Mathew Carr

JMU Registration ID: 261177

CMPGN3007

Computer Games Technology

School of Computing and Mathematical Sciences

B.Sc Project

(2008-2009)

Supervisor: Sud Sudirman

A STUDY OF PROCEDURALLY GENERATED REPRODUCIBLE ENVIRONMENTS

Abstract

The production of the content necessary for computer games is an expensive and longwinded process. Various 'procedural content generation' tools exist to provide a large degree of automation to what would otherwise be a time-consuming manual process.

This study identifies a specific, common problem in content generation: the generation of an 'island' environment, and conducts a thorough survey into the techniques and tools available to assist.

The research presented is then used to formulate a flexible data processing model which can be applied to produce a solution to the stated problem. The model discussed is implemented in an interactive software prototype (attached).

Keyword List

Procedural content, 3D, terrain, environment mesh, height map, fractals, fractal subdivision, noise.

CONTENTS

Chapter 1: INTRODUCTION

Background

Graphical computer games of any complexity require some amount of game assets to allow the simulated scenario to be visualised and presented to the user. These game assets take many forms: bitmap images, model files, shader definitions, audio files, game scripting files and aggregated combinations of these which may include metadata and other aggregation information linking related assets.

As an example, a single 3D 'level' in a modern game may consist of any number of environment meshes, textures and shaders, collision data, event scripting, and references to additional objects with their own externally specified assets. Simpler games with minimal graphics still have the same requirements: a rudimentary version of Pong will still require graphics for the playing field, ball, bats and score numbers.

Creating the content necessary to produce a game is a costly and complex process, which can involve many different people at many stages of production. This is traditionally a manual (albeit computer aided) process: textures must be drawn by artists, sounds created by sound engineers, and level objects and events placed by level designers.

One possible solution for the task of content creation is the application of 'procedural generation' techniques. 'Procedural generation' is a loosely defined term; it can be used as a description of almost any measure of programmatically assisted content generation:

It can be used to describe the process whereby an author can produce a well-defined environment, texture or other piece of content by constructing the content from (or augmenting an existing piece of content using) a stochastic model [4] : a series of configurable mathematical abstractions. These are then evaluated by the game engine at run-time to produce a complete piece of content, effectively using a series of mathematical functions in the place of a large dataset of explicit values.

The use of this process has two advantages: it gives the author the ability to specify content outside the context of a set level of detail. This means the game content can be realised by the software in any desired level of detail upon demand. It also allows for complex pieces of content to be represented using a smaller set of data than would be used to explicitly specify the content. These advantages incur a cost in cycles and time due to evaluating the content at run-time.

This process has been used in the games Just Cause and Darwinia to simulate large, detailed island archipelago environments.

Procedural generation also applies to the process of generating a piece of content based on a series of generalised parameters. For example, Sim City 2000 and Transport Tycoon Deluxe allow the player to specify the nature of the environment they would like to play within by manipulating a number of variables describing the amount of starting forest, severity of elevation, presence of rivers, etc.. These games generate different environments each time they are asked to, even if the input parameters remain

constant. With this system, the games can provide a seemingly limitless supply of environments to challenge the player.

Through the combination of different procedural generation methods, procedural generation can be used to automate the generation and placement of a specific class of object within an existing environment, as performed by the tree generation software SpeedTree, or it can be used to generate the entirety of a game environment, as is done within the games Elite and Frontier: Elite 2 to create, position and name the multitude of in-game galaxies which the player can reach.

If a procedural content generation algorithm is appropriately designed, it will have the ability to exactly reproduce a previously generated piece of generated content when called upon to do so. The galaxies within the game Elite are procedurally generated by using data retrieved from a pseudo-random number generator as input to galaxy creation routines. The result of this is the specification of an incredibly large, random-seeming, yet completely reproducible game environment specified through the application of strict rules.

This project aims to explore how methods such as these can be used to create reproducible environments for interactive software in a specific genre context.

Chapter 2: PROJECT AIMS

Overview

The objective of this project is to provide detailed insight into how the use of procedural content generation (PCG) techniques may be used as effective tools within the computer game asset production workflow.

The following two aims are defined:

- Conduct an in-depth study of the tools and techniques available to generate reproducible procedurally generated 3D landscapes.

- Conduct an in-depth study into how the identified PCG tools and techniques may be applied to create and present a complete 3D environment within criteria representative of real-world computer game requirements.

Quantitative comparisons will be made between the use of these techniques and the manual preparation of content in terms of storage requirements (hard drive required to specify the terrain), retrieval requirements (CPU and memory resources needed to bring the terrain to readiness from it being stored on disc) and memory requirements once loaded into memory.

Study of the Tools and Techniques Available for PCG in 3D Landscapes

I will identify a number of tools and techniques for generating curved, perturbed or otherwise detailed surfaces which may be adapted for use in the generation of 3D landscapes.

The procedural synthesis of 3D landscapes (also known in literature as ‘surface maps’ or ‘topological meshes’) has been the subject of research since the origins of computer graphics and CAD. Much research exists covering the subject, suggesting a number of methods based upon the use of noise and emerging fractals through recursive subdivision.

This study will focus on identifying and analysing methods suitable for the generation of a 3D world consisting of a detailed island environment, for use in a computer game context. Although simple, this scenario occurs frequently across many different genres of computer games and will allow for the exploration of the many different PCG techniques available for the problems.

The techniques identified will be evaluated based on the following criteria:

- The applicability and adaptability of the algorithm to the specified task.

    - A given algorithm may excel at producing one type of landscape but not another. For example: an algorithm may be able to create complex mountain ranges when given a completely flat starting mesh, but it would not be able to readily create useful islands without modification or

adaptation. It is expected that final demonstrative piece of software will use a mixture of related algorithms to achieve the desired effect.

- The range of detail that may be extracted from the algorithm.
    - It is expected that the majority of algorithms will be able to provide 'limitless' detail, if time complexity is to be ignored. However, it is possible that a given algorithm will have additional intrinsic constraints that prevent its use beyond a certain point.
- The relationship between the level of detail sought from the algorithm and the resulting time complexity.
    - It is expected that, for most algorithms, the time complexity will rise rapidly with increasing detail. Most algorithms will become unusable beyond some level of detail.
- The ability of the algorithm to exactly reproduce the same landscape under multiple initialisations with the same seed parameters.
    - It is expected that all algorithms that primarily rely on pseudorandom number generation will be able to repeatedly exactly reproduce the same landscape if the pseudorandom number generator is seeded and used correctly. This will have to be handled appropriately in the design of the generation software.

Application of Identified Techniques Towards a Realistic Scenario

Having identified the tools and techniques available, I will propose a model which may be adapted for use in generating detailed reproducible 3D terrain surfaces.

This will be done within the context of a specific software problem, allowing for a focused discussion of the problem, and how the different methods can be used in combination to produce the appropriate result.

The final model will be implemented as a software prototype and evaluated on its ability to provide realistic environments. A full description on how the terrain generation algorithm and the generated environments will be assessed is included as part of the Software Specification.

Chapter 3: STUDY OF EXISTING RESEARCH

Procedural Content Generation in 3D Landscapes

The procedural synthesis of 3D landscapes has been the subject of research since the origins of computer graphics and CAD. Much research exists covering the subject of procedural generation and stochastic modelling, suggesting a number of methods based upon the use of noise and emerging fractals through recursive subdivision.

For storing 3D terrains, a distinction is made between 'meshes', groups of polygons holding the full surface of a terrain, and 'height maps', 2D matrices of real numbers holding the vertical displacement from a defined value of a series of regularly spaced points on a plane. Meshes can be used to directly represent terrain that has holes or geometry directly above other geometry. Height maps can only be used to represent continuous terrain that does not overlap. It is possible to easily change a height map into a mesh (and indeed this is almost always necessary to render the terrain represented by the height map), but the opposite is not always possible.

Mandelbrot's 1975 paper, *"Stochastic Models for the Earth's Relief, the Shape and the Fractal Dimension of the Coastline, and the Number-Area Rule for Islands"* [1] , on the use of stochastic models for modelling the terrain and coastlines of the Earth provides the basis for almost all research into the use of noise and fractional Brownian methods for terrain modelling. It extends previous studies by Lévy [3] and Mandelbrot [2] into the self-similarity of naturally occurring land formations and their modelling through 'fractal' [2] models.

In the work, Mandelbrot performs a quantitative comparison between recorded real world terrain and coastline values and experimental models based upon fractional Brownian motion. A number of hypotheses are specified, exploring the self-similarity of coastlines and the relationship between hierarchical island groups, including a condition that surfaces and coastlines produced through mathematical simulation should be judged on their resemblance in physical appearance as well as on a quantitative mathematical basis, a viewpoint identified by Mandelbrot as a potentially controversial due to a lack of objectivity in the method.

The paper concludes that little distinction can be made between 'noise' and 'signal' in these models, with the further implication that a well-designed and well-defined mathematical model of simulated noise can be used for the generation of a useful, detailed set of values for the synthesis and simulation of terrains. The paper provides a number of visualisations depicting fractional Brownian surfaces produced by modulating flat terrain against values obtained by Poisson Shot Noise. (See Figure 1)

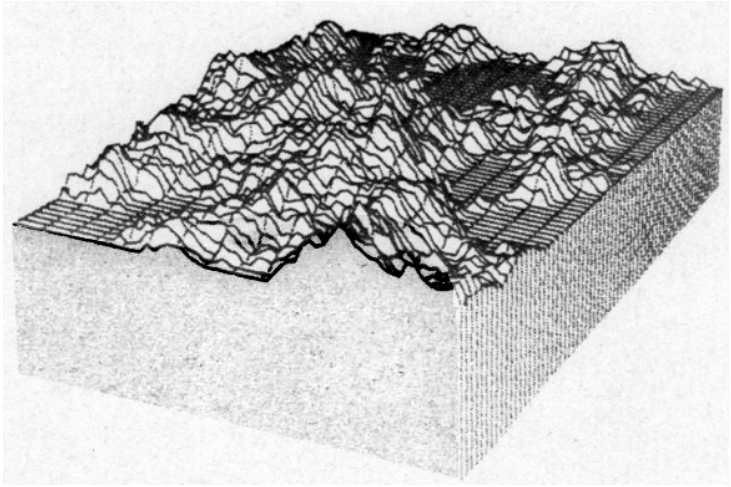

Figure 1: Perspective view of a sample of a Brownian surface of Paul Lévy (From Mandelbrot's paper, with colours altered for increased contrast upon printing)

This idea is further explored by Fournier, Fussell and Carpenter in the 1982 paper *"Computer Rendering of Stochastic Models"* [4] . In this work, the authors expand on ideas previously explored by Mandelbrot and define a new kind of modelling primitive, the stochastic model.

They define a stochastic model of an object as 'a model where the object is represented by a sample path (a realization) of some stochastic process of one or more variables'. Stochastic objects are comprised of stochastic primitives, a superset of traditional deterministic primitives such as polygons and parametric patches.

Stochastic models, when used appropriately, have a number of advantages including:

- It allows for the specification of content outside the context of a set level of detail. This means content can be realised by the software in any desired level of detail upon demand. Furthermore, the level of detail can be increased or decreased at any time without 'running out' of detail to display at the required resolution.

- It also allows for complex pieces of content to be represented using a smaller set of data than would be used to explicitly specify the content.

These advantages incur a cost in cycles and time due to evaluating the content at run-time, and are only apparent when the object or phenomenon to be modelled exhibits stochastic behaviour. The paper covers the concept of 'internal consistency', a property whereby successive realisations of a given stochastic model should retain the same general character as the detail increases. This will be an issue when the noise function uses values provided by a pseudorandom number generator: depending on the order of values called from the generator, the resulting surface will be wildly different if the care is not taken to ensure the random seeds used in the calculation of a set of features remain constant. The solution to this problem is to seed the random number generator with values defined based upon identifiers of points on the surface.

The paper discusses a number of possible implementations of realisations of stochastic models based on fractional Brownian motion, including an in-depth study of a method using recursive polygon subdivision for the generation of reproducible, complex 3D terrain. It is upon this basis that the environments in this study will be constructed.

The paper describes the following method for realising 3D stochastic parametric patches suitable for use as terrain:

- Generate the border of the realisation surface using recursive fractal polyline division.
- Calculate the centre of the quadrilateral as a Gaussian pseudo-random variable whose expected value is the mean of the positions of the four corner points and whose standard deviation is c^{-lH} where l is the level of recursion, H is the self-similarity parameter and c is an application-dependent constant. (See the calculation of 1a in Figure 2)
- Calculate the centre of the new quadrilateral at this level using the new horizontal and vertical points as the neighbours. (Calculation of 1b)
- Repeat this process from the second step, within the newly defined square. (Calculations of 2a and 2b in Figure 3)

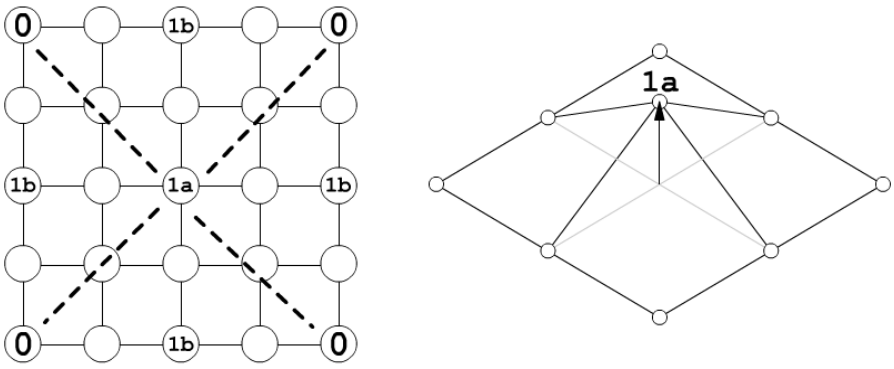

Figure 2: Quadrilateral polygon subdivision, 'Square' step, level 1.

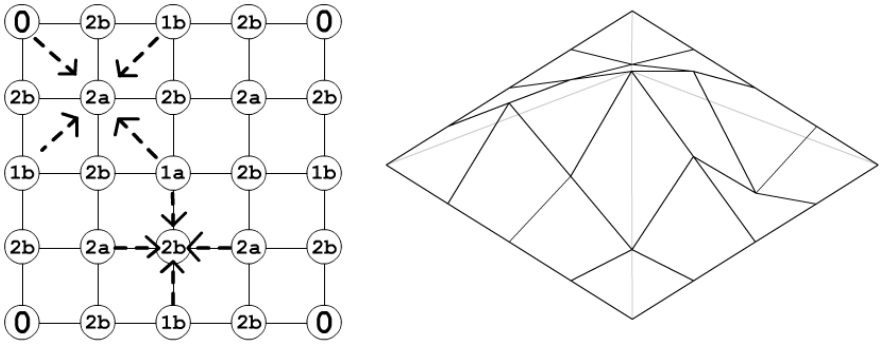

Figure 3: Quadrilateral polygon subdivision, 'Square' and 'Diamond' steps, level 2.

As the level of recursion increases, more detail is added to the surface while the decreasing c^{-lH} term attenuates the range of values the displacement variable can take. The H controls the rate at which the term attenuates with respect to the level of recursion: a H value of zero will cause the term to remain constant, greater values of H will cause the value to decrease with increasing recursion.

The authors propose that the points obtained by this method be used as control points in a bicubic Bezier patch, but also add that this method of interpolation results in a loss of distinct character for intentionally rugged surfaces. It is sometimes sufficient to simply use realise the surface as a large number of quadrilaterals at a sufficiently high level of recursion. The exact method chosen to realise and render the surface will depend on the intended use for the generated environment.

The authors of this paper note that the surface produced by this method cannot be strictly classed as a fractional Brownian surface, however it is reasonably similar. They further state that any realisation of a stochastic model can only be an approximation of the exact surface defined by the parameters of the model, and, like Mandelbrot, subsequently choose to evaluate the success of their realisation algorithms on subjective visual acceptability rather than on a mathematical empirical basis. Stochastic subdivision was revisited in 1987 by Lewis [5] .

Miller analyses a number of 'database amplification' methods using fractal polygon subdivision in his 1986 paper, *"The Definition and Rendering of Terrain Maps"* [7] . 'Database amplification' is a process where stochastic methods are used to interpolate between values from a sparse input database of known values. In the context of terrain generation using stochastic models, database amplification is the process used to generate complex terrain around a number of known fixed points. It is also possible for the 'known' values to be generated stochastically; the major features and general character of the terrain can be influenced beforehand by 'fixing' specific points and generating the terrain around them. For example, the diamond – square interpolation based terrain generation algorithm discussed in [4] may produce any form of complex terrain given a flat horizontal plane, but it can be 'forced' to create complex terrain surrounding a central mountainous peak if the central point in the plane is fixed at a greatly elevated position.

Miller discusses how triangle interpolation and diamond – square interpolation react when used against a completely flat, square input surface with a single raised point in the centre of the plane. The paper reports that both these method exhibit artefacts known as 'creasing', slope discontinuities along boundaries, due to the way the interpolation is effected. A novel method, square – square interpolation, is suggested. This method removes the effects of creasing, with the added effect of a reduction in 'harshness' in the generated landscape. This was demonstrated by fixing the central point in the plane as mentioned above.

