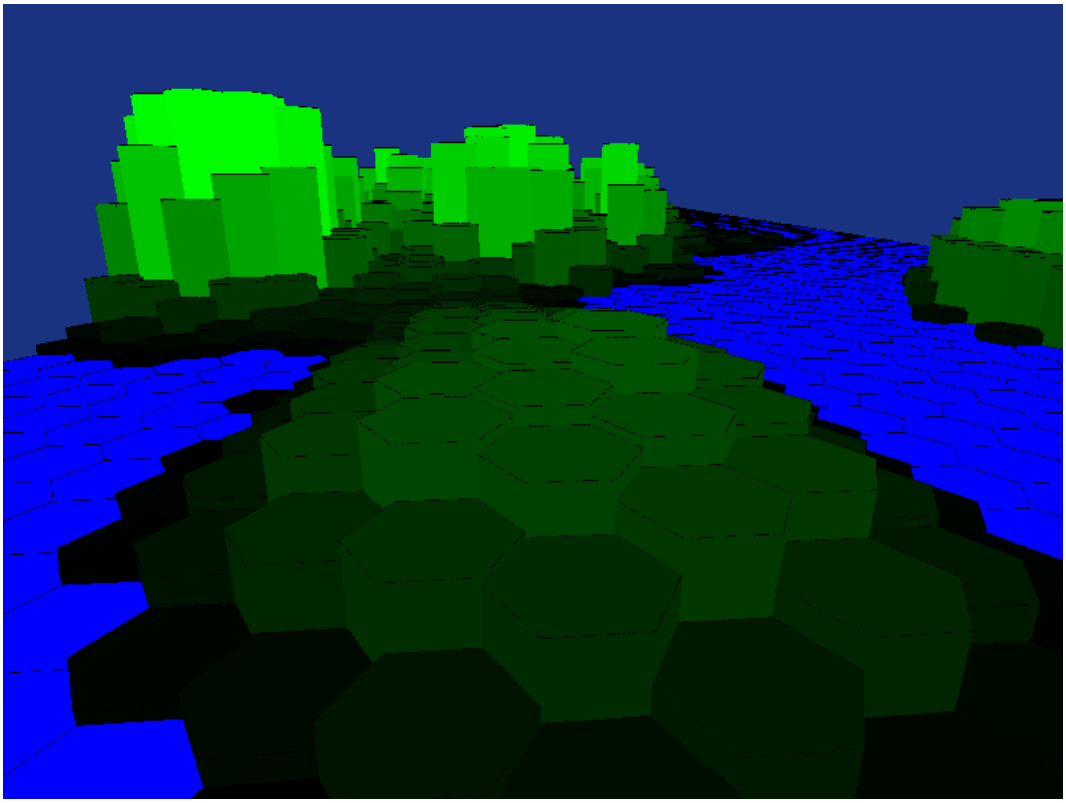


Figure 3.7 3D hexagon towers

3.4 Internal storage of the map

To store the map with the ability to subdivide one hexagon freely into children and zoom into them, a data structure such as a graph or tree can be used. Trees are stored by an initial reference to the root node (hexagon in this case) and references on that node to sub nodes (child hexagons). Another reference is used to point to the current hexagon that is showing, this allows zooming in by changing the view reference without needing to manipulate the map. Rendering is then optimised as it only needs to traverse a fixed number of levels from the view reference node.

Figure 9.3 shows a mock up drawing of a map represented as a tree showing how each node can have the view reference pointing to it. For the ability to zoom out, each node also needs a reference to its parent node unless the node is the root, in which case the reference would be null. Table 3.1 shows a simple plan for the Hexagon data structure and the properties that are stored.

Name	**Type**	**Description**
Parent	Hexagon (pointer)	A single pointer to the parent hexagon.
Children	Hexagon (pointer array)	A variable array of hexagon pointers to the node's children.
TerrainInfo	TerrainInfo	Sub data structure containing information such as height, moisture etc.

Table 3.1 Hexagon data structure

3.5 User interface

The user interface needed to be simple for touch screens. The official Microsoft guidelines [21] for Windows 8 apps were loosely adopted. Figure 9.1 shows mock concept drawings of the main screen and pattern selection screen. The main screen has various buttons which allow the user to generate hexagons, zoom in or out, change the size of hexagons or reset the map. There is also a button to link to the pattern selection screen where the user can change the pattern in which hexagons are subdivided. Should generating more detail or rendering the map take longer than a second, a label or image is shown in the top right of the screen to show if it is currently processing so that the user knows when the application is still performing a task.

The pattern selection screen allows the user to select how they want parent hexagons to split up into child hexagons. The page clearly shows how one hexagon will be subdivided into the pattern of smaller child hexagons for each possible pattern. When the user taps on a pattern, it will navigate back to the main screen, resetting the map in the process.

4 Implementation

As mentioned in section 1.2, the aim of the project was to produce an application that could procedurally generate a hexagonal map of terrain which the user could freely zoom in to and out of to reveal more or less detail respectively similar to a Mandelbrot set.

The challenges to overcome included; storing a seemingly infinite hexagonal map, procedurally generating random terrain on a hexagonal map and rendering a smooth zoom in of a hexagonal map that reveals more detail by generating it when needed.

