Analysis of the Marching Cubes Algorithm for Terrain Generation

Ethan Younger Banks

School of Computing Science, Newcastle University, UK

Abstract

Marching Cubes is an algorithm first developed in 1987 and applied to medical visualizations such as MRI and CT scans. Another application of the algorithm exists in 3D terrain generation for video games. The additional access to the GPU outside of the rendering pipeline afforded to modern programmers in the form of Compute Shaders could allow a robust GPU-based implementation of the marching cubes algorithm with performance far beyond what the CPU is capable of on a single thread. This project explores the possible applications of Marching Cubes in generating 3D procedural terrain for video game applications. Specifically, comparing the performance of a CPU based approach vs a GPU based approach, in generating 3D terrain maps suitable for use in a video game. Performance analysis of each approach revealed that the GPU far outperformed the CPU for large maps, but the CPU was more efficient for much smaller sizes. Neither approach was fast enough to perform generation of high-fidelity terrain in real-time during gameplay. This suggests that while large scale terrain generation using this method is limited to loading screens, utilising the GPU could greatly reduce these loading times, and smaller modifications to the terrain could be performed in real-time by the CPU.

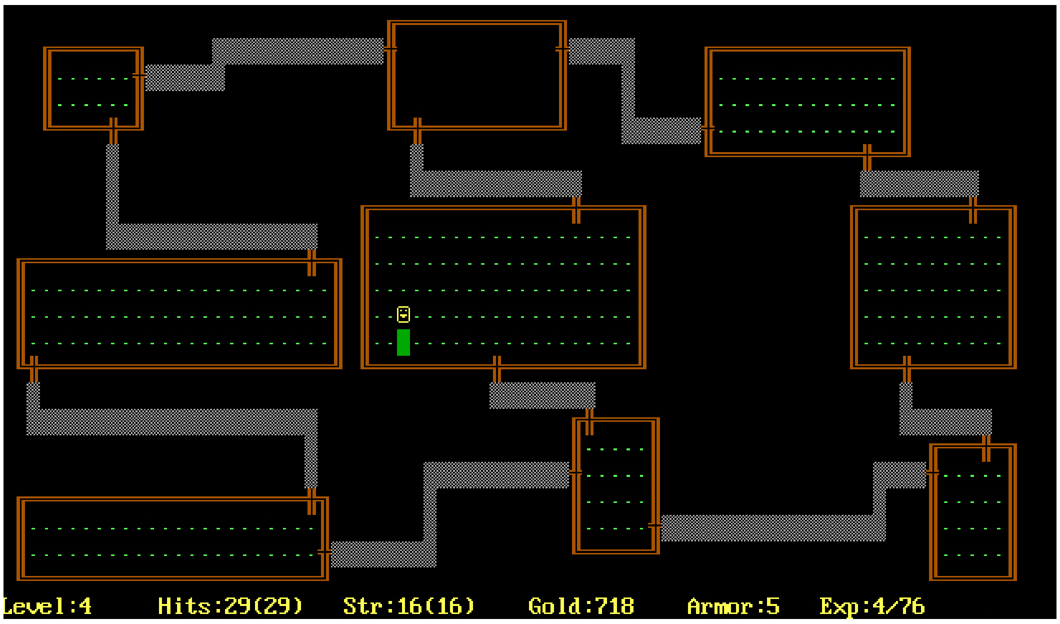
Keywords: Marching Cubes, Graphics, Procedural Generation, Compute Shader