Martz' online tutorial / commentary piece [8] , which provided the inspiration for this study, provides an excellent simple explanation of the use of diamond – square recursive fractal stochastic modelling for terrain generation and responds to a number

of the issues raised in Miller's paper, namely those relating to the inherent creasing resulting from the use of the diamond – square algorithm. Martz' suggests the creasing can be mitigated if multiple fixed points are selected, instead of just one.

A number of studies have been conducted into how fractal terrain generation methods can be adapted to meet specific criteria, such exhibiting a specific shape or passing through predetermined control points. A number of methods have been proposed to guide the generation process such as the use of predetermined spline meshes [11] , Gibbs samplers [12] , iterated shuffle transformations [13] and natural erosion / deposition simulations [14] . If a height map is treated as a 2D greyscale image, a number of techniques derived from those used in the field of image processing can also be applied, such as convolutions. Many of these methods are summarised by Stachniak and Stuerzlinger in [10] , together with a novel method for applying constraints to a generated terrain, minimizing the difference between the generated terrain and an idealised prototype.

The algorithm seeks to place a number of nodes in the terrain space affecting the altitude of the surrounding terrain by values following a Gaussian kernel. The authors note that searching for these values is a costly process in terms of processing, sometimes taking several hours to produce a satisfactory set of modification nodes. This would obviously be an unsatisfactory delay in a computer game scenario if it occurred at the start of every level. However, it would be acceptable if the necessary modification nodes were generated during the production stages of the game and combined with the fractal parameters used in the generation of the terrain to create a multiple stage stochastic model. This composite model would then be realised and adjusted by the game software during run-time without the need to regenerate the nodes.

A disadvantage of this method would be that the game would be limited to generating only the terrains that were compiled in advance; it would not have the ability to create random terrains unless a method to generate the necessary modification nodes were implemented. It may be possible to reuse the modification nodes and change the pseudorandom seed, but there is no guarantee that the resulting terrain would satisfactorily meet the same conditions imposed on the original terrain.

Coherent noise space functions such as Perlin's *Noise* [16] and its successor *Simplex Noise* [17] [18] can be used as a basis for terrain generation if the output from the function is treated as a set of height map values. Both Noise and Simplex Noise have the following characteristics:

- Passing in the same input value will always return the same output value.
- A small change in the input value will produce a small change in the output value.
- A large change in the input value will produce a random change in the output value.

- Statistical invariance under rotation: it has the same statistical character regardless of rotation.
- Its features occupy a narrow band of frequency: its features all lie within a small, specific size range.
- Statistical invariance under translation: it has the same statistical character regardless of position.

These features allow Noise to be easily controlled; features of any character and size can be constructed by adding multiple layers of Noise at different frequencies (commonly called 'octaves').

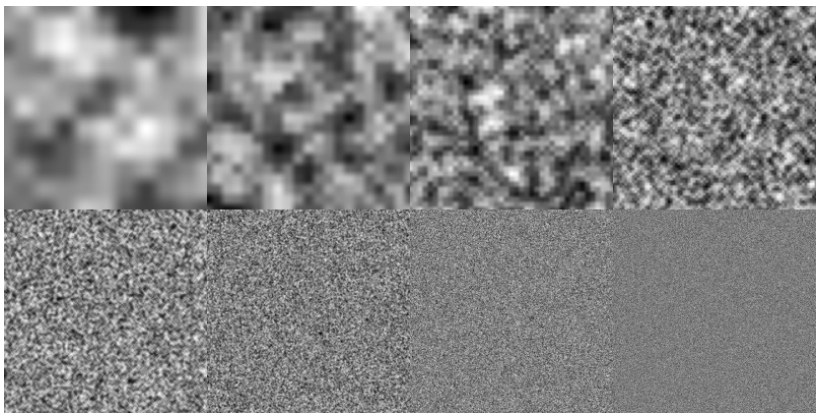

Figure 4: Increasing octaves of Perlin Noise (Image by Davide Coppola / m3xbox.com)

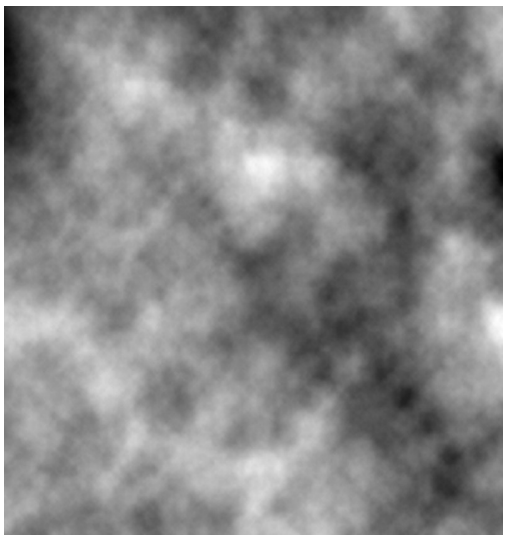

Figure 5: Complex composite texture constructed from multiple octaves of Noise (Image by Davide Coppola / m3xbox.com)

The controllable nature of Noise can be used to augment terrain height maps and meshes by jittering generated values to add extra interest. Furthermore, after the terrain has been generated, Noise functions can be applied in the rendering phase in geometry,

vertex and fragment (pixel) shaders to add extra visual interest to otherwise uninteresting geometry. As Noise is implemented in hardware on some graphics accelerators, a well-designed application can send large amounts of relatively sparse geometry to the rendering pipeline to be later enhanced with Noise with little speed penalty.

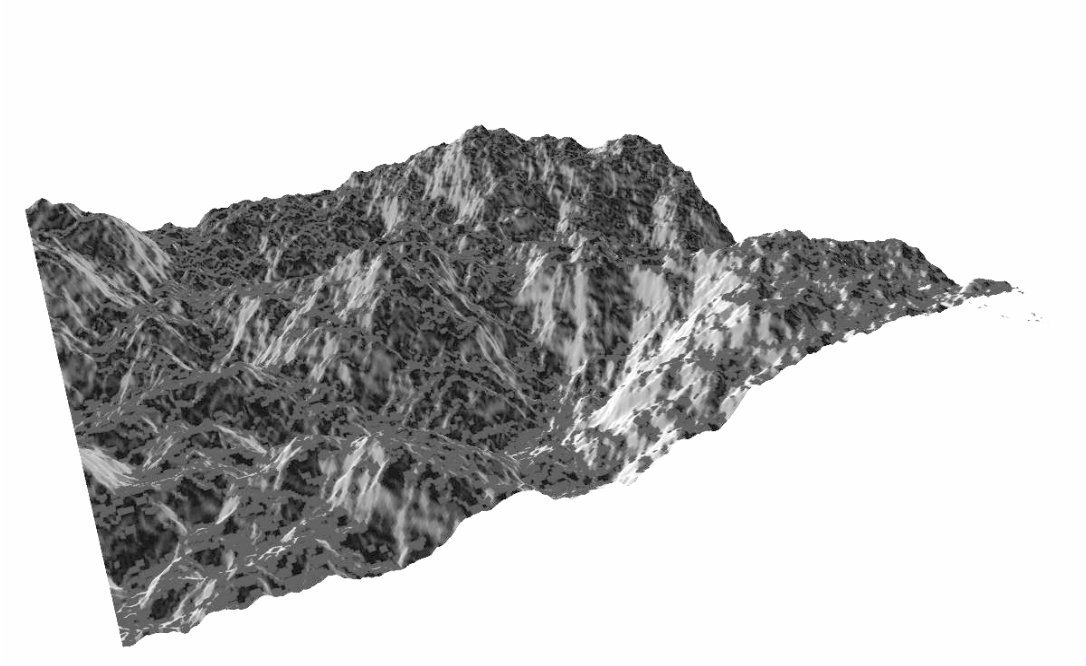

Figure 6: Terrain mesh created using a Perlin noise based height map. (Image by Stevo-88. Released into the public domain and available at Wikimedia Commons. Image adjusted for increased contrast.)

Coherent noise functions can be combined and modulated in many ways to produce almost any texture or effect map. The libnoise website has a detailed example where over one hundred coherent noise functions are used to create a complex map representing a detailed planetary surface [19] . The authors note that this method is intended for offline processing only: the map generated in the planetary surface example took 25 minutes to generate.

Combining multiple techniques is a common tool in procedural content generation. .werkkzeug by .theprodukkt [20] is a development environment allowing the user to specify and edit complex textures in the context of an operation stack. .werkkzeuggg allows for the development of models, textures, sounds and music. This results in incredibly small content specification files, with the cost of texture realisation passed onto the user during initialisation.

Related Course Material

This study will draw from several modules presented as part of the Computer Games Technology degree, particularly those of a purely mathematic nature, such as *Computer Animation and Maths for 3D Computer Games* (CMPCD2035).

I will also draw from my experience gained during my work placement year with Sony Computer Entertainment Europe. (June 2007 – August 2008)

Chapter 4: SOFTWARE SPECIFICATIONS

Design Goals

Overview

During the course of this project, several demonstrative pieces of software will be designed to create *and recreate* environments within the following specification:

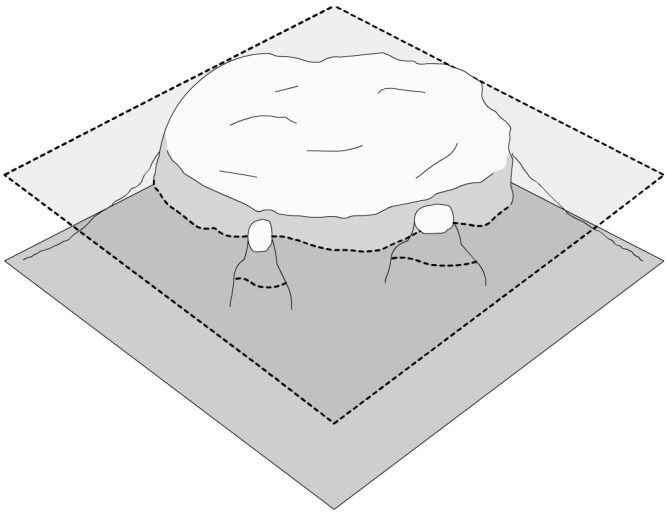

Figure 7: Oblique view of environment to be generated

The software is to generate and present an island environment *visibly* similar to the one shown in Figure 7.

The environment will feature at least one large island roughly in the centre of the environment. The island will have a large, roughly flat plateau, a unique, realistic, unpredictable coastline and the island may be surrounded by smaller islands, rocks or other environmental features. Water will exist in the simulation as a completely flat plane extending infinitely perpendicular to the y-axis. The generated topography should fall to below the water plane on all four sides of the height map, leaving no exposed raised land at the sides of the land mesh. The terrain will be coloured semi-realistically based on its elevation and gradient. The terrain generation algorithm will also provide indirect vertex data (such as normal information) necessary to realistically render the terrain on-screen.

Software Mode of Action

This section covers how the software will interact with the user and present the generated environment.

Upon start-up, the program will ask the user to enter a name for their environment. This name will be used to initialise the pseudorandom number generator, and the generation of the height map will begin.

As multiple initialisations of the pseudorandom number generator with the same input seed will result in the same series of pseudorandom numbers being generated, the same environment will be produced for any single input seed.

When this has completed, the completed terrain mesh will be presented to the user. The user will be able to navigate a virtual camera through the environment and view it at any angle.

The user will be able to be able to view the interim parameters used in the generation of the terrain. The user may also alter these parameters in real-time, with the on-screen landscape constantly regenerating to match the new parameters. This is to allow the user to explore the parameter space used in the preparation stage of terrain generation, with the understanding that the modified parameters no longer reflect the terrain that would be automatically generated with the seed string given.

At any time, the user can return to the seed string entry prompt and enter a different string. The software should produce exactly the same terrain when given a specific input seed by the user. Similarly, the software should create a completely distinct environment when any part of the seed is changed by any amount. The user can also opt to use the current system time as the seed string, producing an effectively unique seed (up to the granularity of the system time).

This design is similar in spirit to the Algomusic series of music synthesizers [24] [25] in that a user specified input string is used to create a specific piece of complex content which can be recreated at any time. An interesting experiment outside the scope of this study would be to run the software prototype created in this study and Algomusic simultaneously using the same string to create a stochastic, procedural 'experience' rather than simply a lone piece of music or a single piece of terrain.

Hardware Resources and Constraints

The software created as part of this project will be designed to run as a real-time application on the following hardware configuration:

- AMD Athlon XP 2500+ CPU (running at 1.83Ghz). (Single core processor)
- 1024 MB of 266Mhz RAM.
- GeForce FX 5200 AGP Graphics Card.
- HDD with at least 100MB free space.
- Colour monitor capable of running at a resolution of at least 1024x768.
- Keyboard.
- Mouse.

This configuration was chosen as it represents a very common baseline configuration for home PC systems. According to the *Valve Software Steam Hardware Survey March 2009* [21] , over 75% percent of participants were using home PC systems of this level or better. Note that this survey represents an up-to-date aggregation of

regular PC game software users who are likely to demand high quality game graphics and content with minimum storage requirements, especially given that Steam's primary method of content delivery is online.

It is likely that the software will run adequately on a lesser configuration, but performance issues may arise. Also, specific API functions may not be available on older graphics accelerator cards.

Software Resources and Constraints
The software created as part of this project will use the following libraries:

- Simple DirectMedia Layer (SDL), for windowing and input handling.
- OpenGL, (through Mesa 3-D graphics library) for graphics display and graphics acceleration.
- OpenGL Extension Wrangler Library (GLEW), for additional graphical techniques.
- GNU ISO C++ Library.
- Libnoise, for the generation of Perlin noise.
- Mersenne Twister MT19937, for the generation of random numbers.

The software will be compiled using the MingW GCC compiler suite and associated runtime libraries.

The following library is also used in the comparison of random number generators:

- Random Number Generation - Multiple Streams, rngs.c rngs.h.

Full copyright information for the libraries used is given in the Attributions section.

The software will run under Windows XP SP1 or later.

The software created as part of this project will be presented to the user as an interactive, real-time application. As such, the software should not remove control of the system from the user for prolonged periods (such as during the terrain generation sequence).

The user should be able to quit the software at any time.

Evaluation and Testing
The terrain generation algorithm and the generated environments will be assessed on their ability to resemble a realistic landform 'as shown' rather than by comparison to fractal models such as fractional Brownian noise, following suggestion by Mandelbrot [1] and Fournier *et al* [4] . By nature of what constitutes a 'realistic looking' landform,

however, some features of the generated terrain (such as the shape of the apparent coastline) may still representative of a ‘correct’ fractal form, even when this is not the direct intention.

Design Rationale

This specification has been chosen as it allows for a full and directed discussion of the PCG methods available to create a specific topographical form through stochastic modelling. The methods discussed in this study will be readily adaptable for use in many computer game development scenarios, thus providing an immediate benefit to developers wishing to include complex terrain in their software.

These methods can also be adapted for more analytic uses, such as the merging of disjoint terrain patches and the treatment of ‘holes’ in existing input datasets, but these are outside the scope of this study.

Chapter 5: DESIGN OF SOFTWARE PROTOTYPE

Simulation Prototype Design

Overview

The software prototype will have two operating states: 'seed entry' and 'view environment'. When the program starts, the simulation is placed into 'seed entry' mode. In this mode, the user is prompted to enter their desired seed string followed by the Return key to dismiss the dialog. During this entry sequence, the seed string is stored in a temporary array. When the Return key is pressed, the program will switch from 'seed entry' mode to 'view environment' mode; the temporary seed string is copied into the random seed string, with unused cells filled with zero.

On program initialisation, the temporary seed string will be set to empty. If the 'seed entry' mode is recalled, the temporary string is recalled, allowing the user to view and modify the last seed they entered.

When the 'view environment' mode is entered from the 'seed entry' mode, the software prototype will generate terrain based on the current random seed value. During 'view environment' mode, the user can freely navigate the virtual camera through the environment using the keyboard and mouse.

Coordinate System

The coordinate system used in the simulation is a standard three-dimensional Cartesian coordinate space, with the x-axis running from left (negative) to right (positive), y-axis running from below (negative) to above (positive) and the z-axis running from in front of the viewer (negative) to behind the viewer (positive).

Data Structures

The terrain is stored as a height map: a two-dimensional matrix of real values giving the displacement on the y-axis. To aid in the application of the terrain generation methods, the matrix will always be a square matrix with $2^n + 1$ values along each edge, giving a grid of 2^n quadrilaterals for easy subdivision, where *n* an integer representing the 'quality' of the terrain. Coordinates in a height map are measured from the back left side of the original land plane, with increasing x coordinates moving toward the right and increasing z coordinates moving toward the viewer.

Other methods such as voxel-based structures may be used to hold 3D terrain data, but these methods often have prohibitive memory requirements, and it is rarely the case that PC graphics hardware is optimised for their display.