4.1 Internal storage of the map

The map was stored as a tree where each node represents a hexagon. The whole map is subsequently represented using just one class which acts as a tree node named *Hexagon*. The class stores three pieces of information, terrain information, a pointer to its parent hexagon and a list of pointers to its children. The pointer to its parent hexagon helps with algorithms that require traversing up a tree when not starting at the tree's root and the pointers to the hexagon's children help with traversing down the tree, as shown in Figure 9.3.

Due to the tree structure nature of the map, the process of generating more sections of the map is simple and effective through the process of grafting. This however can get inefficient very quickly as the graph algorithms take longer to run as the tree becomes larger. Pruning would therefore need to be used to sever the oldest or highest segments of the tree and either remove it completely or store it temporarily until needed again.

The Hexagon class provided three dynamically calculated properties; *Siblings*, *Levels* and *Descendants*. These simplified more complicated algorithms by reducing the amount of code, as retrieving the siblings (for example) was separated into a different method. Siblings are retrieved by returning all the children of the hexagon's parent excluding itself. No hexagons are returned in the case of the pointer to the parent hexagon having a value of null. The number of levels is calculated by calling the calculate levels routine recursively on its children, then taking the maximum value and adding one. If the hexagon has no children, 0 is returned as the number of levels (the base case). Lastly, descendants will yield all the hexagon's children before recursively calling the descendants routine for its children.

4.2 Encapsulating a WPF polygon object

As experimented with in section 3.3.2, hexagons could be rendered using the WPF Polygon class where each point could be set to an offset relative to the position of the polygon. As each hexagon would require a polygon with the same defined points and a reference to the underlying hexagon structure in the map graph, a new class (named *HexagonShape*) was written to encapsulate the polygon with a simple interface for specifying the colour and to hide the point definitions.

The class inherited ContentControl which allowed it be used like any other WPF object and added to a Canvas at an arbitrary position.

The specialised user interface control had a property called *Hexagon* which was a reference to the Hexagon object in the map graph. The control was then able to use that reference and call the hexagon colour derivation method directly (see section 4.4) without any external code providing the colour the polygon should be.

The control was also useful on the pattern selection page. The page showed a list of buttons with the pattern of hexagons shown. These hexagons also used the control but with a fixed colour as opposed to a dynamic colour based on the control's terrain.

4.3 Hexagon rendering optimisation

The view of the map was rendered by creating an instance of a *HexagonShape*, for each hexagon that can currently be seen within the viewport. All these objects were added to a WPF Canvas object where they would be displayed in their associated positions.

Upon rendering more than 500 hexagons, the time taken to render was over 10 seconds downgrading the quality of the user experience. Even if the process was asynchronous from the user interface, the user would still see a long delay from zooming in to the new image being produced.

Various techniques were attempted to reduce the delay, but the most effective was reusing the same *HexagonShape*s so that there was no overhead in disposing the old objects and instantiating the new ones. There is a clear performance cost in how objects are removed and added to a canvas as this was considered to be the process that took the longest according to the performance testing tools used to isolate the problem. Instead of clearing all the *HexagonShape*s in the canvas, each one was marked as available. Each hexagon in the internal map graph was then traversed and an available *HexagonShape* would be modified to reflect the new hexagon and marking it again as not available. Should the number of available *HexagonShape*s run out, a new *HexagonShape* would be initialised. Once all the hexagons had been traversed, all hexagons still available would have their visibility flag set to hidden so that unused hexagons could not be seen.

4.4 Hexagon colour derivation

Each hexagon holds terrain information. This includes its height or altitude, a floating point value ranging between 0 and 1. Using just this value, a colour is derived. The process of determining the best algorithm for deriving the colour was completed by trial and error until the desired look of the map was generated. More information about this process and its results can be seen in section 6.2 and section 5.1, respectively.

The algorithm initially replicated the functionality used in experiment A (section 3.3.1) where, if the altitude of the hexagon was below 0.25 (water level), then a 'gradient' of blue would be derived, `RGB(0,0,64)` to `RGB(0,0,255)`. Otherwise a gradient of green `RGB(0,64,0)` to `RGB(0,255,0)`.

This was then extended to include sand and snow where yellow would be derived for altitudes ranging between 0.25 and 0.40 to represent sand and gradients of white for altitudes above 0.90 to represent snow.

Eventually the same rules were applied but with a simpler technique. A table was defined in the source code which mapped altitudes to RGB colours (inserted as Table 4.1). Then, when deriving a colour for a hexagon, the lower bound and upper bound entries in the table were chosen by their altitudes. A linear interpolation was then performed on the lower bound and upper bound's individual RGB components based on where the hexagon's altitude lay between the lower bound and upper bound table entry altitudes.

The calculation for linear interpolation used is seen below.

$$f(t) = a + t(b - a)$$

Where a and b are the lower and upper bound colour components respectively and t is $\frac{h-l}{u-l}$, where h is the altitude of the hexagon, l is the lower bound altitude and u is the upper bound altitude.