1 Introduction

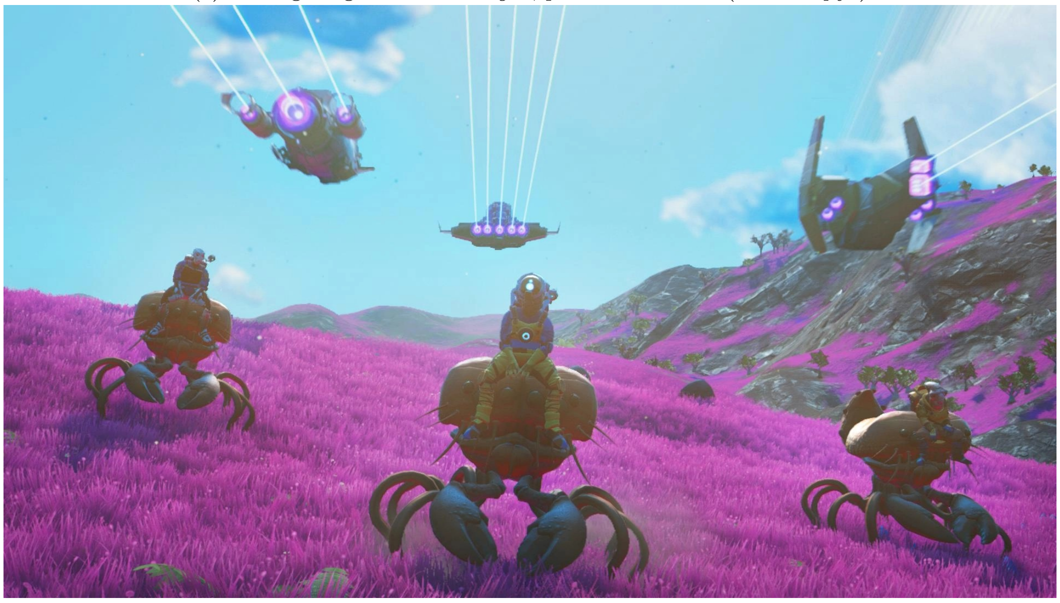
1.1 Context

Procedural generation methods are ways of generating content without the direct input of a designer or artist. Procedural generation techniques can be used for sound effects, level design, or in this case, generating terrain, and form a vital foundation for a wide variety of games. Procedurally generated levels can produce a functionally infinite amount of variety and replayability for relatively low development time and give rise to an entire genre of “roguelike” games. [1] Although traditionally the roguelike genre has been limited to relatively simple 2D games (such as *Rogue* itself), in recent years it has been demonstrated that procedural generation of three-dimensional worlds can be effectively performed. Outside of the roguelike genre, procedural generation can be seen in “infinite exploration” games, in which the game world is generated in real-time as the player moves through it, either continuously to produce the illusion of a single infinitely large world, or in sections such as generating a planet at a time as seen in title such as *No Man’s*

Sky and *Astroneer* [15]. As illustrated in Figure 1, procedural generation in video games has come a long way since its first introduction. Whether real-time or chunk by chunk, procedural generation of 3D terrain is being used increasingly in popular video games, with a wide variety of generation systems producing both realistic and fantastical worlds through a wide variety of methods.



(a) A dungeon generated in *Rogue*, published in 1980 (credit: *Epyx*)



(b) A planet and creatures generated in *No Man's Sky*, published in 2016 (credit: *Hello Games*)

Fig. 1. One of the earliest gaming applications of procedural generation, compared to a more recent implementation

Procedural generation is not just used in games where the world is generated on the fly - even in a bespoke hand crafted world, procedural generation can be used to make production of large sections of landscapes far quicker and even more realistic than a hand-crafted approach may achieve. The terrain of a game world can be procedurally generated, and then specific features added and tweaked by artists to produce the finished world. This technique is widely employed in large open-world games (see Figure 2).

Fig. 2. Important features like this fortress need to be built and placed by hand, but the surrounding landscape could be procedurally generated (credit: *Ubisoft*)

1.2 Problem Space

As the scope of new video games continues to grow, development time is at a premium. Procedural generation offers a way to reduce this time cost, while a well designed and implemented system will still offer the player a high-quality gameplay experience. In order to achieve these benefits, new and innovative systems are always needed for generating terrain that is more interesting, more consistent, or (as is the focus of this project), more performant. The performance of a terrain generation system will be the deciding factor in multiple design decisions - most importantly, whether the game world can be generated in real time during gameplay, or whether it needs to be pre-generated during a loading screen. A generation system which can run alongside gameplay while maintaining a consistent framerate opens up a lot of gameplay possibilities which cannot be achieved with a slower, more resource-heavy system.

Terrain generation almost invariably begins with a *noise function* - an algorithm which generates pseudo-random numbers - which are then used to inform the terrain generation algorithm in some way. Many commonly used terrain generation systems use a *heightmap*, which simply takes the values output from the noise function and applies them as y-axis values to an x/z plane. This can produce very convincing terrains, partly due to the fact that heightmaps can be generated from real-world locations and applied directly to a simulated landscape (see Figure 3). Even when generated from a noise function, heightmaps produce convincing results (see Figure 4) but come with one major caveat: since terrain is modelled as a single plane deformed on the y-axis, there can be no caves or internal features in the generated terrain. Caves or overhangs can be added after the fact by hand, but a game which needs to involve many cave systems or overhanging terrain (see Figure 5) will not get much utility from a heightmap terrain generator.