The software will use this height map structure for all of its terrain modification calculations, and these calculations will be restricted to deformations on the Y-axis only. Figure 9 shows a possible height map that the software may generate. Dark pixels indicate areas of low elevation and light pixels indicate areas of high elevation.

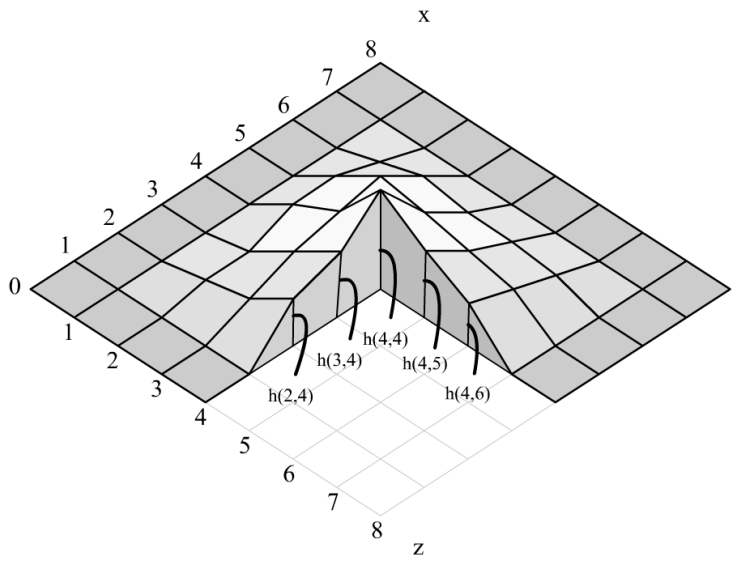

Figure 8: Section through a possible topography showing the displacement of points on the Y-axis on a regular grid on the XZ plane

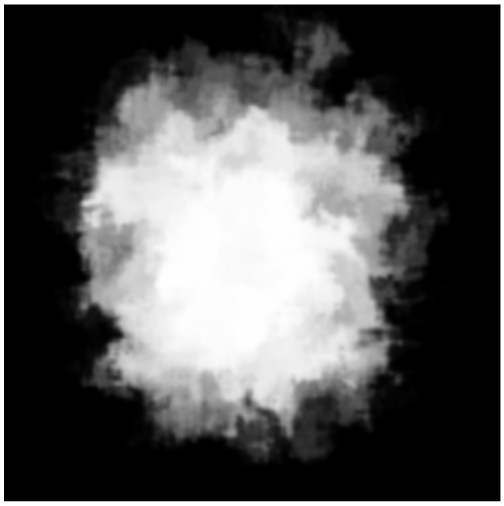

Figure 9: Height map for generated topography

This height map system is chosen for the reasons of simplifying calculations and ease of communication. If all calculations are performed using simple real values within matrices, then the complexity of modifying the terrain algorithmically is greatly reduced. There would be little advantage in terms of the quality of the resulting terrains if a more complicated mesh-based structure were used, especially in the specified 'island' scenario. Height maps allow the complete result of terrain generation to be directly visualised as a simple, easy-to-understand greyscale image. Height maps are also conducive to experimentation: one can readily refer to specific features present in a height map, or refer to metrics such as the distance from the centre of the map.

Random seed values are stored as arrays of 100 unsigned chars. The current random seed is stored in memory, together with a copy of the last seed entered in 'enter seed' mode: this seed is recalled if the user recalls the 'enter seed' mode.

Camera Simulation

The user views the generated environment through a 'virtual camera' specified by its position and a combination of three vectors, extending from the centre of the camera in the direction it is facing, to the left, and upwards. This camera specification allows for an interactive system to be implemented very easily: the ahead, lateral and upwards vectors form an orthonormal coordinate basis and the position vector is readily available, allowing them to be used as the rotation and translation parts of a 4x4 matrix representing the world-to-eye-space transformation necessary to view the environment through the camera. The camera system is designed to ensure internal consistency when these values are modified: if the ahead, lateral and upwards vectors were to not form an orthonormal basis, the camera matrix would become meaningless and distorted visualisation would result.

The user can move the camera in any direction relative to its current orientation using the keyboard, and can pitch, yaw or roll the camera using mouse gestures. A number of special function keys are implemented, allowing the user to restore the camera position and orientation to a predetermined preset, focus on the origin and eliminate camera roll.

Visualisation of Terrain

The topography is visualised as a regular grid of 2^n triangle pairs ABC, ACD on the xz-plane. Y values will be read directly from the height map. It is expected that these values will lie in [0, 1]. To allow the detail level of the texture to be increased without affecting its on-screen size, the resulting terrain mesh is scaled to fit within the coordinates [-1, 1] along the x and z-axes.

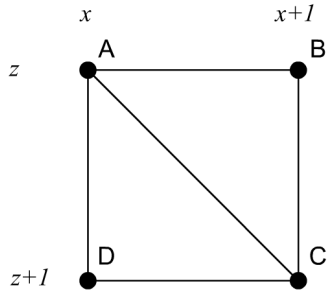

Figure 10: Tessellation of a grid cell

Normals are generated by calculating the cross product of vectors AB and AC. (AC and AD for the complementary triangle.)

The generated vertex positions, colours and normals are inserted into a composite type representing an OpenGL vertex, and passed to the API for rendering. The terrain visualisation is only necessary if the terrain generation parameters have changed. Therefore, when the visualisation has completed, the generated terrain mesh is stored

in an STL vector of this composite vertex type, allowing for the use of the OpenGL vertex array API as a method of accelerating the rendering. Early versions of the prototype used OpenGL Immediate Mode with caching of the generated mesh between frames, resulting in unacceptably slow rendering, especially for large values of n.

Selection of Pseudorandom Number Generator
This software prototype requires the use of a pseudorandom number generator with the following characteristics:

- It should have a uniform distribution.
    - The output of a pseudorandom number generator with uniform distribution can be reshaped to fit any required distribution.
- It must return its results quickly.
    - The speed of any algorithm using the pseudorandom number generator will be dependent on the speed of the generator.
- It should have a large period.
    - The number of possible states of the random number generator places a limit on the number of possible environments that can be produced. This should be maximized where possible.

The prototype developed in this report uses the Mersenne Twister. The analysis of the available pseudorandom number generators has been placed as an appendix.

Terrain Generation Model - Simple

Overview

For the scenario handled in this study, I propose the following model for the terrain generation method for my prototype application:

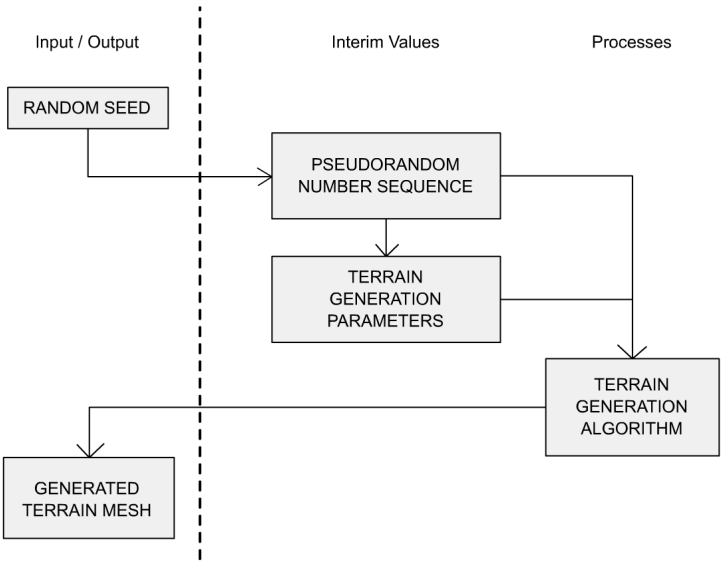

Figure 11: Basic model of processing in the software prototype

This diagram shows the relationship between data in the design of the software prototype. The Random Seed is the input to the system, and is used to produce the Pseudorandom Number Sequence, and from that the Terrain Generation Parameters. The Terrain Generation Parameters refers a collection of parameters for controlling a terrain generation process: for example, 'roughness' and 'decay' parameters for controlling a diamond – square algorithm. To restrict the range of terrains that can be created, the mapping from the pseudorandom number sequence should produce output values in a predetermined optimum range. The Pseudorandom Number Sequence element refers to the complete sequence of pseudorandom numbers used directly in the terrain generation algorithm (as if they were called in advance).

Together, these two elements describe the complete stochastic model used to produce the terrain described by the random seed. The mapping from the random seed to the pseudorandom number sequence is a one-to-one mapping, as are the mappings from the pseudorandom number sequence to the terrain generation parameters and the generated terrain. As a result, the random seed has a one-to-one mapping with the generated terrain: the generated terrain is reproducible.

This model allows a single algorithm to be used to generate terrain procedurally. As the character of the terrain produced solely relies on the choice of algorithm, my goal is to find an algorithm that will constantly produce terrain that meets the requirements listed in the Design Goals section.

Diamond – Square algorithm

Figure 12 shows the first terrain produced by a prototype using the diamond – square algorithm.

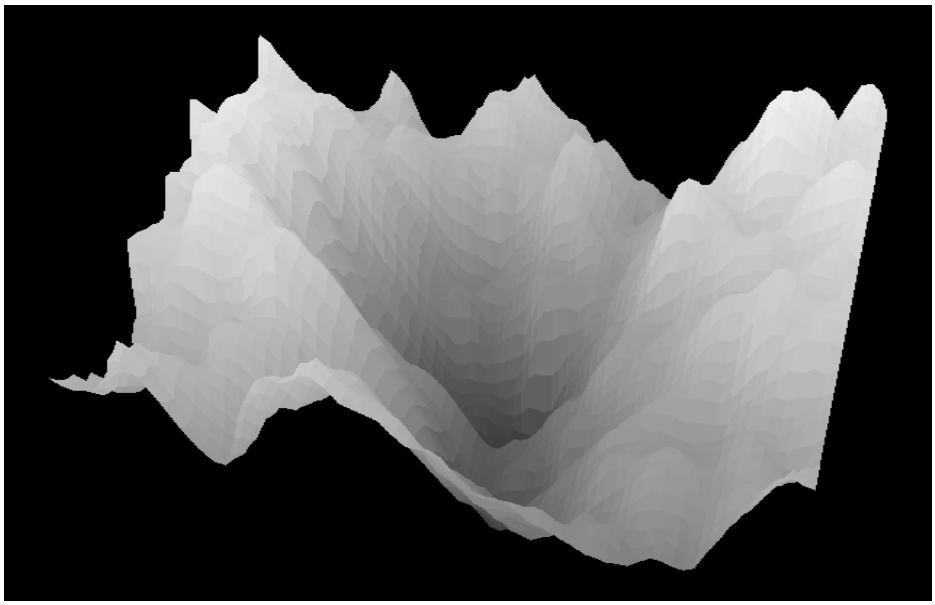

Figure 12: First terrain rendered using the diamond - square algorithm with wrapping height map

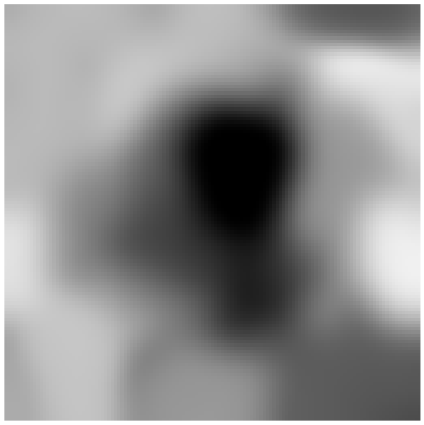

Figure 13: Height map image for previous terrain

This terrain was generated using the modified diamond – square algorithm using vertical displacements only described by Martz in [8] . (The original algorithm discussed by Fournier, Fussell and Carpenter [4] suggests mid-point displacements perpendicular to the original polygon, which is incompatible with the height map system I am using.)

This algorithm works by recursively splitting a square region into smaller square regions in two stages: the ‘square’ step, and the ‘diamond’ step. The centre points of these regions is then displaced by a random variable within a range directly

proportional to the inverse square of the level of recursion. The implementation used to generate the above images produce a terrain that seamlessly wraps with itself.

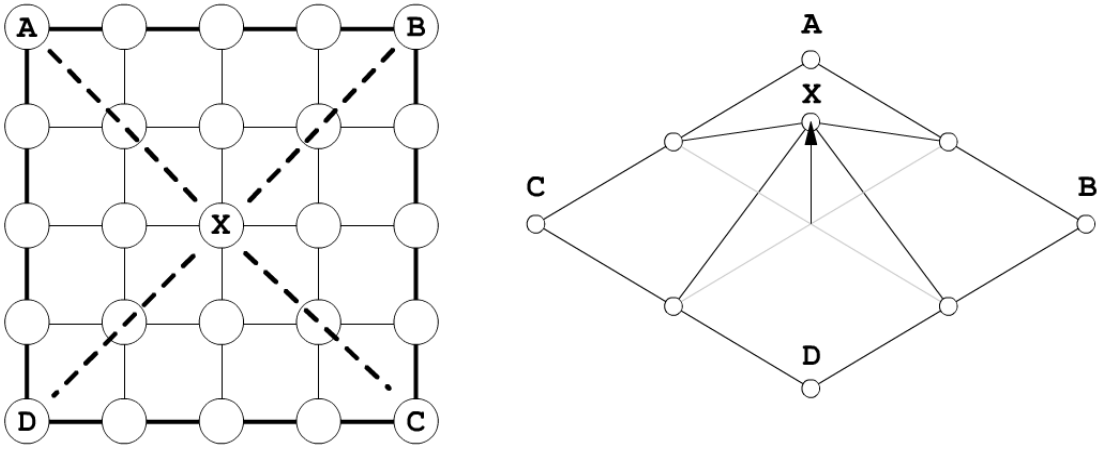

Figure 14: Diamond - square algorithm, square stage at recursion level 1

The above figure shows the first square stage of this algorithm. The bold lines show the edge of the current region being considered. The dashes lines show the relationship between the corners of the region and its midpoint. The four corner points of the square ABCD are averaged to give the original position of X. X is then displaced by a random value in the range [-r, r] where r is a value inversely proportional to the square of the current level of recursion, currently this is 1. As the square ABCD wraps both horizontally and vertically, A, B, C and D are all in fact the same point.

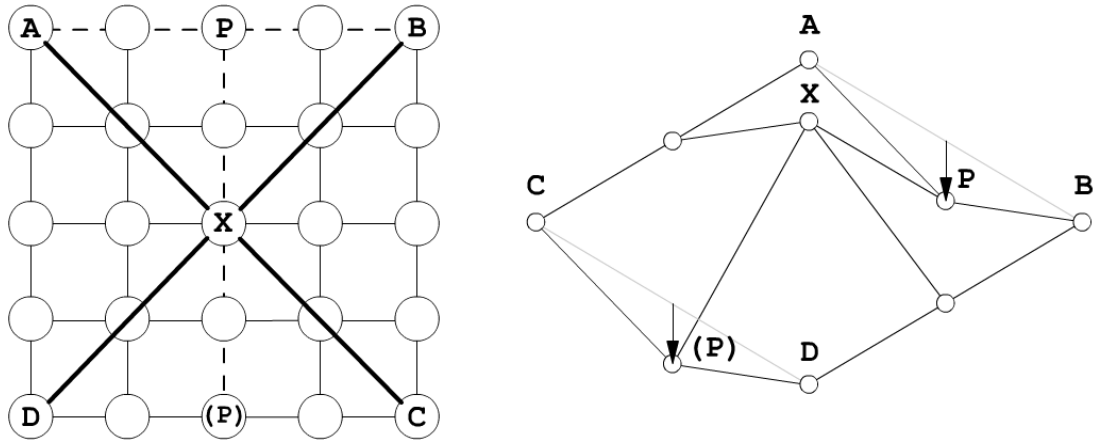

Figure 15: Diamond - square algorithm, diamond stage at recursion level 1

In the diamond stage, the four corner points of the diamond AXBX (specified clockwise), are averaged to give the original position of P. Note that the diamond AXBX starts at A, moves up off the edge of the grid to X, to B, then down to the centre of the grid. P is displaced by a random value in the range [-r, r] where r is a value inversely proportional to the square of the current level of recursion, currently this is 1. P and (P) are the same point, as the original square wraps. This stage is repeated for the point on the horizontal edge of the square at the midpoint of BC.