For example, given an altitude of 0.75, the entry in the table with the highest altitude less than 0.75 would be found and the entry with lowest altitude greater or equal than 0.75 would be found. In this case, altitude 0.5 with an RGB value of `(0,64,0)` and altitude 0.8 with an RGB value of `(0,255,0)` are the lower bound and upper bound entries respectively. Next the following calculations are performed for all three colour components, red, green and blue.

$$R = 0 + \frac{0.75 - 0.5}{0.8 - 0.5}(0 - 0) = 0$$
$$G = 64 + \frac{0.75 - 0.5}{0.8 - 0.5}(255 - 64) = 223$$
$$B = 0 + \frac{0.75 - 0.5}{0.8 - 0.5}(0 - 0) = 0$$

Altitude	Red	Green	Blue
0.0	0	0	64
0.2	0	0	255
0.3	64	64	0
0.4	128	128	0
0.5	0	64	0
0.8	0	255	0
0.9	192	240	192
1.0	255	255	255

Table 4.1 Altitude colour map table

4.5 Sub division of terrain

Using the same technique performed in section 3.3.1, the pseudo-code from Figure 3.2 was initially inserted for calculating the sub node height. However this gave an unexpected look where the overall shape of large hexagons was visible causing long diagonal lines where the terrain switches sharply from one value to another, as seen in Figure 4.1.

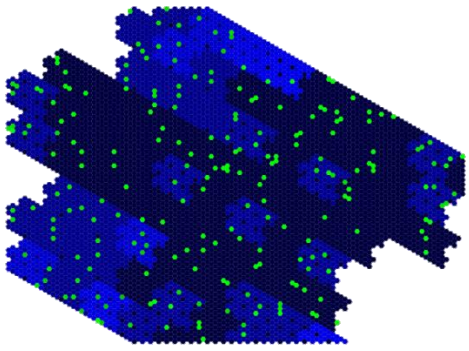


Figure 4.1 Initial look of procedurally generated hexagons.

5 Results

5.1 Colour derivation

Figure 5.1 shows four screenshots between consecutive changes made to the code that determines what colour a hexagon should appear as given its height. The implementation of this was discussed in section 4.4.

Screenshot A shows the original look. Screenshot B shows the look of the map after hexagons that lie between multiple parent hexagons are derived from those parent hexagons as opposed to its theoretical parent. This greatly reduced the artificial straight edges which occurred due to the directional nature of the hexagon pattern.

Screenshot C shows the look of the map after adding a colour to represent sand and snow. Sand is however too dominant and snow can be hardly seen.

Screenshot D shows the final look of the map after using a colour interpolation map.

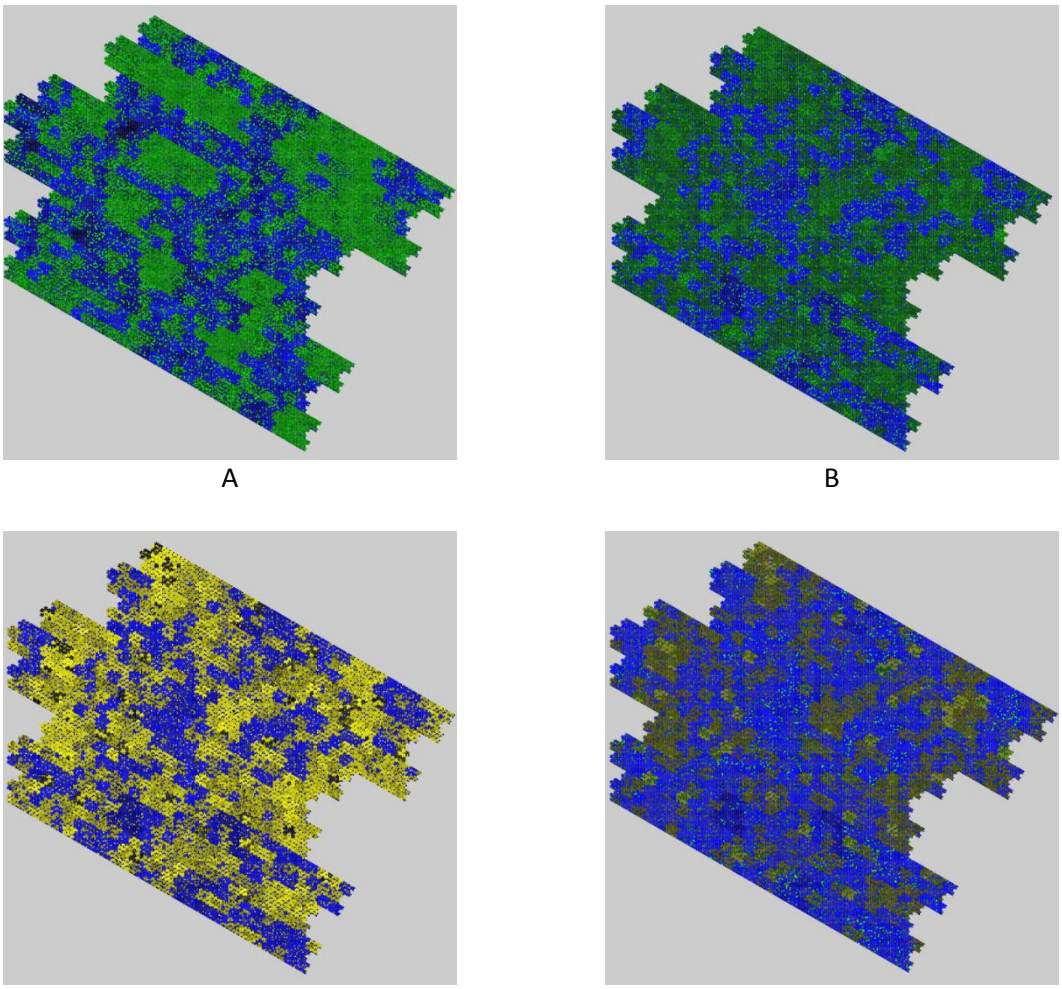


Figure 5.1 Screenshots before and after various changes.

