**Exploration of Procedural Terrain Generation on the GPU**

Joshua Frost

*School of Computing Science, Newcastle University, UK*

**Abstract**

Write your abstract here. This should be a concise summary of what your project has been about and what you aim to show. 9pt text here.

*Keywords: list the main concepts used in your project, e.g. the research area and techniques used.*

1. **Introduction**

The use of procedural generation within games has become increasingly common in recent years, with the growing popularity of open world titles and demand for larger scale experiences necessitating its adoption. Depending on the genre, scale could simply mean a larger playable environment, or in a more linear experience it could mean higher quantities of levels or variation of existing levels. Procedural generation can be used as a tool to address all of these scenarios, and can be used in many forms from automatically generating entire terrains to providing a brush tool which can ‘paint’ features or geometry onto an existing landscape. Once a suitable tool is created it can save huge amounts of development time by either optimising the workflow of designers and artists, or by removing the need for manual human input at all. This allows developers to meet the growing expectations of players, and create rich experiences with vast quantities of playable content within a realistic timeframe.

This paper will focus on procedural terrain generation, which is the process of creating terrain using algorithms and functions rather than by manually designing and shaping it. This system of creating environments is inherently much more scalable in both size and quantity, with the main limiting factors being the computation required and the quality of the results produced. If a suitable system is developed for the specific application, then extremely high quality terrain can be generated in the desired style. There are several popular algorithms which are frequently used as the basis of these systems, and this paper will explore how these algorithms work as well as how we can take advantage of their parallelisable nature to reduce computation time.

This will be done by using the GPU, which is capable of performing large quantities of floating point calculations in parallel due to its inherent architecture and high core count. The GPU was initially introduced to accelerate computer graphics, but was quickly adopted for other tasks which involve highly parallel problems. Procedural terrain generation is one such task, as the methods being explored work by computing a height value for each vertex of the terrain independently of one another using functions which can therefore be called in parallel.

OpenGL is an application programming interface (API) which primarily interacts with the GPU for hardware-accelerated rendering of vector graphics. Originally released in 1992, OpenGL has undergone a number of incremental additions which have expanded its functionality. OpenGL 4.3 was released in 2012, and implemented compute shaders among other features. Compute shaders form an additional shader stage which is used for computing arbitrary information generally not related to rendering. They allow the power of the GPU to be used for other types of computation, and can therefore be used for executing the procedural generation functions on the GPU in parallel.

The core difference between compute shaders and other shader stages is that they have no user-defined inputs or outputs; they must fetch any input data themselves and output by writing to an image or shader storage block. They do not form part of the rendering pipeline, where each stage has well-defined inputs in order to perform the required operations at that step. Instead, compute shaders operate in an abstract space, and are executed via a function which defines the number of work group executions in three dimensions to dictate how many total shader invocations there will be. A work group is the smallest number of shader invocations that the user can execute, defined within the shader by a three-dimensional local size. For example, if a shader has a local size of (8, 1, 1) and is executed with a work group count of (2, 4, 8), then there will be 512 total shader invocations across 64 work groups. Each invocation is uniquely identified by its global invocation ID, and all invocations within a work group will be executed in parallel on the cores of the GPU.

This paper will begin with an overview of popular procedural terrain generation techniques, before detailing a CPU-based implementation of two methods. The same methods will then be implemented in compute shaders to make use of the architecture discussed, so that the difference in computation time can be analysed. This will facilitate the evaluation of the potential performance benefits of utilising the GPU for terrain generation.

***Aim and Objectives***

The overall aim of the project is:

* To investigate procedural terrain generation techniques, and how the GPU can be used to generate large-scale terrain more efficiently.

To achieve this goal, a number of objectives have been created to represent sub-tasks that enable its fulfilment:

* Analyse the algorithms and techniques used for procedural terrain generation.
* Implement CPU-based procedural terrain generation using two different noise techniques.
* Implement GPU-based procedural terrain generation using two different noise techniques.
* Evaluate and compare the performance of each implementation as scale and complexity increases.

1. **Background Research**

***Noise Functions***

In order to procedurally generate planar terrain, a heightmap is typically created which stores an elevation value for each vertex in a single-channel greyscale image, with black representing minimum height and white representing maximum height. This heightmap can then be applied to a flat plane to create the terrain, by translating each vertex by the corresponding elevation value from the image. For the terrain to appear realistic and natural, a smooth distribution of height values is required. This means that completely random ‘white noise’ would not be suitable, and a function which can provide an output where nearby values are typically similar must be used instead.