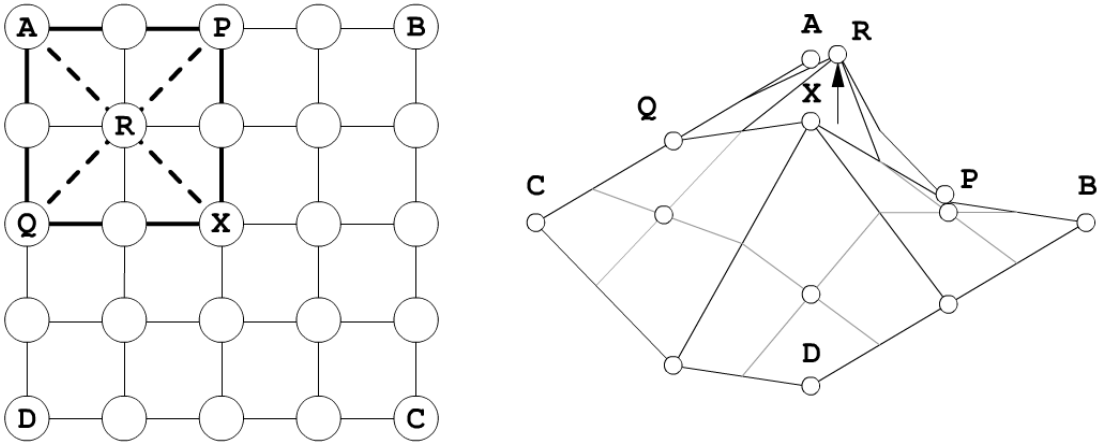

Figure 16: Diamond - square algorithm, square stage at recursion level 2

In the second level of recursion, the process is repeated with the four squares produced when the original square is split in half horizontally and vertically. Here, the four corner points of the square APXQ are averaged to give the original position of R. R is then displaced by a random value in the range [-r, r] where r is a value inversely proportional to the current level of recursion, now 2. This is repeated for the points to the top right, bottom right, and bottom left of X.

This algorithm can run for any number of recursions, with the number of quadrilaterals produced quadrupling with each extra level of recursion. Figure 17 and Figure 18 show the terrains produced using a grid of 32 and 256 quadrilaterals respectively. This algorithm is highly appropriate for terrain generation as its fractal nature is immediately evident from the description of the algorithm.

It is highly efficient in terms of storage as only three values are used to specify the terrain: the number of recursions, the initial value of r and the coefficient used to attenuate r. As the values are taken from the pseudorandom number generator in order of increasing detail, the level of detail of the terrain can be increased or decreased freely with no need to ensure that the random generator produces the values in the 'correct order'. For example, if the algorithm were to calculate the values in 'across first then down' order, it would be necessary to have strict control of order of values produced by the random number generator to insert or remove vertices between others.

In my implementation, the terrain mesh is stored in memory as a triangle list of OpenGL vertex composite types. If the level of recursion is altered, vertices are not merged or split: it is necessary to release the held memory and regenerate the terrain from scratch. In the prototype, a C++ class named WrappingArray is used to encapsulate a rectangular 2D array of elements of type T. The float type is used to store real values.

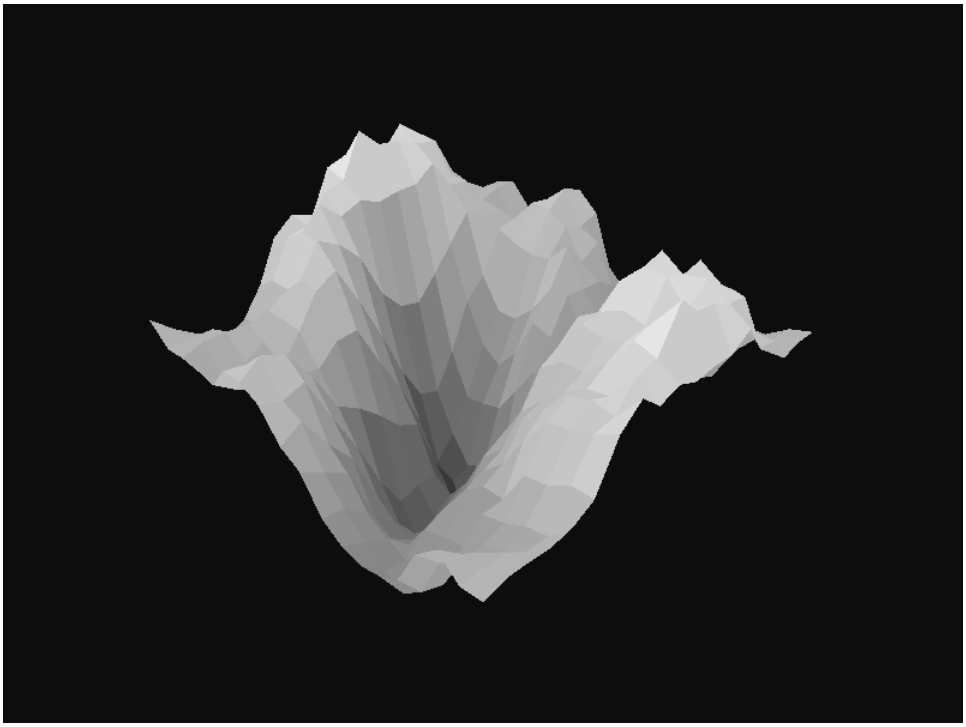

Figure 17: Terrain generated by diamond - square algorithm with 32 cells per side (2048 triangles)

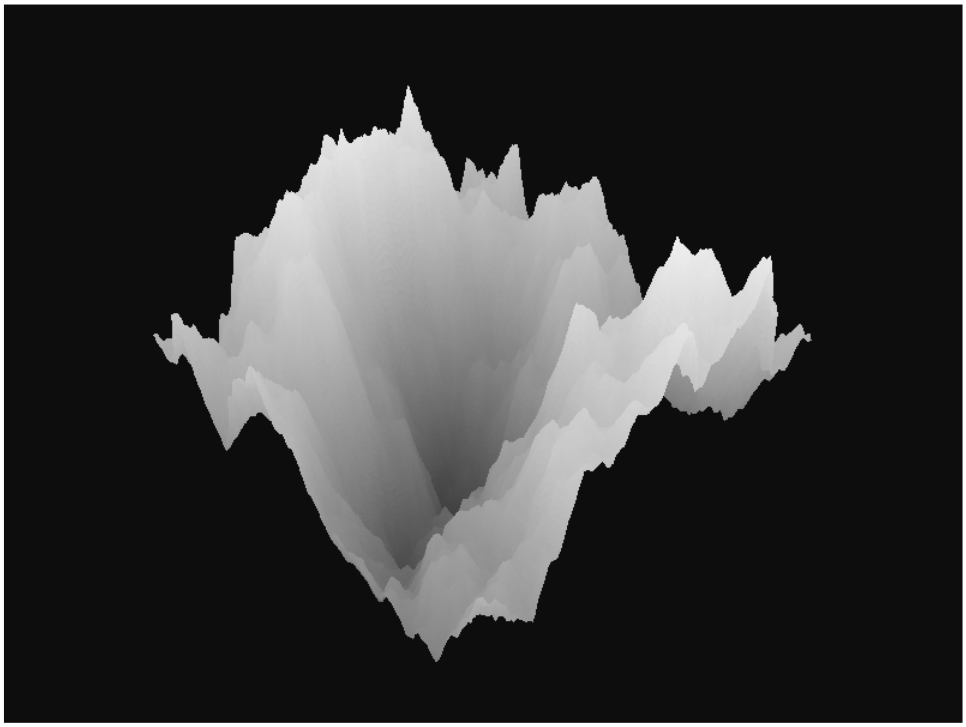

Figure 18: Terrain generated by diamond - square algorithm with 256 cells per side (131072 triangles)

This algorithm, as it stands, is not appropriate for the goals I have specified. The algorithm has too much variation in the landscapes it produces, and it generally produces exaggerated, jagged, mountainous terrain, even when the roughness parameter values are constrained. There is no guarantee that the resulting landform is an island; there is a significant chance that the resulting landform is a concave pit, instead of a convex raised mass.

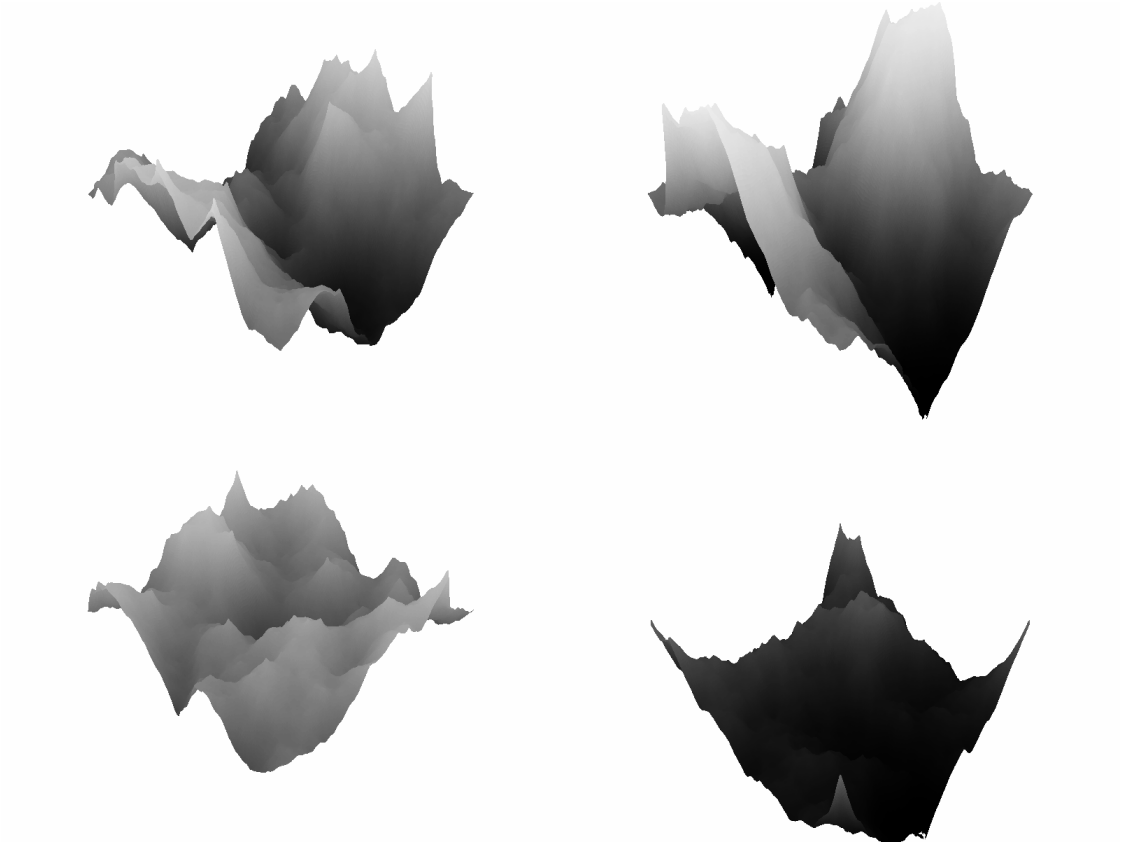

Figure 19: Selection of terrains produced by the diamond - square algorithm

It is possible to alter the algorithm so that it no longer generates terrains that seamlessly wrap. I suggest that this may cause the algorithm to generate more island-like terrain forms, as well as allowing for some extra consideration for the boundary of the surface. If the boundary of the surface were controllable, it would reduce the likelihood that the exposed edges of the terrain would be visible above the water level. This artefact is most visible in the upper right example in Figure 19, where a dominant raised feature lies on the boundary of the surface, causing raised, exposed edges on two sides of the mesh.

The original algorithm by Fossell et al suggests that the border of the terrain mesh be generated first as a fractal polyline followed by the body of the terrain mesh. Exposing the generation of the boundary as a separate process would allow fine control over the generation of the edges, but it would alter the order that the random numbers were retrieved: an increase in the level of detail would cause a variation in the amount of random numbers retrieved before the generation of the surface begins, causing an

entirely different surface to be generated. This could be mitigated by resetting the random number generator to a known position after the boundary polyline has been generated.

A simpler solution would be to use the algorithm without modification, but reject any stages that attempt to retrieve the position of points that lie outside of the surface. This would cause all points on the boundary of the surface to be unmodified, as well as causing all significant displacements to be restricted to the centre of the surface. As the intended surface is an island, it is sufficient that all four edges are below the surface of the water.

As suggested by Martz, it is possible to specify a number of fixed points in the mesh and have the algorithm interpolate other points around these. This method has some limitations due to the order in which the displacements are calculated: a fixed point will have more influence on the surrounding points if it lies in a position where it would be encountered early in the algorithm. If the centre point is fixed, all other points will be affected, if a point ¼ of the way across the surface is fixed, only points in the same ¼ square will be affected.

Through experimentation, I have found that this algorithm is too erratic to try and 'force' the creation of an island using a selection of fixed points near the centre of the height map.

Perlin Noise / Simplex Noise

Perlin Noise is a coherent noise space function mapping multidimensional input values into real values [16] . This discussion also applies to the successor of Perlin Noise, Simplex Noise, as it was designed toward the same goals, has the same features, interface and usage.

Noise is widely used for the procedural generation of textures and other effects. A single sample of Noise has a pseudorandom appearance, with features all of a similar size. By combining multiple scaled copies of Noise ('octaves'), complex textures can be evolved.

There are many techniques for using Noise to generate terrains. By filling a height map using a two-dimensional sample of multi-octave Noise, similar results to the diamond – square algorithm are produced. Attempting to create islands using low frequency, high amplitude Noise octaves results in a surface of a suitable character, but there is no way to specify the position of the island or tell that the island will be sufficient height or size.

Unlike the diamond – square algorithm, this algorithm does not rely on the use of a pseudorandom number generator to amplify the data, and therefore does not have the same conditions on its use between detail levels. It is possible to sample the Noise space at any resolution required; increasing detail exists at all ranges as long as there are sufficient octaves near the frequency of resolution.

As Noise is simply a space function, the inputs to a Noise based terrain generator specify the nature of the mapping between the coordinates in the space of the height map and the coordinates in Noise space. Values taken from the pseudorandom number generator specify the number of octaves of Noise to combine, and a mapping function for each, which can take any form.

In isolation, Noise does not allow for the generation of a specific type of feature, such as the island specified in the Design Goals. It is necessary to combine multiple octaves of Noise with further functions defining the valid boundary of the island to achieve this. This is explored fully in the advanced terrain generation model.

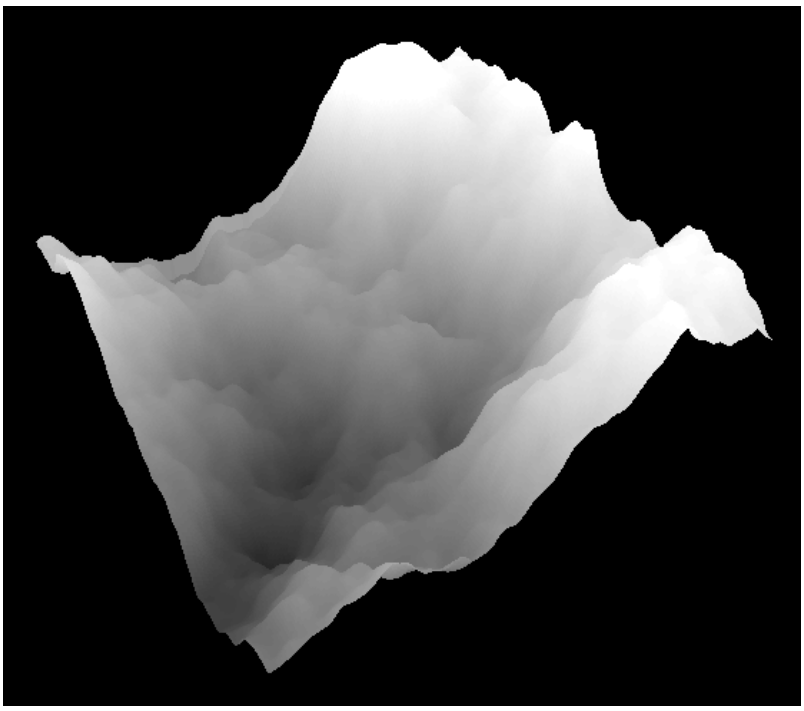

Figure 20: Terrain generated through Perlin Noise (libnoise), 256 cells across (131072 triangles)

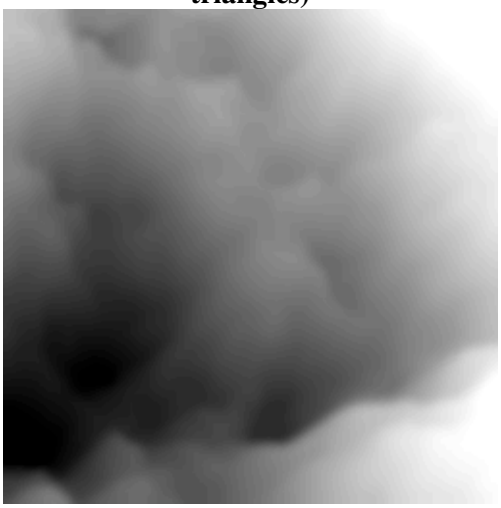

Figure 21: Height map for the previous terrain

Mathematical Specification

It is possible to attempt to specify an island explicitly using parametric functions and patches. The possible range of landforms is limited to what can be specified through the functions, and the output is often considered too abstract or artificial. It is for this reason that Noise and other related coherent noise functions were developed.