5.2 Exported maps

Figure 5.2 show an early stage development map produced by the application and Figure 5.3 shows a map produced by the final version, with improved height calculation and colour derivation.

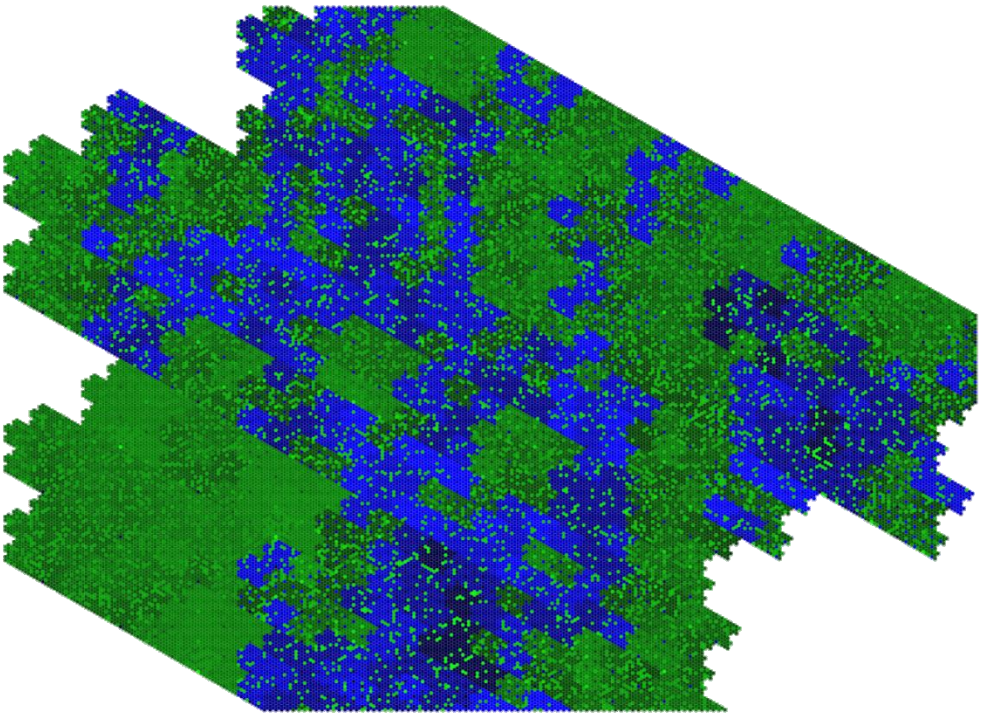


Figure 5.2 An exported map prior to improving the height calculation

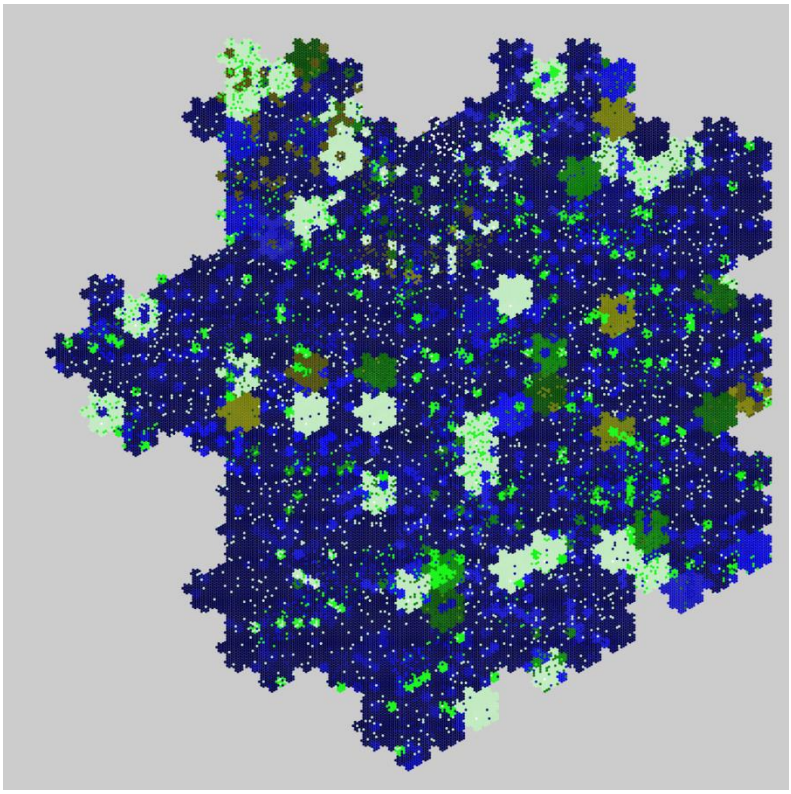


Figure 5.3 An exported map using the 13+3 pattern

5.3 Screenshots

Screenshots of the final version of the application which resemble the mock up drawings in Figure 9.1 are shown below.