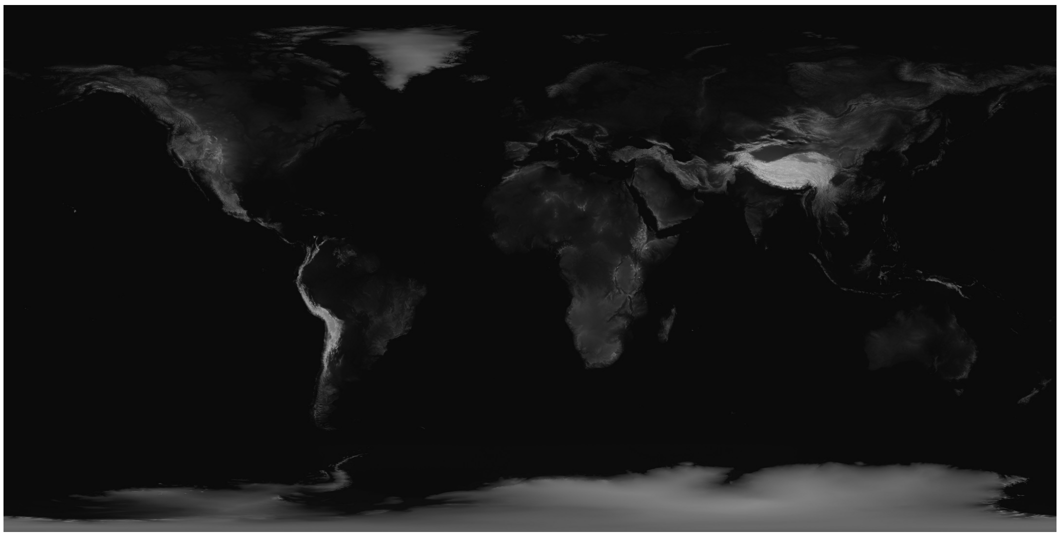
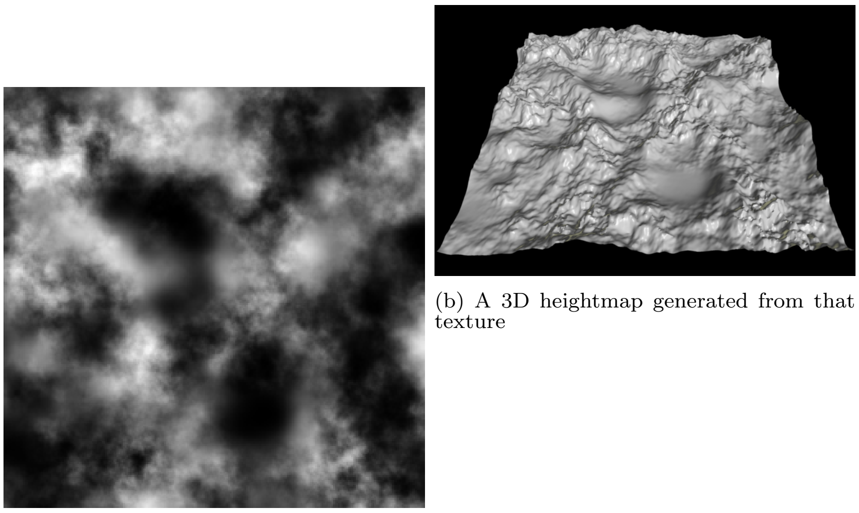


Fig. 3. Even this heightmap of the entire planet Earth could be applied to a plane for use in a video game (credit: *NASA*)



(a) A 2D texture output from a noise function

(b) A 3D heightmap generated from that texture

Fig. 4. Heightmap based terrain generated by *Terragen* (credit: Planetside Software [14])

Other terrain generation solutions exist outside of heightmaps, but are often a combination of many different steps, first generating the outer shell of the terrain, and then generating cave systems and more interesting features on top of the heightmap. An example of this can be seen in *Terraria* and is analysed in an article by Joshua Tippets [18]. This multi-layered approach is capable of producing good results, but is slow, and this is demonstrated by the fact that *Terraria*'s world is generated during a loading screen at the beginning of the game, rather than in

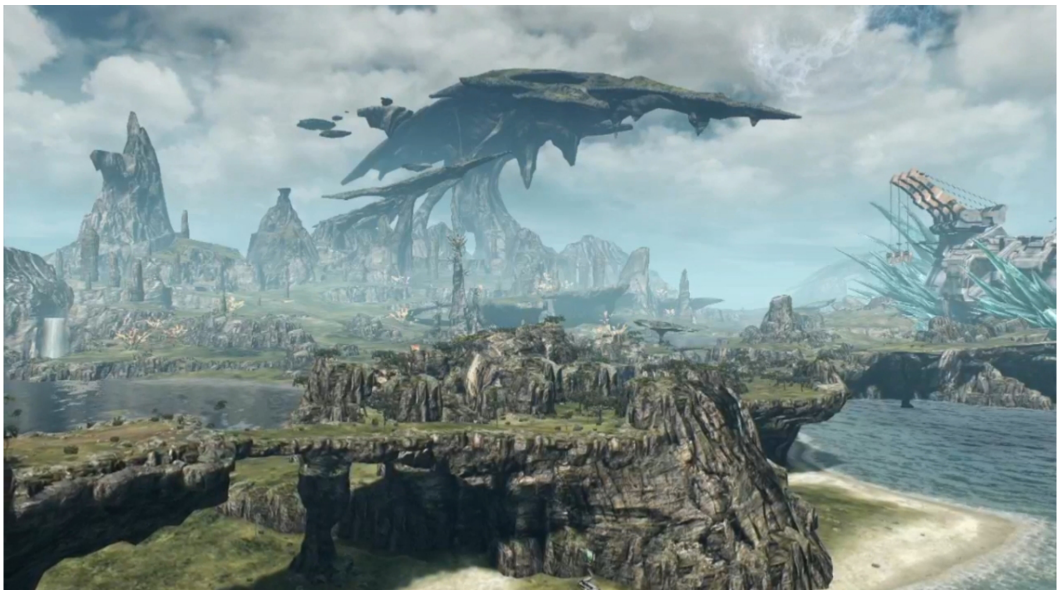


Fig. 5. These visually striking overhangs could not be generated by a heightmap-based approach (credit: *MonolithSoft*)

real-time as the player explores. An all-in-one algorithm which can generate terrain with overhangs and internal features in a single pass would bring major benefits to performance and ease of use, and one such algorithm exists in *Marching Cubes*, an algorithm first developed for medical applications as long ago as 1987.