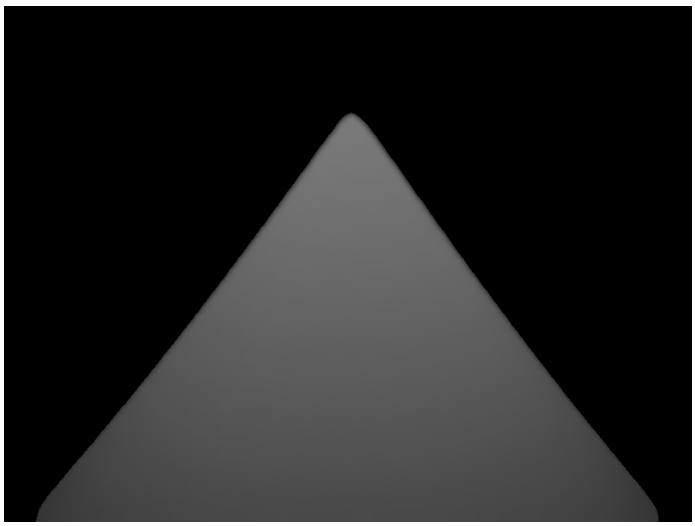

Figure 22: Mock-conical landform generated parametrically

More complex structures can be built up if the number of functions used in their construction is increased dramatically. This will incur a cost in increased processing time.

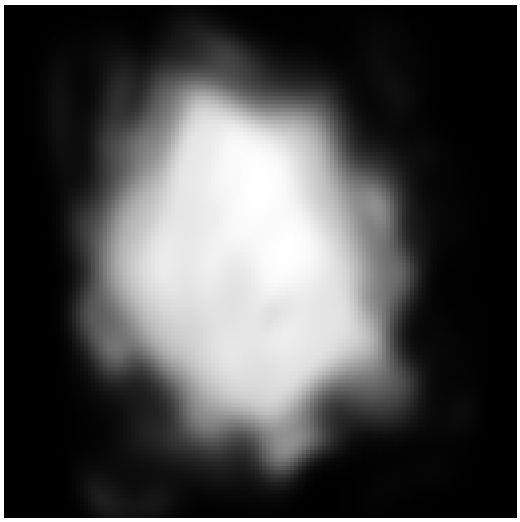

Figure 23: Height map generated through repeated addition of 20,000 normally distributed, randomly placed Gaussian kernels

Conclusion

Using the terrain generation model I have proposed, it is very difficult to produce a terrain that meets the goals I have specified. Each of the methods I have identified all excel at producing arbitrarily detailed surfaces with a specific *character*, but none of them directly allow for specifying the location or position of major features without adversely affecting the rest of the landscape.

However, this is not due to any deficiency in the methods. Both diamond – square fractal subdivision and Noise are emulations of fractional Brownian motion, and as such are designed to exhibit properties such as overall scale independent self-similarity. They are not designed to allow for the specification of major features, and would indeed be flawed if they did.

In response, I propose a more sophisticated terrain generation model.

Terrain Generation Model - Advanced

Overview

Based on my initial analysis of the methods available, I propose the following improved processing model:

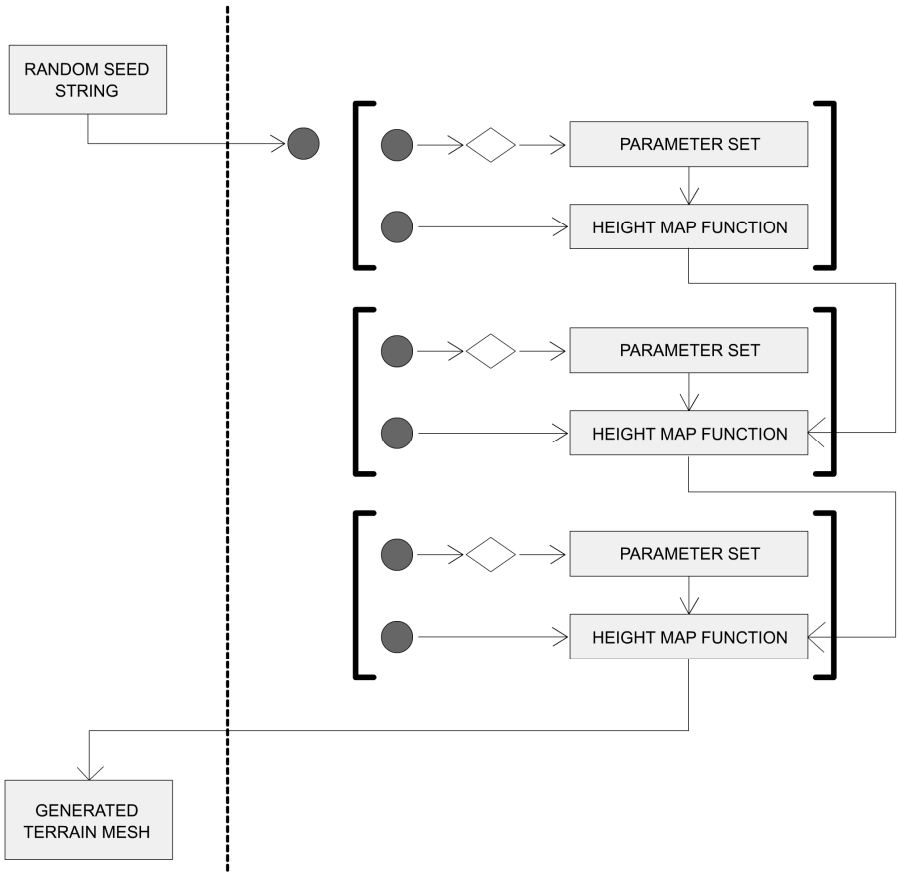

Figure 24: Advanced model of processing in the software prototype

This diagram shows the relationship between data in the design of the software prototype. The shaded circle represents the pseudorandom number generator as a data source. Whenever the pseudorandom number generator is used as a data source, it is reset first to a known offset from the start of the sequence given by the seed string, so that each use of it is independent. The unshaded diamond represents a mapping from real values [0, 1] into appropriate parameter values.

The Random Seed String is the input to the system, and is used to seed the pseudorandom number generator. The terrain generation takes place as a sequence of operations, shown in brackets in the diagram. Each operation takes a number of parameters derived from the pseudorandom number generator and transformed to an appropriate value, together with an input height map, if appropriate. These operations each return a height map as a rectangular array of real values in the range [0, 1] (they are not clamped, so it is possible to overflow). Each operator encapsulates a single specification and realisation of a stochastic model.

As the data is processed by each operation in turn, a complex height map with all of the desired features can be evolved.

As with the previous model, the mapping from the random seed to the pseudorandom number sequence, the mapping from the pseudorandom number sequence to a parameter set or height map function, and the mappings between operations are all one-to-one. As a result, the random seed has a one-to-one mapping with the generated terrain: the generated terrain is reproducible.

The height map functions defined in the operations should all work in a height map coordinate space ([0, 1], [0, 1]) so that they become invariant when the detail level changes.

This model allows the programmer to specify exactly how the data is handled at each step of the process. The overall character of the landscape is defined through the operations chosen and the mapping from the pseudorandom number generator to the parameter values. It will be an experimental process to discover a series of operations that produce the desired outcome. To determine the appropriate mapping for the parameter, the prototype will allow the user to override these values temporarily and see the effects that they have on the environment in real-time. I intend to use this function to discover the 'correct' value for each exposed parameter, and then instruct the operation to use parameters within a certain deviation from this value. This will cause the generated environments to all have a similar but not identical nature.

The overall result of this model is that every environment generated will have the same overall features, but will show a wide variation in the supplementary details.

This model is similar to the .werkkzeug environment by .theprodukkt [20] as discussed in the Study of Existing Research section. My model proposes that the order of operations is 'baked' into the application with only the seed string exposed as a parameter, whereas .werkkzeug defines content by its parameterised operator stack, allowing the user full control.

Design of Suitable Composite Model

The choice of operation and parameter ranges is directly influenced by the nature of the goal environment: each operation (or combination of operations) in turn contributes one of the features found in the goal environment.

The major features required to create the desired island environment are:

- A dominant landmass near the centre of the height map.
- An irregular coastline similar to a real island.
- The possibility of smaller islands close to the dominant landmass.
- No land above water at the boundary of the height map.

Before constructing the composite model, I will formalise a number of operations based on the algorithms already discussed:

Fill(value) returns a height map filled with copies of value.

DiamondSquareNoWrap(original_r, r_attenuation, input_height_map, fixed_point_set) returns a height map filled with values calculated using a diamond – square fractal subdivision algorithm (with wrapping disabled by disallowing off-height-map reads) against input_height_map with original deviation value original_r, r attenuation constant r_attenuation. The array fixed_point_set defines the location and height of points that are to be fixed.

PerlinNoise(perlin_parameters) returns a height map filled with values retrieved from a Perlin Noise space defined by the mappings in the structure perlin_parameters.

Add(height_map_a, height_map_b) returns a height map containing the sum of each of the elements in height_map_a and height_map_b by element-by-element addition.

Subtract(height_map_a, height_map_b) returns a height map containing the value of (a – b) for each of the elements in height_map_a and height_map_b by element-by-element subtraction.

Multiply(height_map_a, height_map_b) returns a height map containing the product of each of the elements in height_map_a and height_map_b by element-by-element multiplication.

ClampLower(height_map_a, minimum_value) returns a height map containing the value of max(height_map_a.element(…), minimum_value) by element-by-element comparison.

ClampUpper(height_map_a, maximum_value) returns a height map containing the value of min(height_map_a.element(…), maximum_value) by element-by-element comparison.

R[…] returns a random value in the indicated range. This is a value retrieved from the pseudorandom number generator and scaled.

The level of detail and water level are global variables accessible by all operations. All height values should be between 0 and 1. The water level is a value between 0 and 1 indicating the y-coordinate of the water plane. All height values below this are considered to be under-water, and are culled by the renderer to accelerate rendering.

Using this syntax, the original diamond – square algorithm would be expressed as:

Output = DiamondSquareNoWrap(1.0, 0.5, Fill(0))

Instinctively, the 'no land above water' condition can be enforced by the formulation of an operation that causes the edges of the height map to be lowered if they are close to the edge of the height map:

Attenuate(input_height_map, k, p) returns a height map containing values from input_height_map that have been altered based on the proximity of the current element to the edge of the height map.

To ensure a smooth gradient, I specify Attenuate() using the formula:

$$\text{value} = \text{input} \cdot k \left(1 - \frac{\text{distance from element to centre}}{\text{distance from corner to centre}}\right)^{p}$$

This formula causes the input values to falloff toward zero in a parabolic fashion. It assumes the values are non-negative.

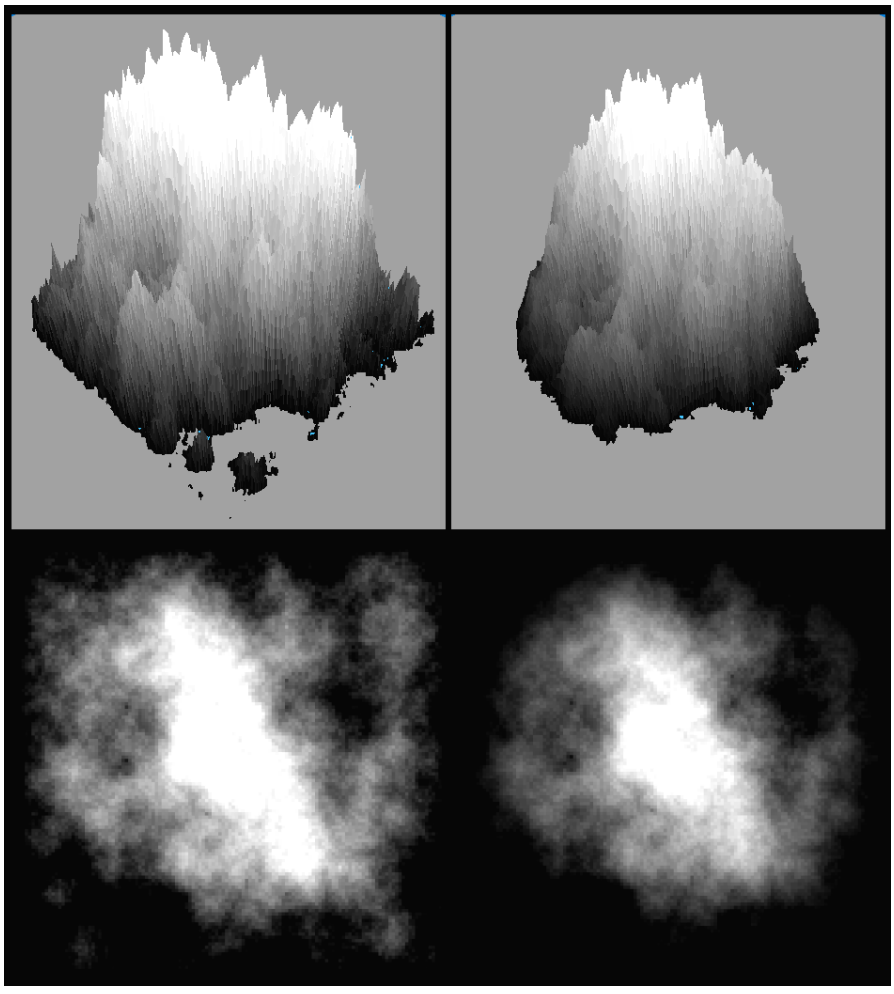

Figure 25: Left: landscape before Attenuate(), right: landscape after Attenuate()

Generally, the Attenuate() operation will vertices around the periphery of the height map towards zero, and as a side effect, will cause the generated landform to have a more circular appearance. The jagged character of the landscape remains, however.

To flatten out the highest values of the height map, the Clamp…() operations can be applied to simply 'cut off' the top of the landscape.

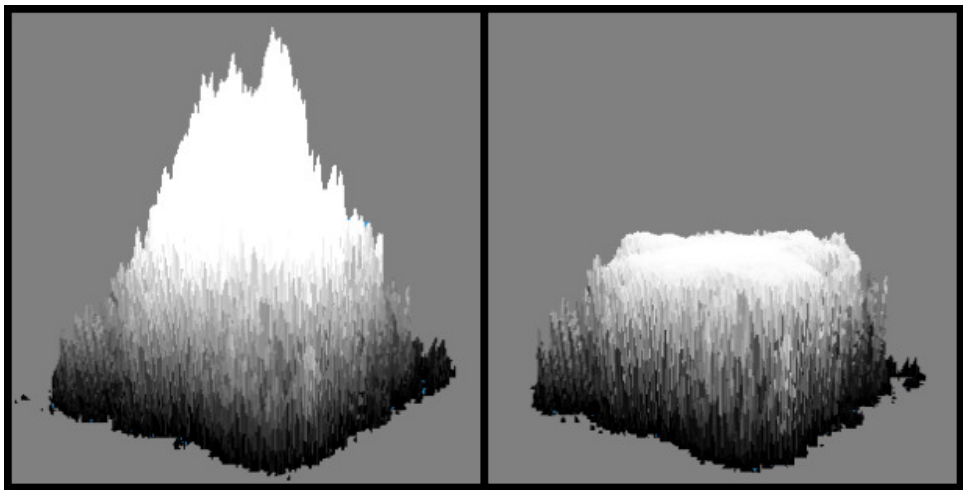

Figure 26: Effect of the ClampMax(1) operation on the terrain mesh

The ClampLower() operation should be used to ensure that vertices that extend below the water line do not exceed a minimum negative value. Extreme negative values, even below the water line, may cause graphical anomalies during rendering.

The ClampMax operation leaves a harsh edge where a steep gradient may intersect the, now flat, top plane. A more sophisticated approach is to use a non-linear saturation function to collapse extremely high height values into a narrow range.

SigmoidCollapse(input_height_map, pre_subtract, pre_scale, a, b, c) returns a height map where the values from input_height_map have been transformed using the following equation:

$$j = pre\_scale \,.\, (element - pre\_subtract)$$

$$output = \frac{a}{1 + e^{b.(c-j)}}$$

The resulting curve forces extremely low values to become near zero, and extremely high values to collapse towards a. b is a constant controlling the steepness of the curve. The curve becomes more steep with increasing b. c translates the original terrain values left or right along the curve. Experimentation is required to determine the appropriate parameters for b and c. a should be 1, causing all the highest points in the terrain to saturate there.