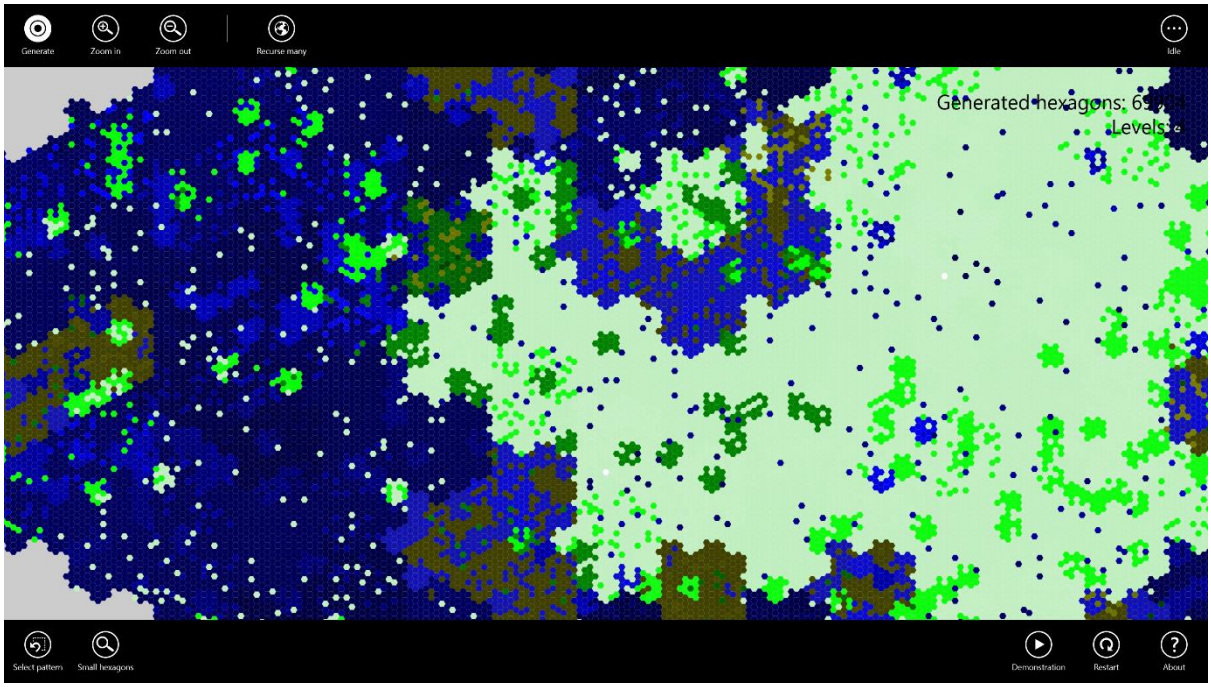


Figure 5.4 Screenshot of the main screen and a recursed map

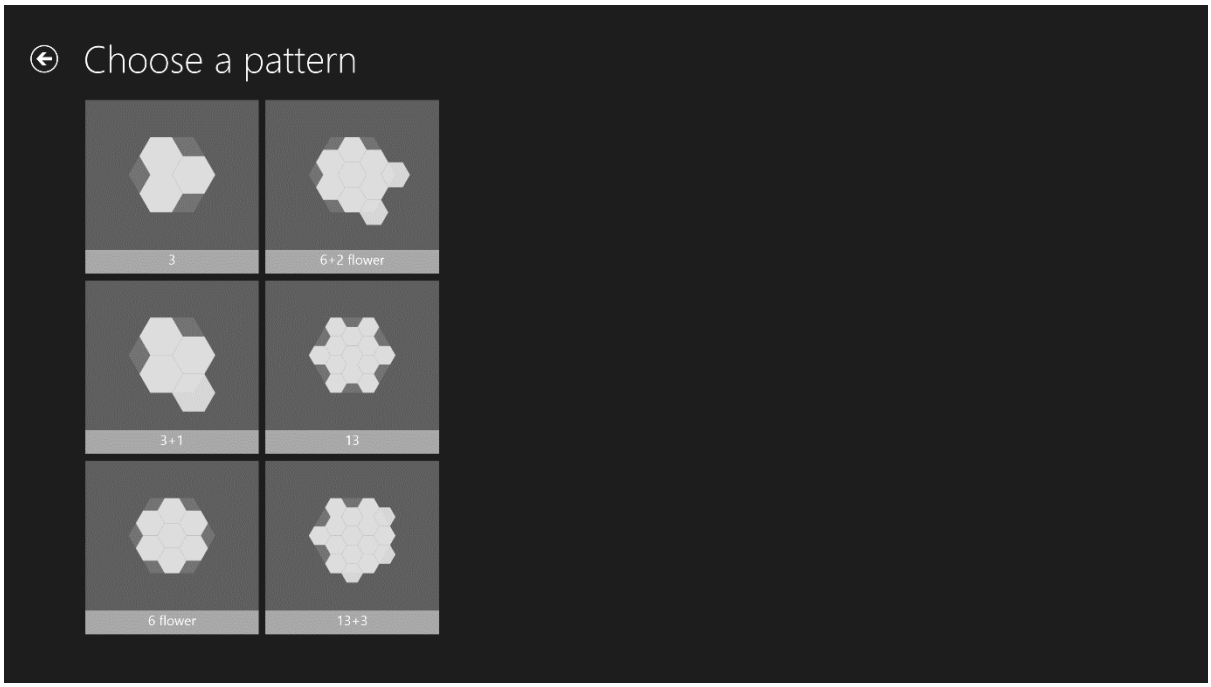


Figure 5.5 Screenshot of the pattern selection screen

6 Testing and Evaluation

Over the course of development, several testing methods were used. Some of these also evaluated the effectiveness of implemented functions to determine whether they needed improving or changing in any way.

6.1 Unit testing

The application was developed in a bottom up approach. This meant that a lot of code had to be written before it was ultimately used by, for example, the user interface. To test the code after it had been written but before the top layer of the application, unit tests were constructed which called the written function with test input data and assert the output data with the expected. Visual Studio provided a test workbench to do this easily and regressively and thus providing regression testing to check if further changes to the program break any features already implemented.

6.2 Trial and error

Some parts of the application such as the rendering of the hexagons could only by tested by manually examining the graphics produced. This then became a process of refining values and formulae until the desired look was achieved.

In each case, a screenshot would be taken prior to changing the code so that the look of the new version could be compared with the previous, see section 5.1. The change was also logged so that between each pair of screenshots, there is record of what was changed should the need to look back over the history of changes arise. It helps conclude what changes were more significant and produced the better graphics.

6.3 User testing

During the development of the application, the output and was shown to various people. Feedback was solicited about the appearance and opinions on whether it looked natural or not. Suggestions were also put forward as to possible ways the height and colour derivation algorithms could be improved.

When the user interface was complete, the application was given to several users for them to use. Users were able to quickly learn how to control the interface and recurse the maps to their own preference. Most users stated that the interface was intuitive and touch friendly. Some users however struggled to grasp the concept of the tools at the top of the screen without further explanation. It was therefore not clear that the tool buttons at the top of the main screen were toggle buttons and what their function was. This could be made clearer with either a tutorial at the start of the application or more descriptive text underneath the buttons. When users were questioned about the appearance of the rendered landscape, most thought the hexagons added a unique style to the terrain but did not like the colour contrast and suggested using darker colours for the hexagons.

Many users attempted to use pinch and pull gestures to zoom in and out. Sadly this was not implemented due to time constraints, had it been then it would the users would not have any trouble using the feature due to many other applications having similar interfaces such as map apps.

7 Conclusions

The final software that was developed was a Windows 8 Application written in C# and XAML. The application successfully demonstrated the concept of hexagonal terrain generation despite the drawbacks and problems that occurred during the development phase.