Noise functions are real-valued functions that vary between 0 and 1 over some domain, which can be defined for any number of dimensions to be suitable for the given application. Each value is generated by determining a pseudo-random number within the domain, and the output values will form a relatively smooth distribution which makes them appropriate for terrain heightmaps. For terrain generation 2 or 3 dimensions are typically used in order to map to the input coordinates of vertices on a plane or sphere, and the project will cover several types of noise that are generated via different methods with a focus on 2 dimensional implementations. While the resulting terrain will be a 3-dimensional mesh, it is simply a 2D plane with a vertical offset applied to its vertices, therefore this use of noise functions is 2-dimensional in nature.

***Value Noise***

Value noise works by setting an arbitrary grid across a surface, and assigning a random value to each grid point using a random number generator [2]. In most cases a deterministic generator function is used, so that these values can be calculated independently of one another in real time using the grid coordinates as input. The value of each pixel between the grid points is calculated by interpolating between the nearest *n* grid point values, where *n* is the number of dimensions of noise squared. These nearby grid points are found by using a floor function to round the local pixel coordinates up and down to integer values. Almost no memory is required for value noise provided that a deterministic random number generator is used, and it is a relatively fast noise function due to its simplicity. Despite this lack of complexity value noise is not as efficient as other techniques, particularly in higher dimensions. It also provides less control over the frequency and amplitude of noise, making it harder to fine-tune the output to achieve a specific result. These qualities mean that despite being a fundamental technique which contributed significantly to the field of procedural generation, value noise is not commonly used at this point.

The main computation involved is the interpolation between grid point values, so naturally the function used will greatly affect performance. Linear interpolation (LERP) provides fast but lower quality results with jagged lines in the noise, while cosine interpolation is slower with rounded results, and cubic interpolation provides the best results but is the least performant. The interpolation method used must be scaled to the number of dimensions of noise, so for this 2-dimensional application bilinear interpolation could be used for the fastest results. This applies LERP in 2 dimensions by linearly interpolating in one direction before applying LERP again between the results to give the final value.

[ image ]

***Diamond-Square Noise***

Diamond-Square noise also initialises a grid, but only assigns random values to the corner points. To assign values to the rest of the grid, the following steps are alternated between:

* Diamond step: for each square in the grid, the midpoint value is set to the average of the four corner points of the square plus a random displacement within a given range.
* Square step: for each diamond in the grid, the midpoint value is set to the average of the four corner points of the diamond plus a random displacement within a given range.

[ image ]

Each iteration halves the step size to use smaller diamonds and squares, and reduces the displacement range so that smaller changes are made at finer detail levels. The algorithm produces high quality noise despite being extremely fast and simple to implement. It produces fractal-like results, meaning at different scales it has self-similar properties, which is also characteristic of real-world natural terrains. However due to the dependence of each grid point on its neighbours, values cannot be computed in parallel and the entire grid must be stored in memory so that nearby values can be accessed at each step. It also suffers from requiring a fixed grid size that is a power of two plus one due to the way in which the algorithm iterates, which makes it less flexible for generating heightmaps of different sizes. Additionally, the output resolution will always be fixed by the initial grid size, which can be problematic for making level-of-detail adjustments in real-time rendering situations. As a result, it is not a popular choice for procedural terrain generation.

***Perlin Noise***

Perlin noise builds upon the foundations of value noise to provide smoother, more natural-looking results with greater control and computational efficiency. It is the most well-known and popular algorithm for procedural generation [7], and many still consider it to be the industry standard despite its age and the subsequent release of an improved algorithm by creator Ken Perlin. It uses the same arbitrary grid and value calculation via interpolation as value noise, but instead of assigning a value to each grid point it assigns gradient vectors. For 2-dimensional noise, these are typically unit vectors pointing in random directions. For a given pixel within the grid, distance vectors are computed to nearby grid points, and then the dot products between the grid point gradient vectors and their distance vectors are calculated. These dot product values represent the contribution of each grid point to the final noise value, and are interpolated between for each dimension similarly to value noise. A fade function is applied to the local pixel coordinates before interpolating to ensure smooth transitions between values. A common fade equation is the quantic curve:

***f(t) = 6t^5 – 15t^4 + 10t^3***

Rather than using a random number generator as in value noise, Perlin noise typically uses a permutation table of size *n*, containing the numbers 0 through *n-1* which are randomly shuffled. This ensures the output is deterministic, and the resulting use of indexing allows quick lookup within the precomputed array rather than generating new random values. The permutation table also ensures a uniform distribution of gradient vectors, and allows for tileable noise by repeating permutation table values in a way which aligns gradients at the tile edges to create seamless terrain.