Figure 27: Curve described by SigmoidCollapse()

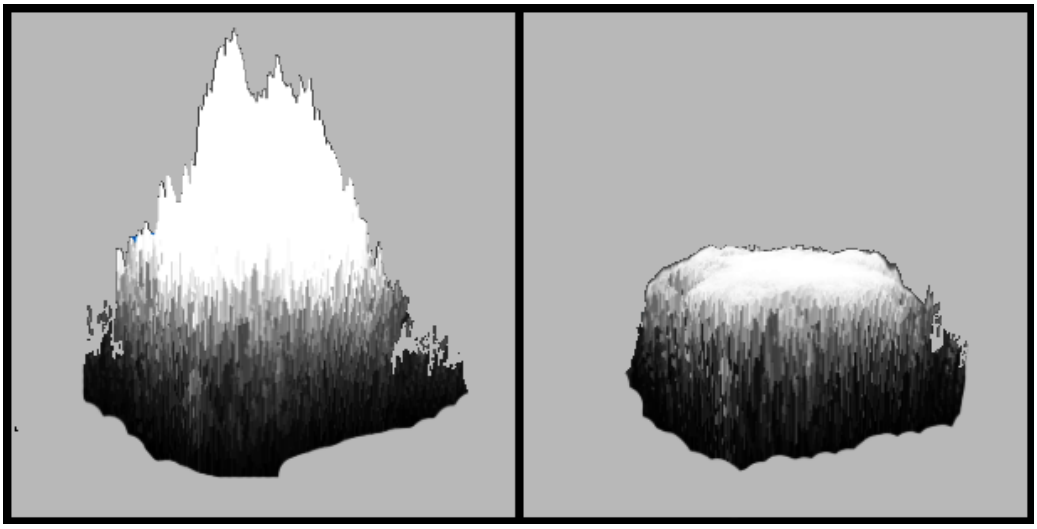

Figure 28: Effect of the SigmoidCollapse() operation on the terrain mesh

The SigmoidCollapse() operation has the effect of increasing the gradient of all values that lie in the range (0.3, 0.7). This can be advantageous if the water level is set at 0.9; all vertices but the highest will be discarded and those that remain will have a rounded gradient leading towards a flattened plateau.

The resulting land mass is a very good representation of the desired island. It meets all of the goals listed in the Design Goals. Using SigmoidCollapse(), the height of the island is specified by (a – water_level).

However, the results of this algorithm are dependent on the original height map saturating against the curve. If the original height map is relatively flat or negative, there will be no visible land.

Figure 29: Terrain mesh produced by modifying the parameters of SigmoidCollapse()

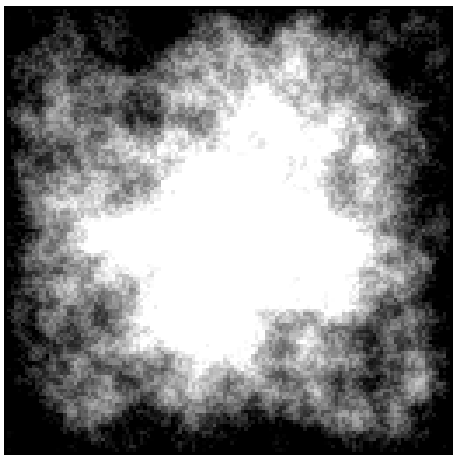

Figure 30: Height map for the above terrain

To counteract this, a fixed point structure is defined instructing the original diamond – square algorithm to fix the central point in the height map at a parameterised value. All of the other points in the height map will then 'reach' toward this point while exhibiting the normal Brownian motion characteristics. By intentionally saturating the height map and collapsing it with the SigmoidCollapse() operator, a certain minimum area of land can be guaranteed, regardless of the results of the subdivision. A similar strategy would be to simply seed the original height map with high conical or spherical values before the subdivision is called.

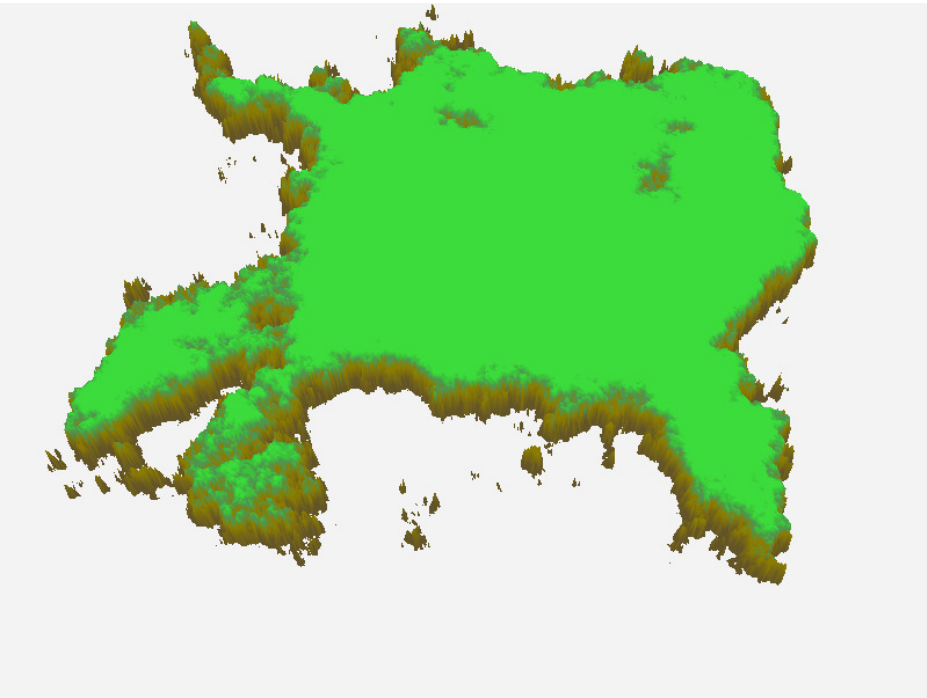

Figure 31: Landmass generated by fixing the central point and intentionally saturating.

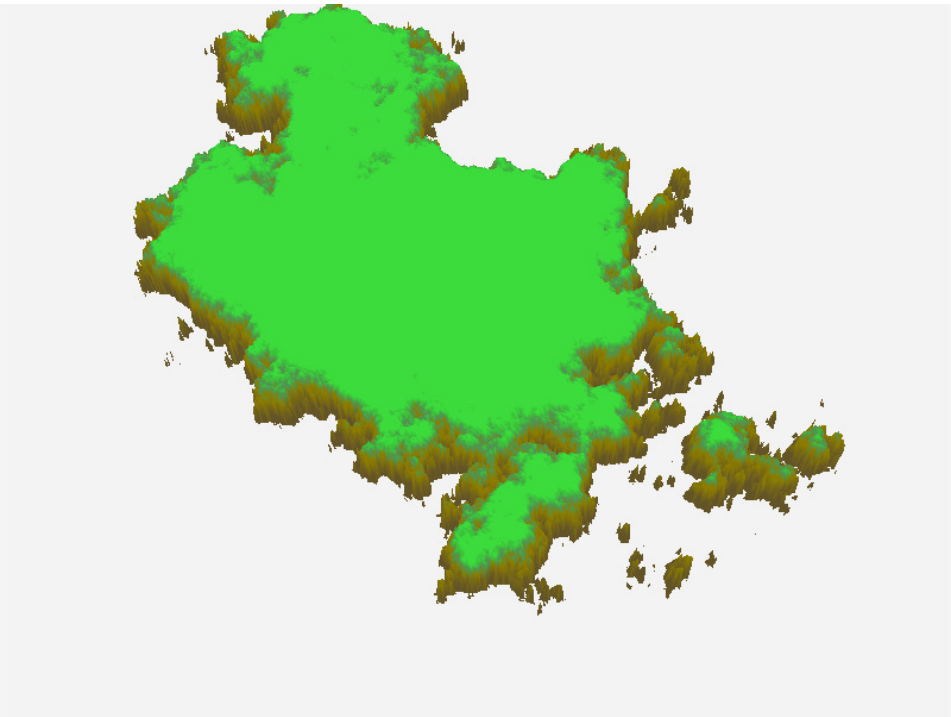

Figure 32: Landmass generated by fixing the central point and intentionally saturating.

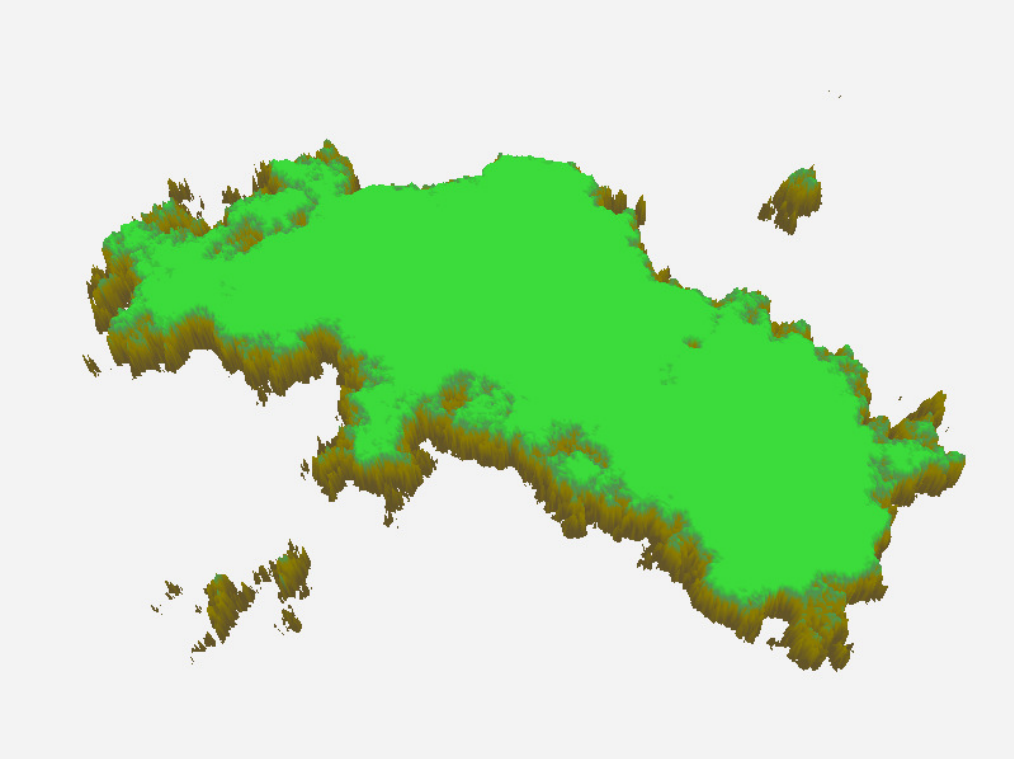

Figure 33: Landmass generated by fixing the central point and intentionally saturating.

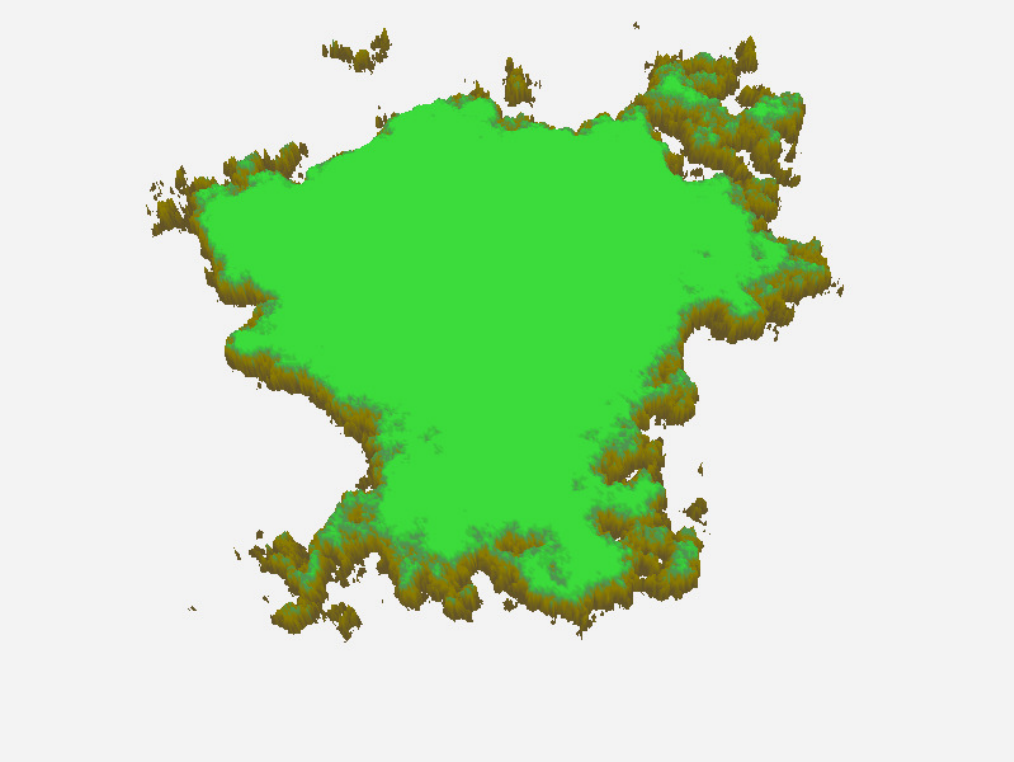

Figure 34: Landmass generated by fixing the central point and intentionally saturating.

To allow for slight variations in the produced islands, the values used for fixing the central height point, the values used in the sigmoid curve and the parameters used in the subdivision are all specified in the model as ranges rather than constants. For example, the peak height used to generate the above images experimentally was set at 10. Now that I have determined that this is a value that returns good results, the final algorithm has been programmed to run using values that are approximately 10.

All of these islands now meet the defined goals, regardless of the input seed. The use of the non-linear mapping function has caused the environment to no longer represent fractional Brownian motion, but this is not a concern. As a point of interest, the coastline remains representative of Brownian motion, as it is an isosurface defined by the intersection of the horizontal water plane and the results of the original height map.

The top surface of the island may appear flat, but it still retains the detail from the original subdivision, albeit in a compressed fashion. It may be desired that the island have some additional roughness within the final landform. To achieve this, the surface can be multiplied by Perlin noise to cause it to become arbitrarily perturbed.

Output = Multiply(PerlinNoise(…), output_height_map)

This algorithm is highly flexible, allowing for a large amount of customisation and experimentation. For example, the original height map may be filled with values from Perlin Noise, prior to the subdivision stage, to provide additional randomness. If the final model does not use diamond – square subdivision, it will no longer be limited to height maps that are 2^n+1 in length.

See the Usage Notes section in Appendix A for a tutorial on how to manipulate the software prototype to explore how the parameterised values alter the generated land.

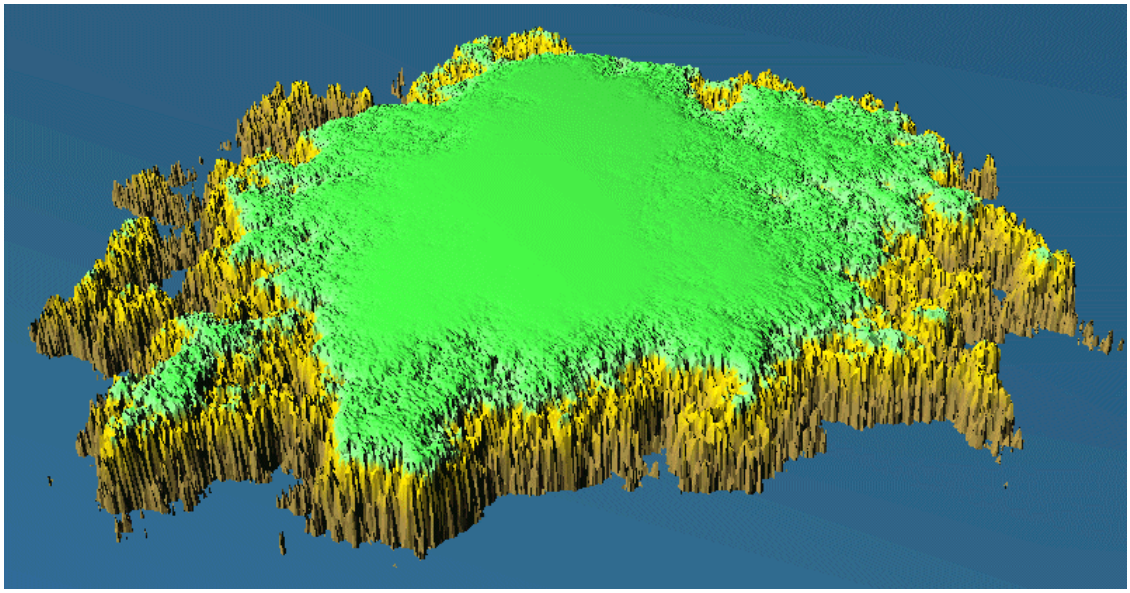

Figure 35: Complex reproducible procedurally generated landscape with lightning enabled.

Chapter 6: TESTING AND CONCLUSIONS

Testing and Conclusions

As stated in the Software Specifications section, the terrain generation algorithm is assessed on its ability to resemble a realistic landform 'as shown' rather than by comparison to fractal models such as fractional Brownian noise.

From my experience with the prototype, and an informal survey carried out during development, it appears that the resulting terrain meshes are good on-screen representations of genericised landforms.

It may be possible in future to define exact metrics that describe what is a 'good' environment, such as minimum 'habitable' land surface, and then adjust the experimental model parameters to correct the surface. A multi-layer Perceptron neural network may be a useful tool for this task as it is well-suited to seeking ideal values in a large parameter space. However, the implementation of this is outside the scope of this study.