The project began with ideas and rough sketches of hexagons subdividing and researching how procedural generation works and the implementations that have been used. The idea was to bring together recursive and infinite fractal landscape generation and apply it to tessellated hexagonal cells to prove whether hexagons provide a different or otherwise more realistic artificial world due to the frequent natural occurrences of hexagons in real life.

7.1 Achievements

The experiments written and discussed in section 3.3 were beneficial to succeeding in completing the objective. They allowed the large problem to be split up into its constituent parts and technologies. Procedural generation and hexagon rendering could be separately worked on and allowed quick prototyping with minimal implementation issues. There was time to work on an experiment that was slightly beyond the scope of the project, the 3D hexagon rendering was an interesting idea that could spawn future investigations in that area.

The final application produced demonstrated multiple hexagon pattern subdividing and ultimately showed that the more hexagons that a single hexagon divided into produced a more varied and less artificial look to the map. Many small changes were made alongside so improve the output as well such as intelligently using different parent neighbours depending on the child hexagon's location.

7.2 Drawbacks

Rendering time was a significant drawback in the development phase of this project. A lot of time was spent attempting to optimise the rendering algorithm instead of concentrating on designing more elaborate generation and sub-division algorithms. It was however necessary to improve the rendering speed by improved object handling as it was vital when a lot of the testing was trial and error.

The framework used to build Windows 8 applications proved more complicated and time consuming to get the hang of than anticipated. Much time was spent looking through the framework manual, Q&A websites and forums to develop some parts of the system including the user interface and optimisation of the rendering algorithm. More time may have been saved if a more low level API such as OpenGL was used, where the library is simpler and more help is available.

7.3 Further work

Due to time constrains and the drawbacks, more terrain features such as biomes, rivers and moisture were not implemented. Implementing these could allow the generated maps to be much more interesting whilst adding complexity to the infinite generation of a section of the map. Mandlebrot discusses implementing rivers on hexagonal lattices using hexagonal subdivision where rivers are lines between two points in a hexagon [22]. A similar example of this can be seen in the article, Polygonal Map Generation for Games [10], where rivers are calculated using the elevation and moisture to determine the direction of flow along the edges of polygons. This would add extra rendering details

and thus now create a map not entirely made from hexagons. Whether this is a good idea or not and how zooming in would work is yet to be determined. The project therefore did not fully prove the advantages to procedural generation with hexagons.