1. **Design and Implementation**

After gaining a sufficient understanding of noise functions and their application to terrain generation, the designing of the software implementation could begin. The decision was made to use a framework which had been made available through the university by Dr Richard Davison, which provides a functional C++ implementation of OpenGL with core features such as window creation and shader loading. This enables the focus of the project to be directed on the noise implementation and GPGPU programming. The terrain generation code would be written in C++ and GLSL, and build upon the foundations provided by the framework. In order to visualise the resulting terrain, the code would inherit from the provided mesh class and make use of the existing rendering implementation.

Following the conduction of the background review, the decision was made to implement Perlin and Simplex noise as they are the most popular and high-quality algorithms within the scope of the project. Each approach would be implemented in C++ to be run on the CPU, and also in GLSL within a compute shader to make use of the GPU. The time taken to generate terrain with different properties would then be measured for each approach, in order to evaluate the potential benefits of using the GPU for this task.

***Perlin Noise Class***

The implementation phase begun with creating a 2D Perlin noise class within C++ using the techniques discussed in the background review. A permutation table of size 512 was initialised with the values defined by Ken Perlin, a randomly arranged set of the numbers 0-255 inclusive repeated once in the same order to avoid overflow.

Three helper functions were then implemented – a simple linear interpolation function, a fade function using the equation discussed, and a gradient function which is based on Ken Perlin’s 3-dimensional method [cite his paper]. For 2D noise 8 gradient vectors are used instead of 12, which was padded to 16 for easier division. These 8 vectors represent the directions from the centre of a square to each of its vertices and edges. This means that the gradient function can take the first 3 bits of the hash value, rather than 4 as in 3D. This effectively limits the hash value, which is obtained from the permutation table, to an integer between 0 and 7 representing the 8 gradient vectors. This value determines the gradient direction in order to assign values to the variables *u* and *v*, which represent the components of the distance vector from the grid point to the given pixel being computed. If the hash value is less than 4, *u* represents the horizontal component and *v* represents the vertical component, otherwise these values are swapped. This ensures that the function covers all 8 directions mentioned.

[ snippet of grad function ]

Finally, the dot product of the gradient vector and the distance vector is calculated and returned by conditionally negating the values of *u* and *v* as a representation of positive of negative gradient vector values. If the least significant bit of the hash value is 0, then the gradient vector *x* component is positive and therefore *u* remains positive, otherwise it is negated. The same operation is applied using the second least significant bit for the *y* component, and the results are added in accordance with dot product calculation.

For example, should the gradient function be called with a hash value of 220 from the permutation table, then performing the modulo operation gives a value of 3. As this is less than 4, *u* is set to the *x* value and *v* is set to the *y* value. The gradient vector being used is (1, -1), and the dot product expression therefore evaluates to *u – v*, which in terms of coordinates is *x – y.*

Next, a ‘Perlin’ function was written to make use of these helper functions in order to calculate a Perlin noise value given 2-dimensional coordinates. It determines which grid square the point is in by using a floor function to round its coordinates, which enables the four corners of the grid square to influence the output value using the gradient function described. The local coordinates of the point within the grid square are also calculated to provide the *x* and *y* inputs of the function.

[ snippet of perlin function ]

The coordinates of the four square corners are masked with 255 in order to wrap them around the fixed size permutation table of 256 values. They are then hashed using the permutation table, to ensure the gradient vectors used are pseudo-random but repeatable. These hash values are inputted into the gradient function along with the local coordinates, and the four outputs are bilinearly interpolated between to get the final noise value at the given point. The value to interpolate by on each axis is given by calling the fade function on the local coordinates, in order to ease the transition between different gradient values and smooth the final output.

***Simplex Noise***

Next, the noise class was expanded to include an implementation of 2D Simplex noise. This made use of the same helper functions and permutation table, but works with triangles rather than squares as that is the shape of a 2-dimensional simplex. This adds additional steps to the function, as skewing transformations must be applied to the input coordinates to transform them from grid space to simplex space and vice versa. The skewing factors *F2* and *G2* are defined at the start of the ‘Simplex’ function, and then applied to the input coordinates to determine which simplex they fall within. The simplex coordinates *i* and *j* are then converted back to grid space using *G2* to find the distances from the unskewed simplex origin to the input point.

[ snippet of simplex p1 ]

After this step, the *x* and *y* coordinates are compared to determine if the point is within an ‘upper’ or ‘lower’ triangle, so that the relative positions of the other two simplex points can be computed. Once the *x* and *y* coordinates of each simplex vertex relative to the input point have been found, their gradient indices and contributions can be calculated.