The model described in this study has been specified outside the context of scale, as is the case with most abstract graphical constructions. It is up to the user to determine their exact needs, i.e. do the islands represent continents, or small islands? The algorithm will accommodate islands of any nature through adjustment of the parameterised operations stack.

Chapter 7: FURTHER WORK

Further Work

The model proposed in this study is simple, but effective. There are many additions to the algorithm which may be implemented to provide a more detailed or realistic terrain:

The terrain produced through this model takes the form of a solid, unbroken mass of land; the algorithms do not allow for the generation of rivers or streams through the land. To do this, algorithms such as those described in [10] and [15] where predetermined points cause specific shapes to be evident in the final terrain mesh. To achieve this, a fractal polyline subdivision method may be used to define 'rivers' that should run through the final terrain. These 'river' definitions are then used to constrain the terrain generator, resulting in evident rivers in the final height map. Also, as water is specified here as a completely horizontal plane, the height map structure does not allow for rivers that lie along elevated land.

The generated islands are completely unpopulated. A real game scenario would require the surface to be populated with buildings, road infrastructure, vegetation and other additional features. The terrain mesh may be analysed, and appropriate items added to the specification. This could take the form of additional operations that accept height maps as input and output classes storing building and vegetation placements. The use of shape grammars such as L-systems provides an interesting starting point into procedural city generation techniques. The papers [22] [23] provide a possible city generation algorithm that is compatible with the height maps produced by the proposed algorithm. Similarly, by careful use of the pseudorandom number generator, the resulting city would also be reproducible.

The renderer used in the software prototype is functionally primitive. An enhanced renderer may run the algorithm presented by the model multiple times and retrieve copies of the mesh at different detail levels. Then, the mesh can be subdivided, allowing the renderer to control the level of shown detail dynamically: far away features would be rendered using a lower detailed representation, while close features would be rendered using a highly detailed representation.

Chapter 8: REFERENCES

[1] Mandelbrot, B. (1975) "Stochastic Models for the Earth's Relief, the Shape and the Fractal Dimension of the Coastline, and the Number-Area Rule for Islands" PNAS Volume 72 Number 10. 3825 – 3828.

[2] Mandelbrot, B. (1967) "How long is the coast of Britain? Statistical self-similarity and fractal dimension" Science 155, 636 – 638.

[3] Lévy, P. (1965) in Processus Stochatiques et Mouvement Brownien. Gauthier Villars, Paris. 2nd ed.

[4] Fournier, A., Fussel, D., Carpenter, L. (1982) "Computer Rendering of Stochastic Models" Communications of the ACM Volume 25, Issue 6. Pages 371 – 384.

[5] Lewis, J. P. (1987) "Generalised Stochastic Subdivision" ACM Transactions on Graphics (TOG), Volume 6, Issue 3. Pages: 167 – 190

[6] Carpenter, L.C. (1980) "Vol Libre" Computer generated animated movie first showing at SIGGRAPH '80.

[7] Miller, G. S. P. (1986) "The Definition and Rendering of Terrain Maps" Proceedings of the 13th Annual Conference of Computer Graphics and Interactive Techniques. ACM SIGGRAPH. Pages 39 – 48.

[8] Martz, P. (1997) "Generating Random Fractal Terrain" Game Programmer, gameprogrammer.com. http://www.gameprogrammer.com/fractal.html (retrieved October 2008)

[9] Max, N. L. (1981) "Vectorized Procedural Models for Natural Terrain: Waves and Islands in the Sunset" ACM SIGGRAPH Computer Graphics Volume 15, Issue 3. Pages 317 – 324.

[10] Stachniak, S., Stuerzlinger, W. (2005) "An Algorithm for Automated Fractal Terrain Deformation" Proceedings of Computer Graphics and Artificial Intelligence.

[11] Szeliski, R., Terzopoulos, D. (1989) "From Splines to Fractals" Proceedings of the 1989 ACM SIGGRAPH conference. ACM SIGGRAPH Computer Graphics. Volume 23, Issue 3. Pages: 51 – 60.

[12] Vemuri, B.C., Mandal, C., Lai, S.H. (1997) "A Fast Gibbs Sampler for Synthesizing Constrained Fractals," IEEE Transactions on Visualization and Computer Graphics, vol. 3, no. 4, pp. 337.

[13] Fujimoto, T., Ohno, Y., Muraoka, K., Chiba, N. (2002) "Fractal Deformation Using Displacement Vectors Based on Extended Iterated Shuffle

Transformation" The Journal of the Society for Art and Science, Vol. 1 (2002), No. 3 pp.134-146.

[14] Kelly A.D., Malin M.C., Nielson G.M. (1988) "Terrain Simulation Using a Model of Stream Erosion" SIGGRAPH '88, Pages 263 – 268.

[15] Belhadj, F. (2007) "Terrain Modelling: A Constrained Fractal Model" Proceedings of the 5th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa. ACM SIGGRAPH. Pages 197 – 204.

[16] Perlin, K. (1985) "An Image Synthesizer" Courant Institute of Mathematical Sciences. Computer Graphics, Vol. 19, No. 3. (also in Computer Graphics: Image Synthesis, IEEE, Salem, 1988)

[17] Perlin, K. (2001) "Noise Hardware" In Real-Time Shading SIGGRAPH Course Notes.

[18] Gustavson, S. (2005) "Simplex noise demystified" http://webstaff.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf (retrieved December 2008)

[19] Bevins, J. Libnoise maintainers (2003) "Libnoise, A portable, open-source, coherent noise-generating library for C++. Example: Complex planetary surface" http://libnoise.sourceforge.net/examples/complexplanet/index.html (retrieved October 2009)

[20] .theprodukkt, ".werkkzeug" (2004) Procedural content generation development environment. http://www.theprodukkt.com/

[21] Valve Software (2009) "Steam Hardware Survey: March 2009" http://store.steampowered.com/hwsurvey/ (retrieved April 2009)

[22] Parish, Y. I. H. (2001) "Procedural Modeling of Cities" Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques. ACM SIGGRAPH. Pages 301 – 308.

[23] Kelly, G., McCabe, H. (2006) "A Survey of Procedural Techniques for City Generation" ITB Journal, Institute of Technology, Blanchardstown, Dublin.

[24] Schürger, T. (1995) "AlgoMusic" Amiga music synthesizer. Available on AMINET. Official website: http://web.archive.org/web/*/http://fsinfo.cs.uni-sb.de/~schuerge/AlgoMusic/

[25] Timiney, D. (2008) "Dunc's AlgoMusic" Windows music synthesizer. Official website: http://www.wikihost.org/w/doodleblog/dunc_s_algomusic

Chapter 9: ATTRIBUTIONS

Libraries Used

The following third-party libraries are used in this project:

- Simple DirectMedia Layer (SDL), for windowing and input handling. SDL is licensed under the LGPL. http://www.libsdl.org/
- OpenGL, (through Mesa 3-D graphics library) for graphics display and graphics acceleration. Mesa 3-D is released under the MIT License.
- OpenGL Extension Wrangler Library (GLEW), for additional graphical techniques. GLEW is released under the MIT License. http://glew.sourceforge.net/
- GNU ISO C++ Library. Licensed under the LGPL.
- Libnoise, for the generation of Perlin noise. Licensed under the LGPL. http://libnoise.sourceforge.net/
- Mersenne Twister MT19937, for the generation of random numbers. Freely made available by the authors for any purpose.

The software was compiled using the MingW GCC compiler suite and associated runtime libraries.

The following library is also used in the comparison of random number generators:

- Random Number Generation - Multiple Streams, rngs.c rngs.h. Made available for this project. http://www.cs.wm.edu/~va/software/park/

Chapter 10: APPENDICES

Appendix A

Guide to Using the Prototype Software

The prototype software used to generate the images in this report is included on the attached CD-ROM in the directory '10 Final Prototype'.

When the programs initialises, you will be prompted to enter a seed string. Feel free to type any word or phrase you wish, then press Return. You will then be presented with an interactive simulation of the resulting environment.

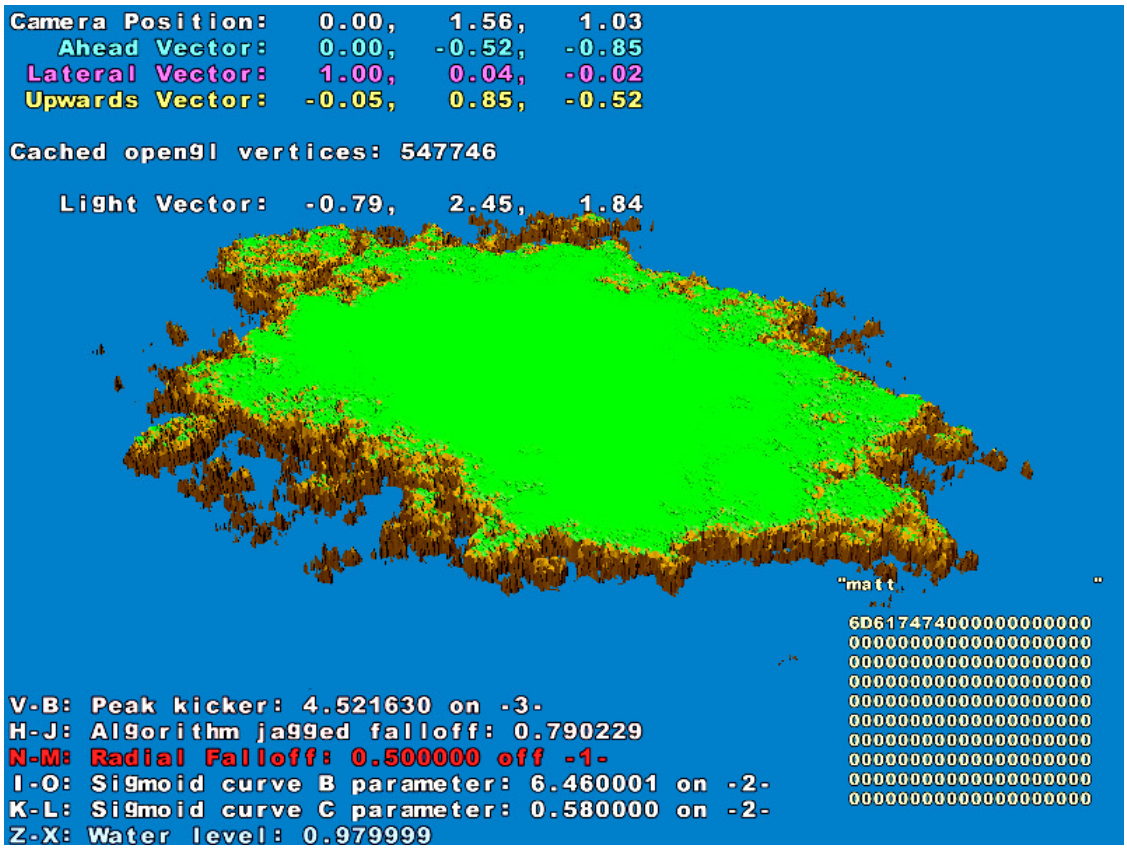

Figure 36: Screenshot of prototype application

To recreate the terrain shown in Figure 36, do the following:

- Enter the seed string 'matt' (lowercase, without quotes) and press Return.
- Use the VB keys to set the 'Peak kicker' value to 4.521630, and enable the fixed peak value by pressing 3.
- Use the IO keys to set the 'Sigmoid curve B parameter' to 6.460001, and enable it by pressing 2.
- Use the KL keys to set the 'Sigmoid curve C parameter' to 0.580000.
- Use the ZX to set the water level to 0.979999.
- Press F2 to position the camera for viewing terrains based between 0.95 and 1.00

- Press 6 to enable lighting.
- Press Page Down to cycle to 'pseudorealistic terrain colouring'
- Press + (plus) to increase the detail level.

Controls

W, S: Move the camera forwards and backwards.
A, D: Move the camera left and right.
Q, E: Move the camera up or down.

Hold left mouse button and move mouse: Alter orientation of camera.
Hold right mouse button and move mouse: Move camera forwards and backwards.
Hold both mouse buttons and move mouse: Roll camera around current direction.

Main keyboard plus / minus: Increase and decrease the level of detail. Warning: increasing the detail too high may cause the program to slow down or crash due to lack of memory.

H, J: Alter the roughness attenuation coefficient for the diamond – square subdivision.

1: Enable the radial attenuation function.
N, M: Increase and decrease the radial attenuation power parameter.

2: Enable the sigmoid curve.
I, O: Increase and decrease the B parameter of the sigmoid curve.
K, L: Increase and decrease the C parameter of the sigmoid curve.

3: Enable or disable the fixed peak point.
V, B: Increase and decrease the peak fixed point value (if enabled).

4: Enable/disable water semitransparency. (Shows partly obscured not-yet-culled underwater polygons.

5: Enable/disable smooth shaded terrain.

Z, X: Raise or lower water level.

R: Return to the seed entry screen. Enter the same seed string and return to an environment previously visited!

F1: Restore the camera to the original position facing (0, 0, 0). This is the position the camera is in when the application begins.

F2: Restore the camera to the position facing (0, 1, 0). This is an ideal position to observe the land when the water level has been raised when using the sigmoid curve operation.

F3: Move the camera to look at the environment from above.

F9: Cause the camera to become upright. This is useful if you have rolled the camera over by accident.

F11: Aim camera at (0, 1, 0). (Elevated sigmoid / water level land position) This is useful if you move the camera away from the environment and want to face it again.

F12: Aim camera at (0, 0, 0). This is useful if you move the camera away from the environment and want to face it again.

Home: Take a screenshot and save it in the 'screens' folder. (Does not work when run from CD-ROM)

End: Enable/disable high contrast water.

Page Down: Enable/disable smooth shading.

Delete: Toggle between three different colouring modes:
- Height map dependent colouring.
- Height map 'pseudorealistic' green/brown colour pallete.
- Totally white.

Lowercase g: Output a tga copy of the current height map, with colours adjusted so that black is 0 and white is 1. (Does not work when run from CD-ROM)
Uppercase G: Output a tga copy of the current height map, with colours adjusted so that water level is black. (Does not work when run from CD-ROM)

In the lower right corner is the current seed. The ASCII values of the current seed are shown above a grid showing the hex values of the entire 100 byte string. To increment the seed string by 1, press TAB. This results in a unique environment using the current settings.

Usage Notes
To begin, attempt to replicate the 'matt' terrain using the instructions on the previous page. When you have done so, experiment by changing the parameters gradually.

You will find that when the peak kicker / sigmoid curve are both activated, the peak kicker will have a global effect on the overall size of the major land mass in the centre of the height map. The roughness constant can be altered to change the erraticity of the terrain. If it is reduced sufficiently, the land will become almost circular.

You can explore how the attenuation function alters the terrain by enabling it and altering its value. You can notice that with increasing attenuation, the land will contract and become more circular.

Altering the sigmoid curve values will alter the 'corners' of the terrain. Reducing the B parameter will cause the land to become more rounded at the corners. If the B parameter is reduced to zero, the sigmoid curve will become a line and the terrain will

cease to be sensible. Increasing B will cause the land to have almost right-angled corners. You can change the C parameter to alter where the 'hotspot' of the curve lies: there will be a point where the island appears to contract suddenly. When this happens, you have moved the highest point of the island to the left hand side of the steepness in the sigmoid curve.

Additional Prototypes

There are a number of other prototypes included on the CD. These prototypes are largely uninteractive and simply display an example of a specific terrain technique.

In these, the QW keys change the roughness attenuation coefficient for the diamond – square algorithm, or advance the Perlin noise mapping by a short value (to show that the noise is continuous and coherent). OP keys zoom the camera in and out.

Appendix B

Quantitative Analysis and Selection of Random Number Generator

This software prototype requires the use of a pseudorandom number generator with the following characteristics:

- It should return real values in the range (0, 1)
    - Binary pseudorandom number generators exist, but are not appropriate for this project.
- It should have a uniform distribution.
    - The output of a pseudorandom number generator with uniform distribution can be reshaped to fit any required distribution.
- It must return its results quickly.
    - The speed of any algorithm using the pseudorandom number generator will be dependent on the speed of the generator.
- It should have a large period.
    - The number of possible states of the random number generator places a limit on the number of possible environments that can be produced. This should be maximized where possible.
- It should not gather its data from a large external storage file of values.
    - Loading values from external storage limits the range of the data source and reduces the period of the returned data (if it were to be accessed sequentially using an array-like interface). External data values would also be considered an integral part of the project as a whole, increasing its total file size significantly, contrary to the aims of this project.
- It should not rely on a large amount of memory or external storage during run-time.
    - Devoting a large amount of memory to the data source is contrary to the aims of the project.

The following pseudorandom number generators are considered for this project:

- Mersenne Twister (MT19937)
- Default ANSI C rand() function
- rngs.c by Steve Park.

All of these random number generators have the ability to (or, in the case of the default ANSI C rand() function, can be adapted to) return real numbers in the range (0, 1).

The memory usage requirements for the Mersenne Twister is less than 20 kilobytes, with rngs.c needing even less. The memory requirements for the ANSI C rand() function are unknown.