With the used of hardware accelerated graphics and dedicated shader programs, the map could be rendered in real-time so that more detailed maps could be previewed quickly to see the effect when the hexagon subdivision settings are altered.

Instead of solid colours per hexagon, gradients or textures could be used to make the map appear more realistic.

8 References

[1] R. L. Devaney, “Unveiling the Mandelbrot set,” Plus, 1 September 2006. [Online]. Available: http://plus.maths.org/content/unveiling-mandelbrot-set.

[2] F. K. Musgrave, “Methods for Realistic Landscape Imaging,” pp. 250-251, 1993.

[3] T. C. Hales, “The Honeycomb Conjecture,” *Discrete & Computational Geometry,* vol. 25, no. 1, pp. 1-22, 2001.

[4] Fiskörn network, “Mandelbrot Wallpapers,” [Online]. Available: http://fiskorn.se/mandelbrot_wallpapers/.

[5] K. Perlin, “Improving Noise,” *ACM Trans. Graph.,* vol. 21, no. 3, pp. 681-682, 2002.

[6] “Perlin Noise,” [Online]. Available: http://freespace.virgin.net/hugo.elias/models/m_perlin.htm.

[7] “Explanation of Fraction Brownian Motion,” 2010 December 16. [Online]. Available: https://code.google.com/p/fractalterraingeneration/wiki/Fractional_Brownian_Motion. [Accessed 16 April 2014].

[8] “Perlin noise,” Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/Perlin_noise. [Accessed 16 April 2014].

[9] B. B. Mandelbrot, “Fractal landscapes without creases and with rivers,” in *The Science of Fractal Images*, New York, Springer-Verlag New York, Inc., 1988, pp. 251-252.

[10] A. Patel, “Polygonal Map Generation for Games,” Red Blob Games, 4 September 2010. [Online]. Available: http://www-cs-students.stanford.edu/~amitp/game-programming/polygon-map-generation.

[11] Microsoft, “Graphics APIs in Windows,” [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/ee417756(v=vs.85).aspx.

[12] Microsoft, “MSDN Windows 8 App Development,” [Online]. Available: http://msdn.microsoft.com/en-us/windows/apps.

[13] Microsoft, “Official .NET Framework website,” [Online]. Available: http://www.microsoft.com/net.

[14] Microsoft, “Official Microsoft Visual Studio website,” [Online]. Available: http://www.microsoft.com/visualstudio.

[15] “Official LINQPad website,” [Online]. Available: http://www.linqpad.net.

[16] “Official Git website,” [Online]. Available: http://git-scm.com.

[17] “GitHub,” [Online]. Available: https://github.com.

[18] “Official GitHub for Windows website,” [Online]. Available: http://windows.github.com.

[19] P. Måhlén, “Code Sharing: Use Git,” 28 March 2010. [Online]. Available: http://pettermahlen.com/2010/03/28/code-sharing-use-git/.

[20] Microsoft, “Introduction to WPF,” Microsoft, [Online]. Available: http://msdn.microsoft.com/en-us/library/aa970268(v=vs.110).aspx.

[21] Microsoft, “Index of UX guidelines for Windows Runtime apps,” [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/apps/hh465424.aspx.

[22] B. B. Mandelbrot, “Fractal Landscapes Without Creases and with Rivers,” in *Fractal landscapes without creases and with rivers*, New York, Springer-Verlag New York, Inc., 1988, pp. 258-260.

[23] A. Patel, “Hexagonal Grids,” Red Blob Games, March 2013. [Online]. Available: http://www.redblobgames.com/grids/hexagons.

9 Appendix

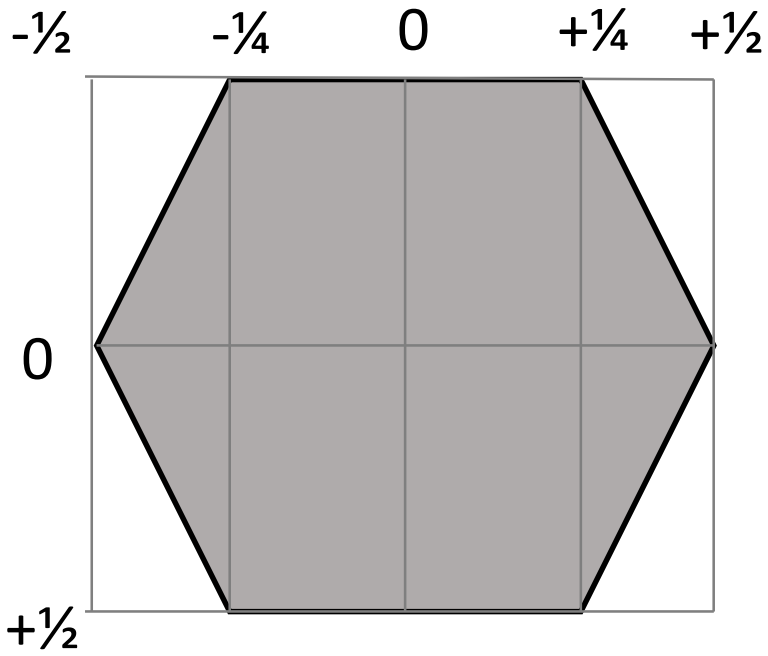
9.1 Hexagon size calculation

$$\text{Width} = 2 \cdot \text{size}$$
$$\text{Height} = \sqrt{3} \cdot \text{size}$$

As provided by Hexagonal Grids [23].