The use of the GPU for terrain generation is largely unexplored in commercially published games, although academic examples such as in *GPU Gems 3* [2] have demonstrated that this idea does have potential. Utilising a massively parallelised approach to terrain generation could greatly speed up generation time which could in turn provide players with shorter loading times or more seamless exploration.

1.3 Project Aim and Objectives

1.3.1 Aim

The aim of this project is to investigate the benefits of a GPU based marching cubes implementation using a compute shader, compared to a single-threaded CPU based implementation, when generating large terrain maps suitable for use in a video game.

1.3.2 Objectives

1: To investigate existing terrain generation methods

By investigating currently used terrain generations systems, a background can be established as to what challenges are faced by modern games, and what improvements must be made to ensure the utility of a newly developed system. An understanding of how existing systems operate is also essential as a starting point for any new development.

2: To investigate existing use cases of the Marching Cubes Algorithm

An understanding of how the marching cubes algorithm is implemented and applied in existing systems can be carried over and applied to the utilisation of the algorithm in this terrain generation scenario.

3: To develop a system for generating 3D terrain using the Marching Cubes algorithm

With an understanding of existing solutions, and a knowledge of the improvements that must be made, a system can be developed which utilises the strengths of the marching cubes algorithm to fix some of the problems with existing terrain generation systems.

4: To assess the performance of the marching cubes terrain generation system when using a CPU vs GPU based implementation

The idea of using the parallelisation offered by the GPU in terrain generation is intriguing, and has the potential to improve performance significantly, since each cube of the marching cubes algorithm can be solved independently of any other. The addition of Compute Shaders to OpenGL is a relatively new development, having been introduced in version 4.3 in 2012, and while the technology has seen widespread use in ray-tracing applications, a less graphics-focused implementation such as this is not so well explored.

2 Background and Research

2.1 The Marching Cubes Algorithm

The Marching Cubes algorithm was developed by Lorensen and Cline, first published in SIGGRAPH in 1987 [9]. Marching Cubes was designed to be used in medical applications, as a way to improve the images produced by MRI and CT scans. Marching Cubes takes a 3D grid of data points (in the MRI application, this grid is built up out of 2D "slices"), each containing their co-ordinate position within the grid, and a "surface value boolean". This value represents whether the point lies inside (beneath the surface) of the object, or outside (above the surface) of it. Using the "divide and conquer" nature of the algorithm, these nodes are analysed in groups of 8, forming a cube in the modelled 3D space, with each cube being processed independently of any other. This means that for a grid of $x \times y \times z$ nodes, there will be $(x - 1) \times (y - 1) \times (z - 1)$ cubes to analyse (see Figure 6). A cube consists of 8 points, each of which could be inside or outside of the surface, and the goal of the algorithm is to produce a set of triangles within each cube which satisfies the surface boolean values of its points. This is achieved by placing the points of the triangles on the edges of the cube. With 8 points, each with one of 2 possible values (inside or outside of the surface), there are 2^8, or 256 possible arrangements of triangles that any given cube could generate. As Lorensen and Cline point out, every one of these cases need not strictly be considered - half can be removed due to the symmetry of inside/outside points. That is to say, if the surface boolean value of every point of a cube is inverted, the surface remains the same (the surface of a donut is the same as the surface of that donut's hole). Using the reflectional and rotational symmetry of the cube, these 128 cases can then be further narrowed down to only 15 possible cubes (see Figure 7). This reduction is useful for human understanding of the algorithm, but in a computer application, it is more computationally efficient to simply use the full set of 256 values. In this case, the surface boolean values of the points of the cube can be bitmapped to an 8-bit integer, which then acts as an index into the array of cube cases (see Figure 8).