The period for the Mersenne Twister is $2^{19937} - 1$, the period for rand() is 4,294,967,296, the period for rngs.c is 2,147,483,647.

Running the program included at the end of this appendix, the following results were obtained for the generation of 600,000,000 random numbers:

- 36.632s for ANSI C rand() function
- 18.497s for Mersenne Twister
- 19.838s for rngs.c

These results were obtained on a 1.6Ghz Intel Pentium M processor.

For this project, the Mersenne Twister will be used for its exceptionally large period and fast calculation speed. The Mersenne Twister also has a useful interface allowing it to be seeded using a data string of arbitrary length, which is highly appropriate for the seed string concept explored in this project.

```
#include <cstdlib>
#include <cstdio>
#include <ctime>

#include "mt19937ar.h"
#include "rngs.h"

inline double ansi_c_rand_wrapper()
{
   return ((double)rand()) / ((double)RAND_MAX);
}

inline float ansi_c_rand_wrapperf()
{
   return ((float)rand()) / ((float)RAND_MAX);
}

template <typename T>
void PerformQuantitativeRandomAnalysis(
   T (*random_function)(),              /* Function pointer to function producing Ts */
   unsigned int repetitions_per_cycle,  /* How many numbers to generate within a cycle */
   unsigned int cycles,                 /* How many cycles to run */
   float time_for_cycle[]               /* Output: Array of floats to store
                                           the time in seconds to run a cycle */
)
{
   clock_t time_start, time_end;

   /* Run each cycle. */
   for (unsigned int cycle = 0; cycle < cycles; cycle++)
   {
      time_start = clock();

      for (unsigned int repetition = 0; repetition < repetitions_per_cycle; repetition++)
      {
         random_function();
      }

      time_end = clock();
```

```
      time_for_cycle[cycle] = ((float)(time_end - time_start)) / ((float)CLOCKS_PER_SEC);
   }
}

int main(int argc, char *argv[])
{
   MT::init_genrand(time(NULL));
   RNGS::PutSeed(time(NULL));
   srand(time(NULL));

   const unsigned int NO_REPETITIONS = 20 * 1000 * 1000;
   const unsigned int NO_CYCLES = 30;

   double (*randfunction[3])();

   randfunction[0] = ansi_c_rand_wrapper;
   randfunction[1] = MT::genrand_real2;
   randfunction[2] = RNGS::Random;

//   float (*randfunction[3])();
//
//   randfunction[0] = ansi_c_rand_wrapperf;
//   randfunction[1] = MT::genrand_realf2;
//   randfunction[2] = RNGS::Randomf;

   for (int f = 0; f < 3; f++)
   {
      float outputs[NO_CYCLES];

      PerformQuantitativeRandomAnalysis<double>(randfunction[f], NO_REPETITIONS, NO_CYCLES,
outputs);

      float total = 0.0f;

      for (unsigned int i = 0; i < NO_CYCLES; i++)
      {
         total += outputs[i];
         printf("%7.3fs ", outputs[i]);
         if (((i+1)%5) == 0) printf("\r\n");
      }

      printf("= %7.3fs for method %d \r\n\r\n", total, f+1);
   }

   return 0;
}
```

Appendix C
Final Year Project Specification – 26th October 2008

Mathew Carr – Student Registration Number 261177
CMPGN200X Final Year Project Specification

Degree course:
Computer Games Technology (BSc. Hons)

Project title:
A Study of Procedurally Generated Reproducible Environments

Subjects to be studied:
Repeatable construction of complex 2D and 3D environments through the evaluation of procedural content generation algorithms given known or unknown seed data, with the aim of using the method in a manner comparable to compression.

Aims:
This project aims to explore how procedural generation methods can be used to create reproducible environments for interactive software in different genre contexts, and to create software prototypes capable of generating re-enterable 2D and 3D interactive environments based on this.

Description of background:
Graphical computer games of any complexity require some amount of game assets to allow the simulated scenario to be visualised and presented to the user. These game assets take many forms: bitmap images, model files, shader definitions, audio files, game scripting files and aggregated combinations of these which may include metadata and other aggregation information linking related assets.

As an example, a single 3D 'level' in a modern game may consist of any number of environment meshes, textures and shaders, collision data, event scripting, and references to static within-level objects with their own required assets. Simpler games with minimal graphics still have the same requirements: a rudimentary version of Pong will still incorporate graphics for the playing field, ball, bats and score numbers.

Creating the content necessary to produce a game is a costly and complex process, which can involve many different people at many stages of production. This is traditionally a manual (albeit computer aided) process: textures must be drawn by artists, sounds created by sound engineers and level objects and events placed by level designers. Technologies have been developed to automate many parts of this process; these are described as 'procedural generation' techniques.

'Procedural generation' is a loosely defined term; it can be accurately used as a description of almost any measure of programmatically assisted content generation.

It can be used to describe the process whereby an author can produce a well-defined environment, texture or other piece of content by constructing the content from (or augmenting an existing piece of content using) a series of configurable mathematical abstractions. These are then evaluated by the game engine at run-time to produce a complete piece of content, effectively using a mathematical function in the place of a large series of explicit values. This process has been used in the games Just Cause and Darwinia to simulate large, detailed island archipelago environments.

The use of this process has two advantages: it gives the author the ability to specify content outside the context of a set level of detail. This means the game content can be realised by the software in any desired level of detail upon demand. It also allows for complex pieces of content to be represented using a smaller set of data than would be used to explicitly specify the content. These advantages incur a cost in cycles and time due to evaluating the content at run-time.

Procedural generation also applies to the process of generating a piece of content based on a series of generalised parameters. For example, Sim City 2000 allows the player to specify the nature of the environment they would like to play with by manipulating a number of variables describing the amount of starting forest, severity of elevation, presence of rivers, etc. Sim City 2000 generates different environments each time it is asked to, even if its input parameters remain constant.

Through combining different procedural generation methods, procedural generation can be used to automate the generation and placement of a specific class of object within an existing environment, as performed by the tree generation software SpeedTree, or it can be used to generate the entirety of a game environment, as is done within the games Elite and Frontier: Elite 2 to create, position and name the multitude of ingame galaxies which the player can reach.

If the procedural content generation algorithm is designed to do so, it is possible to exactly reproduce a piece of content when called upon to do so. The galaxies within Elite are procedurally generated by using data retrieved from a pseudo-random number generator as input to galaxy creation routines. The result of this is the specification of an incredibly large, random-seeming, yet completely reproducible game environment specified through strict rules.

This project aims to explore how similar methods can be used to create reproducible environments for interactive software in different genre contexts.

Problems to be addressed:
This project aims to create software prototypes capable of generating re-enterable 2D and 3D interactive environments.

It will do this by accepting some measure of seed data and applying this to an algorithm to create a data source that provides repeatable pseudo-random data. It will then be possible to generate an interactive environment through the use of a procedural environment generation program, using this data source as a source of input parameters.

Through this method, several key areas will need to be explored:

- Choice of data source
  A data source will need to be identified for use in the procedural environment generation. The data source should have the following characteristics:

  - It should not gather its data from a large external storage file of values.
    Loading values from external storage limits the range of the data source and reduces the period of the returned data (if it were to be accessed sequentially using an array-like interface). External data values would also be considered an integral part of the project as a whole, increasing its total file size significantly, contrary to the aims of this project.

  - It should not rely on a large amount of memory or external storage during run-time.
    Devoting a large amount of memory to the data source is contrary to the aims of the project.

  - The data source must ideally work as if it were a transparent array with a similar interface, and queriable in constant time.
    If it became necessary to access distant elements of the data source when the environment is explored in a certain manner, a fast access time is necessary. The values in from the data source must be identical given identical input.

- Choice of scenario to demonstrate environment generation or configuration.
  The environment chosen will determine the methods used in the adaptation of the data retrieved from the data source. The type of environment chosen must be wholly generatable given only a series of parameter inputs (though creator specified additional directives may be included to force the presence of a specific configuration in the output), and the environment generated must be identical given identical input. Almost any type of environment can be constructed in this manner.

- Implementation of chosen scenario.
  Interactive prototypes of the chosen scenarios will be created in a suitable programming language.

Milestones:

- Case studies of existing software incorporating procedural generation to create environment.
- Case studies of existing software incorporating procedural generation to create other significant pieces of content.
- Study, evaluation and implementation of suitable pseudo-random number sequence source.
- Software prototype creating re-enterable 2D environment based on user input or system clock.
- Software prototype creating re-enterable 3D environment based on user input or system clock.
- Evaluations of software prototypes, to include comparisons with environments made with manual methods and discussion of possible enhancements or further work.

Software and hardware constraints:

There will be no hardware constraints enforced other than that of the use of a modern PC running Windows XP.

There will be no software constraints other than the use of Windows XP and a suitable development suite. The project will aim to keep system resource use to a minimum, but no constraints will be put in place.

Initial Ideas:

A possible very effective source of pseudo-random numbers would be to simply take the 'array index' input to the function and return a convolution of it. For example, the MD5 hash function can be used to mapping the 'array indices' into a 128-bit value, creating a fully populated, pseudo-random source of numbers that can be queried at any location in linear time.

Array index	MD5 Hash
0	CFCD208495D565EF66E7DFF9F98764DA
1	C4CA4238A0B923820DCC509A6F75849B
2	C81E728D9D4C2F636F067F89CC14862C
3	ECCBC87E4B5CE2FE28308FD9F2A7BAF3
4	A87FF679A2F3E71D9181A67B7542122C
5	E4DA3B7FBBCE2345D7772B0674A318D5
6	1679091C5A880FAF6FB5E6087EB1B2DC

To attain the convoluted number when seeded with a given value, this could be achieved by multiplying the array index by the seed value, or some other function such as XOR. The hashes can be manipulated or interpreted in any way necessary. Multiple hashes can be combined if necessary.

Using a pseudo-random number generator such as the Mersenne Twister or the ANSI C function may have disadvantages when it comes to attempting to retrieve numbers at a specific location within a sequence. If a separate random number generator sequence were used for multiple instances of an object (for example, the parameters of a given planet, of which there may be many), seeds would need to be stored, or a method for determining seeds defined.

Possible ideas for a software prototype include the production of a 2D environment island scenario:

- The island land shape could be expressed using any number of different methods, ranging from the simple drawing of a number of pseudo-randomly influenced shapes of land against a completely blank water 'canvas', or by more complex methods using overlapping samples of scaled Perlin noise to simulate a complex island formation. An even more involved simulation could simulate weather effects on the created land mass.
- The island could possess a number of basic 'zones' such as grassland, beach or desert based on their positions from the coast of the island, among other factors.
- Pseudo-randomly placed cities could be placed on the island where land permitted, up to a maximum amount or maximum density.
- Roads could join together cities accessible within a certain criteria such as distance, if the island were to be analysed as a graph and a path finding algorithm applied. Roads would then automatically avoid 'hazardous' zones by assigning them a prohibitive cost, and road creation attempts between unfeasibly distance cities would be aborted

Any one algorithm should produce the exact same island layout each time it is invoked using the same input seed to the pseudo random number generator.

This is a basic and crude algorithm, but it can be used to great effect if not taken as an exact representation of the island, but instead used as a starting point for extrapolation. For example, 'forest' zones may exist as an irregular green area of colour in basic prototype software, but a more developed prototype would use the zone as a region within it could activate an automatic tree generation algorithm to populate the zone. Cities, represented by coloured rectangles on the basic prototype, would have their own building and road placement algorithm ran within their circle of influence.

My use of the word 're-enterable' applies only to the environment generated by the algorithm, i.e. its initial state after generation. If I were to create a simulation similar to Scorched Earth or Worms whereby the level would be automatically generated based on a number of parameters such as 'number of hills', 'hill erraticity', terrain deformation performed by the players during gameplay would not be retained if the randomly generated level were to be to re-generated at another date. In a game context,

such alterations to the environment's state after generation would have to be recorded in a save game either as a complete set of the current state (meaning that procedural level regeneration would not be used to recover the saved game), or as a set of deltas representing the difference between the state as it was when the level was generated and the current state as it stands now (meaning that to recover the saved game, procedural regeneration would take place to restore the level, followed by the effects against the level stored in the save file).

There are many articles on the use of procedural content generation.

Ken Perlin has created a Java applet simulation of a procedurally defined planet that is rendered in increasing detail within a browser.
http://www.mrl.nyu.edu/~perlin/demox/Planet.html

This article by Sean O'Neil describes a way to generate the media needed to simulate a universe in real-time:
http://www.gamasutra.com/features/20010302/oneil_01.htm

Real time use of procedural generation can be CPU heavy. This article by Haim Barad, Mark Atkins, Or Gerlitz & Daniel Goehring describes how processor SIMD commands can be integrated manually into the routine as an enhancement (in available):
http://www.gamasutra.com/features/19980501/mmxtexturing_01.htm

There exists an in-depth wiki regarding the subject of procedural content generation in games. I will investigate a number of the games on their 'Games Featuring Procedural Content Generation' list in my report.
http://pcg.wikidot.com/category-pcg-games

Other games not included on the wiki will also be investigated such as Sim City, Lotus Turbo Challenge 3, Timestalkers and others.

I will also keep searching for additional papers and other resources.

Outline plan of action:

- Identify existing pieces of software incorporating procedural generation as part of their environment creation, and produce case studies.
    - The game genre known as ‘Roguelikes’ are a good basic example of this.

- Identify existing pieces of software incorporating procedural generation to create other pieces of content, and produce case studies.
    - .kkrieger is a good example as it uses procedural generation to generate all of its models and textures from abstractions.

- Identify pseudo-random number sequence generator.
    - This may take the form of a run-time speed comparison of various different alternatives.

- Prepare presentation for delivery in December.

- Produce software prototype creating re-enterable 2D environment based on user input or system clock.
    - I.e. the user can provide predictable input to produce the same level multiple times, or the system clock can provide an unpredictable one.

- Produce software prototype creating re-enterable 3D environment based on user input or system clock.
    - This may be an extension of the above prototype, though it may not strictly simulate the same scenario.

Timetable:

October, Week 5 to November, Week 4

Produce case studies of existing software employing procedural generation of 2D and 3D environments. Identify techniques which may be studied and expanded upon further within the project.

November, Week 4 to November, Week 5

Design software prototypes to exemplify techniques identified previously.

November, Week 5

Investigate pseudo-random number sequence generators, and produce quantitative comparison. Upon identification of a suitable sequence generator, develop the necessary utility software to allow the generator to be used in subsequently developed prototypes.

December, Week 1

Prepare project presentation for delivery in subsequent week. Presentation to include discussion of case studies in progress up to this point, and any prototypes developed.

December, Week 2

Deliver project presentation to staff and students.

December, Week 2 to January, Week 2

Formalise research and prepare interim report for submission on or before 11th January 2009.

January, Week 2 to March, Week 2

Software implementation of prototypes designed in the previous year. Alterations to previous designs will be documented and discussed.

March, Week 2 to April, Week 2

Preparation of report, to include evaluation of software prototypes developed during the course of the project. Evaluation will include quantitative comparison between procedural content generation methods and manual content production methods. Suitable areas for further exploration will be identified, and possible methods of expansion will be discussed.

April, Week 4

Project submission date: 22nd April 2009

Appendix D

Project Management Log

November, Week 1

Produce case studies of existing software employing procedural generation of 2D and 3D environments. Identify techniques which may be studied and expanded upon further within the project.

My investigation into existing research has revealed a deep history of research into procedurally generated environments dating from over thirty years ago. These have been added to the section on References.

November, Week 4

I have developed a number of basic prototype applications demonstrating some of the basic techniques I will explore in my report. I have also begin formulating a model structure that will allow me to specify a 'combination operations' model linking together many different structures to create a complex terrain of a specific type.

December, Week 1

Prepare project presentation for delivery in subsequent week. Presentation to include discussion of case studies in progress up to this point, and any prototypes developed.

I have prepared the presentation.

December, Week 2

Deliver project presentation to staff and students.

I have delivered the presentation to the project supervisor and other members of staff.

December, Week 2 to January, Week 2

Formalise research and prepare interim report for submission on or before 11th January 2009.

I have been in communication with the project supervisor. He has confirmed that there is no need for an interim report (in contradiction of some of the project guideline documents). No interim report was produced.

January, Week 2 to March, Week 2

Software implementation of prototypes designed in the previous year. Alterations to previous designs will be documented and discussed.

I have begun implementing the composite model structure discussed previously. I will experiment with combining different stages to try and recreate a specific scenario.

March, Week 2 to April, Week 2

Preparation of report, to include evaluation of software prototypes developed during the course of the project. Evaluation will include quantitative comparison between procedural content generation methods and manual content production methods. Suitable areas for further exploration will be identified, and possible methods of expansion will be discussed.

I have composed a thorough report detailing my experiences in attempting to recreate an 'island' environment using procedural generation techniques. I believe I have been successful: the islands sufficiently resemble the specification I have proposed, and my model is fast enough to allow for an exploration of the parameter space to show how the model can be adapted to create environments with different characteristics.

April, Week 4

Project submission date: 22nd April 2009

Project Submitted

Chapter 11: TABLE OF FIGURES