9.2 Hexagon proportional offsets

To calculate the offsets relative to the centre of the hexagon the width and height in Section 9.1 are multiplied by the fractions below [23].



9.3 Design drawings

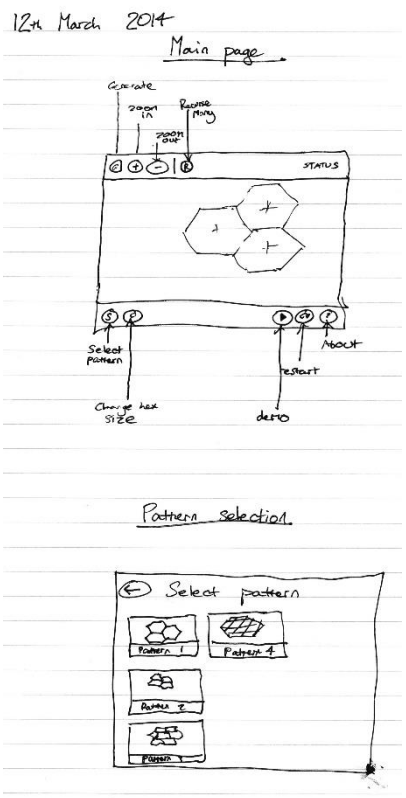


Figure 9.1 UI Concept

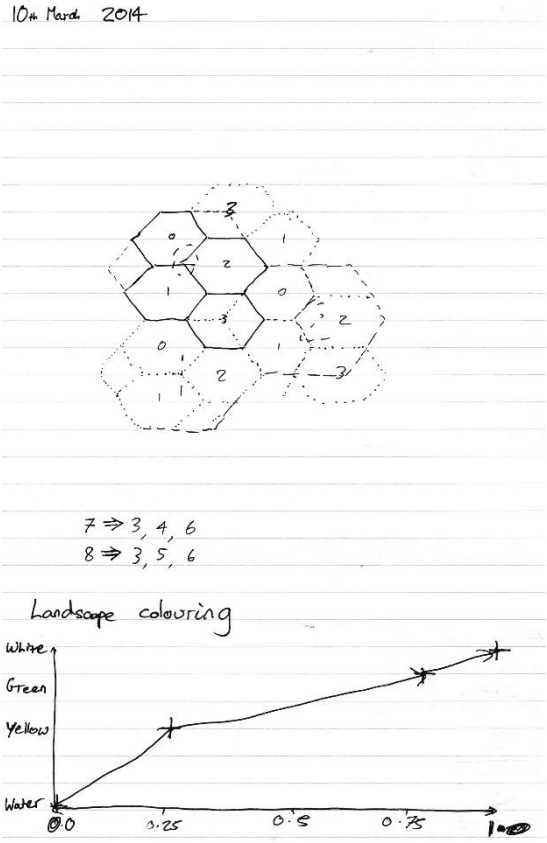


Figure 9.2 Landscape colouring

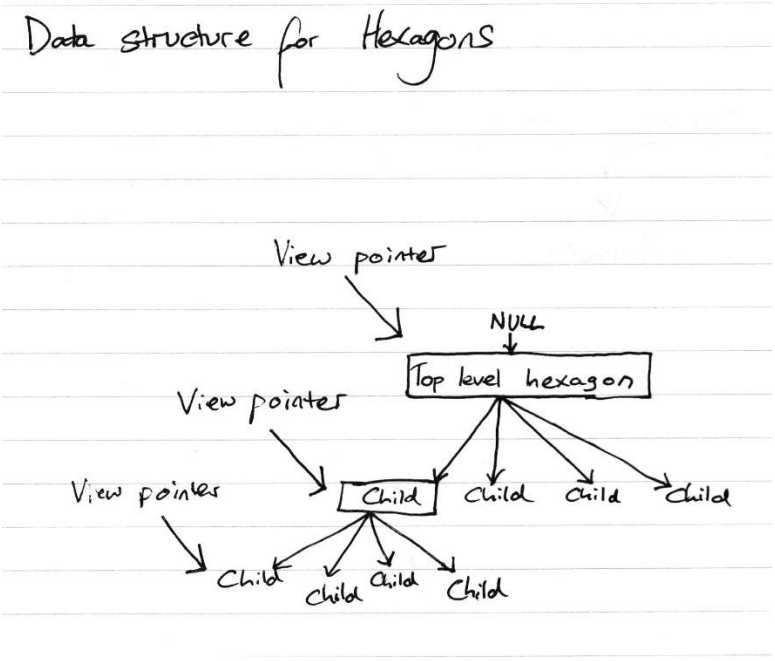


Figure 9.3 Hexagon tree