[ snippet of simplex p2]

This computation is done by using the permutation table to select a gradient vector, similarly to in Perlin noise. The simplex coordinates *i* and *j* are masked with 255 and used to find the gradient hash value of the first point, then of the other two with offsets applied. The same gradient function as Perlin noise is used to compute the dot product of each gradient vector with its distance vector, however some additional steps are required to compute the contribution of each gradient using radial attenuation rather than linear interpolation. Firstly the squared distance from the input point to the corner is calculated and subtracted from 0.5 to determine whether the input point is within a unit circle radius of the simplex corner. If the result is negative, the point is outside of the range and is therefore not influenced by the corner so its contribution is zero. If the result is positive, a polynomial weighting function is applied by raising it to the fourth power in order to ensure smooth transitions between gradient vectors similarly to the Perlin fade function. This value is then multiplied by the corner’s dot product found using the gradient function to give its final contribution. The contributions of each point are summed, and scaled by 70 to provide values in a useful range as suggested by creator Ken Perlin.

***CPU Terrain***

After both noise functions had been implemented in C++, they could be used to generate elevation data for a terrain heightmap. A CPU-based approach was developed first, which uses the ‘mesh’ class provided within Dr Davison’s framework as a foundation. This class handles OpenGL calls for drawing the mesh, uploading buffer data and other fundamental operations. In order to create a terrain mesh, vertex position and triangle index data must be generated and stored in the appropriate arrays before being buffered. The ‘CPUTerrain’ class takes in a size parameter which determines the width and height of the terrain, and dynamically allocates arrays that are sized based on these dimensions. A scale is set for enlarging the mesh on the x and y axes, and the inverse is applied to the texture coordinates.

Each vertex has an integer x and z coordinate between 0 and the width or height, effectively creating a uniform plane. The y coordinate value, or elevation, is given by sampling a noise function using the x and z coordinates with scaling and transformation applied first. A series of variables are applied to the noise sampling, in order to provide flexibility and the ability to fine-tune the output to create the desired result. These parameters are applied regardless of which noise function is used, and the impact of each will be briefly explained below.

[ explain each parameter ]

As is evident in the code snippet, this CPU approach loops through each vertex sequentially to calculate its elevation value and store it in the array. This highlights the bottleneck of this approach, as the generation could clearly be executed faster should multiple vertices be computed simultaneously using parallel operations. The GPU approach will address this problem, and the expectation is that it will reduce total computation time.

Once the vertex data has been computed, indices are calculated and stored so that the terrain can be rendered using triangles. The normals and tangents are also calculated using provided functions, and all of the computed data is uploaded to the relevant buffers. The terrain is then rendered within a simple demonstration scene as pictured below.

[ pic of demo scene ]

Then use of noise in cputerrain.cpp – different parameters with explanation of effect, mesh creation, loops and how they show its parallelisable

Then explanation of GPU coding + differences + challenges etc, breakdown of computeshaderterrain.cpp and texture stuff

1. **Evaluation**

This section might alternatively be called “Experimental results”. It describes how you assessed your project and what you found out. Where you give results in tables or graphs, remember to highlight in the text the key points that the data shows and, if possible, try to explain why you got any unexpected results.

1. **Conclusions**

In this section you summarise what you did and discovered and (importantly) what else you would have done (or done differently) if you had the chance. It is a chance to reflect on your success.

1. **Acknowledgements**

If anyone else helped you, e.g. by providing code or data, then you should acknowledge their contribution here. Avoid long tributes or anything too personal and try to stick to acknowledgements which are specific to the project. You do not have to acknowledge your supervisor or second marker; they are just doing their jobs and acknowledging them or not will not affect your mark.

In the references below you will give all the references (in 9 pt text) for papers you have cited in the text above. Ensure that all references are complete and consistently formatted. Make sure that web pages include the date you last accessed them. In general it is good practice to use a bibliography support tool such as Endnote (for Word) or BibTeX (for LaTeX) to compile your references, but it is not essential.

Section headers numbered and are 14pt bold with 12pt spacing above and 10t spacing below the heading. Body text is 11pt and fully justified (text aligned to both margins). Left margin 4.1cm, right margin 3.5cm, top margin 3cm and bottom margin 5cm. The page header should include module code and page footer includes page numbers only. The first paragraph of each section is not indented; subsequent paragraphs are indented by 0.5cm. Line spacing for general text is 1.15 and there is no additional spacing between paragraphs within a section.

***Aim and Objectives***

The overall aim of the project is:

**References**

[1] Thomas, N., *Formatting guidance for CSC8498 final project report*, School of Computer Science, Newcastle University, 2016.

[2] [www.entcs.org](http://www.entcs.org), last accessed 01/03/2016.