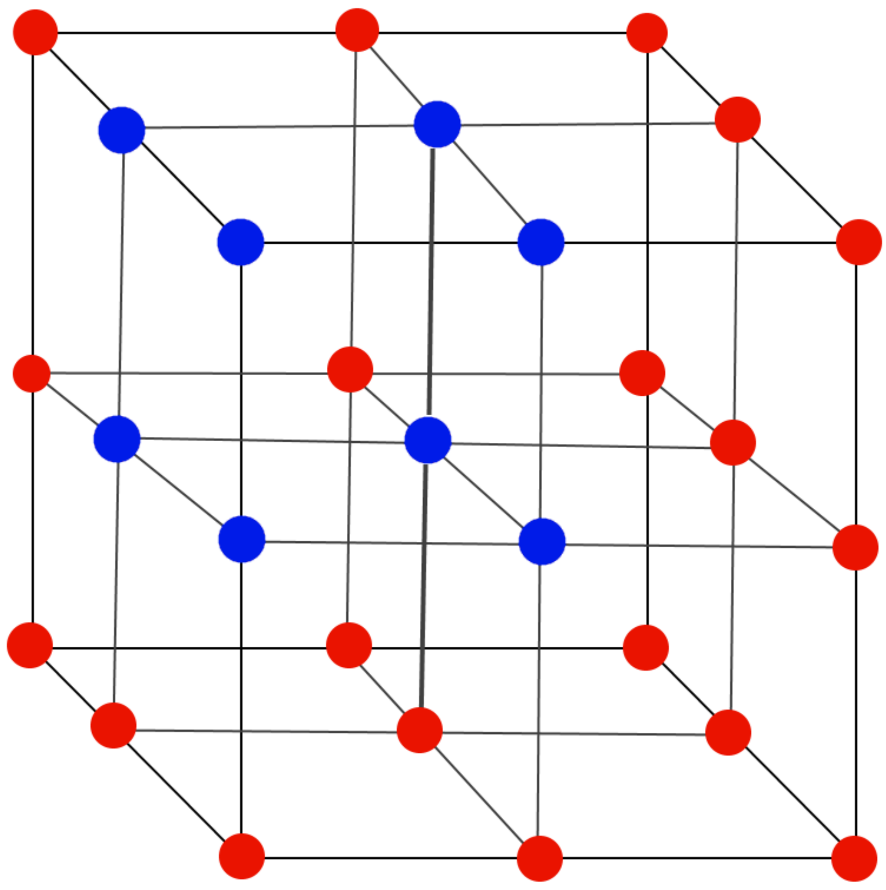


Fig. 6. A grid of 3x3x3 points is a grid of 2x2x2 cubes

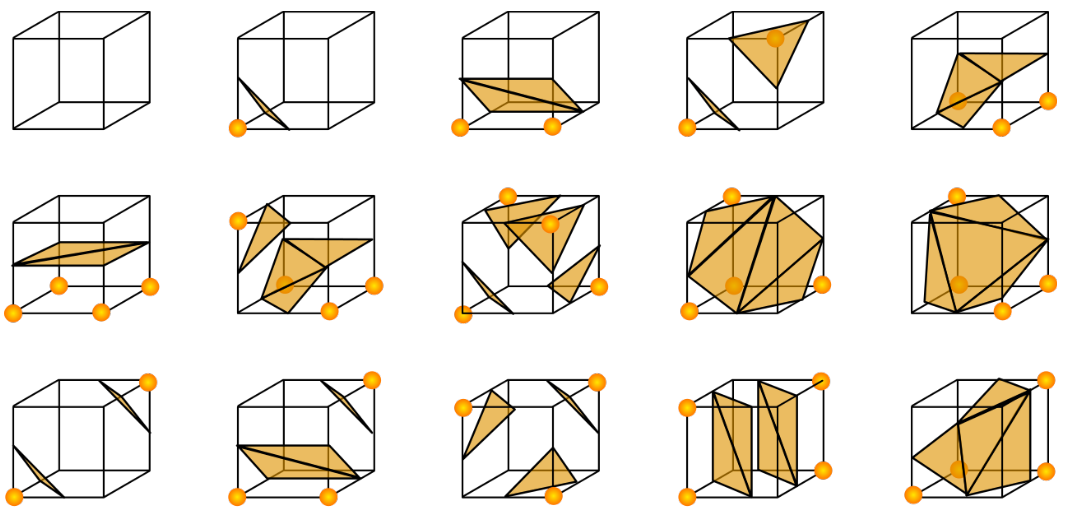


Fig. 7. The 15 possible cube configurations, all other cases are reflections or rotations of these, to produce 256 total cases (credit: *Jean-Marie Favreau - http://jmtrivial.info*)

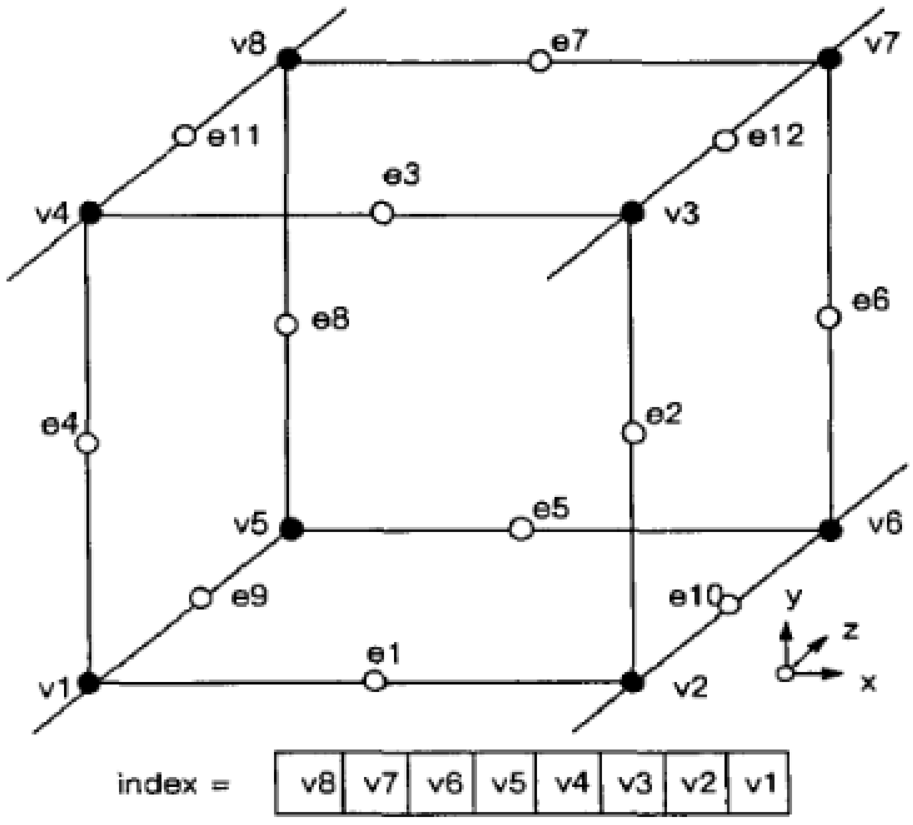


Fig. 8. The surface boolean value of each corner corresponds to a bit in the cube case index [9]

Lorensen and Cline outline some improvements to the base algorithm, most interestingly the idea that, for cubes which straddle the surface of the object, the results of previous cubes can be used to effectively skip some of the edges of the new cube. As shown in Figure 9, nine of the twelve edges of any given cube have already been a part of a previously computed cube.

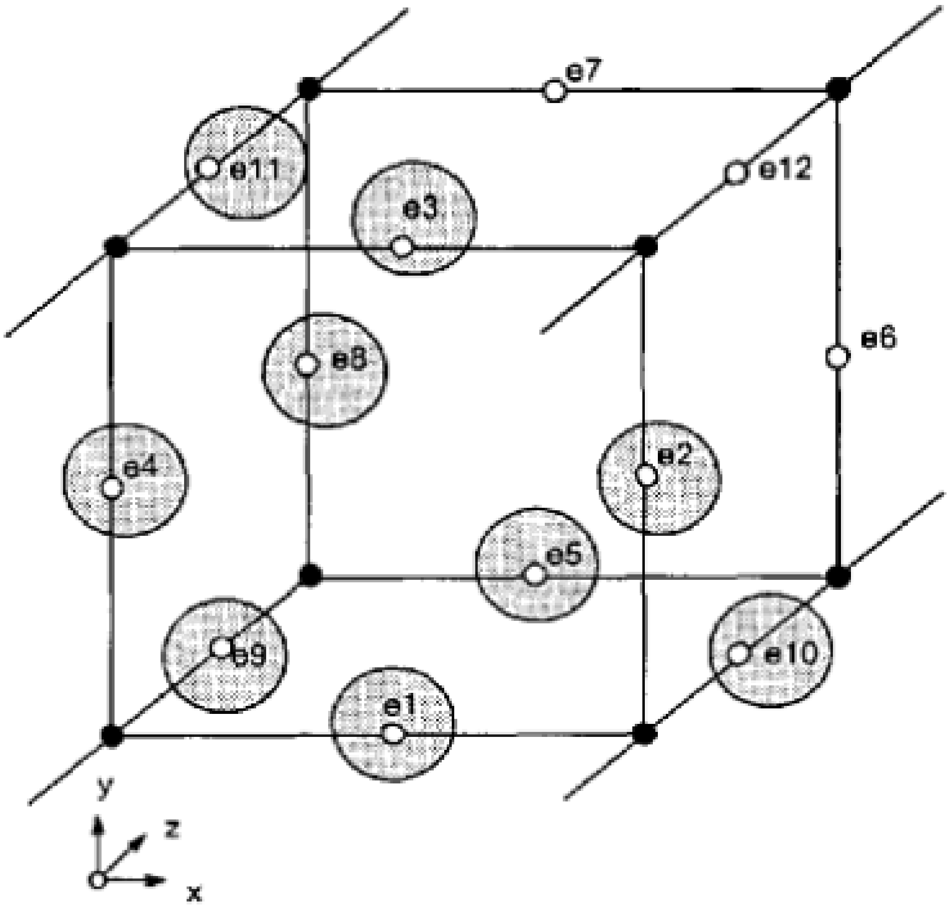


Fig. 9. The *coherency* of adjacent cubes [9]

Re-using these calculated edges reportedly speeds up the algorithm by a factor of three. This may seem like a very dramatic improvement, but when considering that three quarters of the edges of most cubes are being skipped by this optimisation, such a jump in performance is to be expected. This technique of re-using the results of previous cubes relies on the cubes being calculated in a set order, which, while true of Lorensen and Cline's implementation and of any single-threaded CPU approach, cannot be relied upon for a massively parallelised GPU based implementation. In essence, if the coherency-naive GPU based implementation is less than three times faster than the single-threaded CPU based implementation, then the CPU based implementation can be expected to perform faster than the GPU based implementation with this optimisation applied.

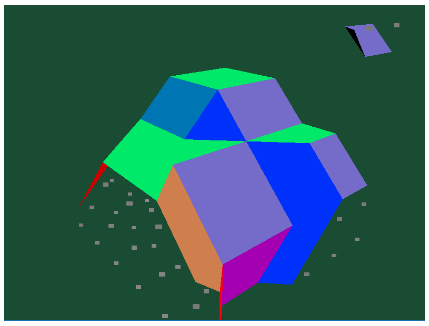
2.2 Utilising the GPU

The nature of the Marching Cubes algorithm immediately lends itself well to parallelisation - "solving" each cube is a relatively small, finite operation, and each cube can be solved independently of the results or order of any other cubes. This quality makes utilising the massive parallelisation of the GPU an obvious choice - why work through each cube in turn when you could compute all (or, at least, a large number of them), at once?

The well-known and popular book *GPU Gems 3* [2] details a very comprehensive implementation of marching cubes for terrain generation on the GPU. In chapter 1, Ryan Geiss describes a system which generates large procedural terrain by splitting the potentially infinite landscape into large "blocks" of terrain, which are then subdivided into the 32x32x32 grids used for the marching cubes algorithm. This technique of dividing a large procedural terrain into smaller, more manageable blocks can be seen in many large-scale procedurally generated games such as *Minecraft*.

In order to generate terrain, each node in the grid must be given its crucial *surface boolean value* but when the terrain is to be created from scratch, rather than modelled from real world data, this hard true/false divide is difficult to generate in any meaningful way. Instead, Geiss proposes using a noise function which generates floating-point numbers in 3D space, which he refers to as the *density function*. The surface of the generated terrain exists where the output of the *density function* is equal to zero, with negative values lying outside the terrain, and positive values inside it.

Geiss improves the "accuracy" of the rendered terrain by interpolating the points of the triangles along each edge of the cube, rather than simply using the midpoint of each edge. Because the noise function used to generate the weights of the nodes outputs a floating-point number instead of a boolean, this extra data can be used to the advantage of the generation system. The points at the start and end of the edge are given "delta" values, which are equal to the difference between the surface value (in this case, a constant 0.5) and that node's weight value. The surface point is then interpolated along the edge according to these delta values. For example, if the start point has a value of 0.0 and the end point is at 1.0, then the surface point will fall on the midpoint of the line as it does without interpolation, but if the values are skewed e.g. 0.9 and 0.45, the point will lie far closer to the end point at 0.45. This results in a less angular and much more convincing look (see Figure 10).


(a) Using midpoints

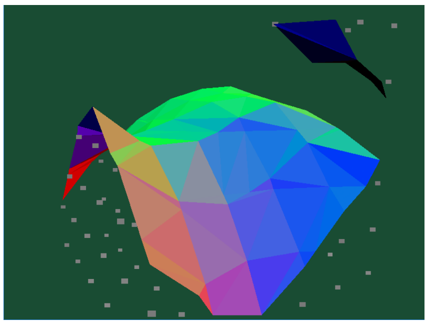
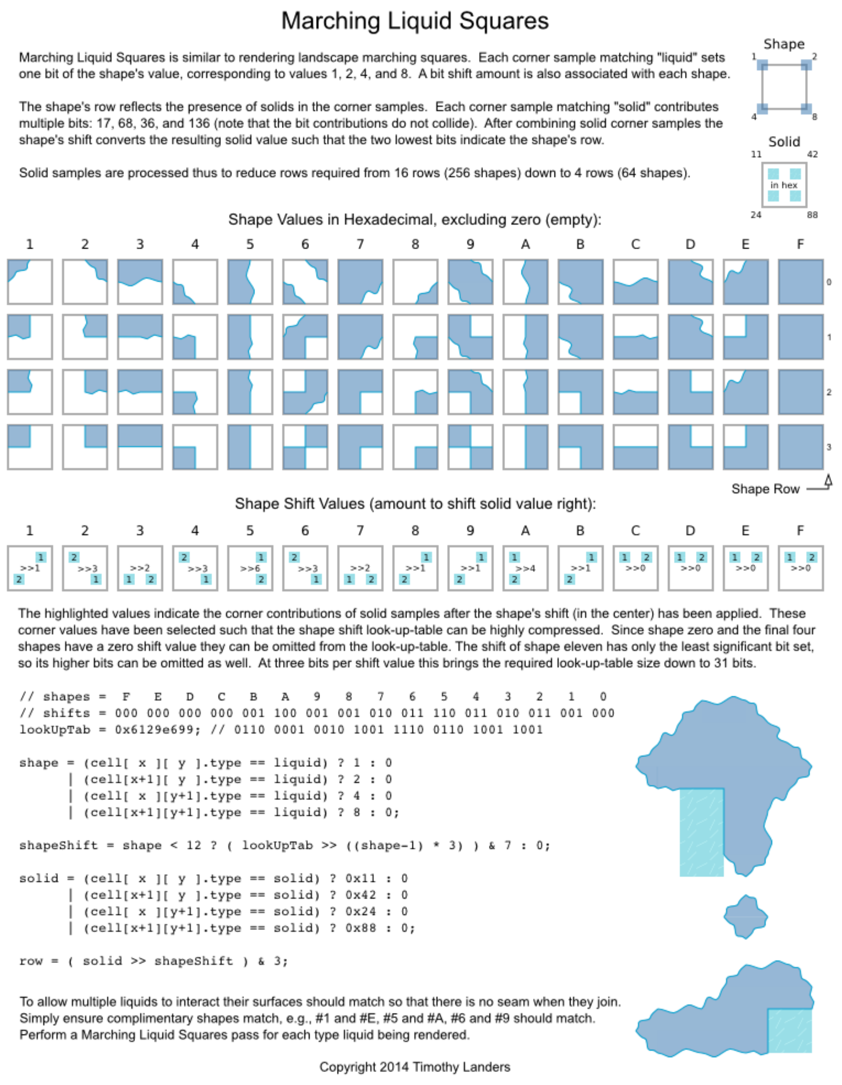

(b) Using linear interpolation

Fig. 10. The "smoothing" effect of linear interpolation is quite dramatic

GPU Gems 3 predates the introduction of compute shaders to the OpenGL standard, and instead demonstrates the use of a 3D texture to store the output of the density function, and a geometry shade for the execution of the marching cubes algorithm. This approach does appear sufficiently performant for real-time gameplay applications (although Geiss does not include any analysis of performance) but limits gameplay logic involving the generated terrain to functions which can also be executed on the GPU. Shader-based collision detection and AI functions are possible - Geiss describes a football colliding with the terrain, or flying AI agents navigating to avoid it - but a game involving a large number of complex physics calculations would benefit greatly from being able to use the CPU for these operations. The addition of compute shaders allows us to achieve a $CPU - GPU - CPU$ workflow, in which the GPU can generate the terrain, which is then accessible by the CPU for physics and gameplay calculations before finally rendering.

A key consideration when implementing a compute shader is how best to utilise *work groups*. Briefly, a work group is a collection of shader invocations. Work groups are modelled in three dimensions, and an individual invocation is identifiable by its xyz coordinate gl_LocalInvocationID within that workgroup, or gl_GlobalInvocationID in the set of all active invocations. [3] The maximum work group size and maximum number of work groups varies between GPU hardware and OpenGL driver implementation. For this project, the GPU in question is the AMD Radeon RX 480. The maximum work group size is 1024 total invocations, and the maximum number of work groups is 65535 in each dimension, for a theoretical limit of just over 67 million invocations. In practice, both the max workgroup size and max number of workgroups could not be filled at the same time, as there would not be enough VRAM (in this case, 8GB) to run that many invocations. A higher number of invocations results in better parallelisation, and so this project will maximise parallelisation by processing a single cube per invocation where possible. The relationship between different work group sizes with the same number of total invocations (for example, $3 \times 3 \times 3$ workgroups with 1 invocation each, vs a single work group with $3 \times 3 \times 3$ invocations), was also investigated. For this application, a larger work group size, with a smaller number of work groups, produced faster performance.


Shape
1
2
4
8
Solid
11
42
in hex
24
88
1 2 3 4 5 6 7 8 9 A B C D E F
0
1
2
3
Shape Row
1 2 3 4 5 6 7 8 9 A B C D E F
>>1
>>3
>>2
>>3
>>6
>>3
>>2
>>1
>>1
>>4
>>1
>>0
>>0
>>0
>>0

Marching Liquid Squares

Marching Liquid Squares is similar to rendering landscape marching squares. Each corner sample matching "liquid" sets one bit of the shape's value, corresponding to values 1, 2, 4, and 8. A bit shift amount is also associated with each shape.

The shape's row reflects the presence of solids in the corner samples. Each corner sample matching "solid" contributes multiple bits: 17, 68, 36, and 136 (note that the bit contributions do not collide). After combining solid corner samples the shape's shift converts the resulting solid value such that the two lowest bits indicate the shape's row.

Solid samples are processed thus to reduce rows required from 16 rows (256 shapes) down to 4 rows (64 shapes).

Shape Values in Hexadecimal, excluding zero (empty):

Shape Shift Values (amount to shift solid value right):

The highlighted values indicate the corner contributions of solid samples after the shape's shift (in the center) has been applied. These corner values have been selected such that the shape shift look-up-table can be highly compressed. Since shape zero and the final four shapes have a zero shift value they can be omitted from the look-up-table. The shift of shape eleven has only the least significant bit set, so its higher bits can be omitted as well. At three bits per shift value this brings the required look-up-table size down to 31 bits.

```
// shapes =  F   E   D   C   B   A   9   8   7   6   5   4   3   2   1   0
// shifts = 000 000 000 000 001 100 001 001 010 011 110 011 010 011 001 000
lookUpTab = 0x6129e699; // 0110 0001 0010 1001 1110 0110 1001 1001

shape = (cell[ x ][ y ].type == liquid) ? 1 : 0
      | (cell[x+1][ y ].type == liquid) ? 2 : 0
      | (cell[ x ][y+1].type == liquid) ? 4 : 0
      | (cell[x+1][y+1].type == liquid) ? 8 : 0;

shapeShift = shape < 12 ? ( lookUpTab >> ((shape-1) * 3) ) & 7 : 0;

solid = (cell[ x ][ y ].type == solid) ? 0x11 : 0
      | (cell[x+1][ y ].type == solid) ? 0x42 : 0
      | (cell[ x ][y+1].type == solid) ? 0x24 : 0
      | (cell[x+1][y+1].type == solid) ? 0x88 : 0;

row = ( solid >> shapeShift ) & 3;
```

To allow multiple liquids to interact their surfaces should match so that there is no seam when they join. Simply ensure complimentary shapes match, e.g., #1 and #E, #5 and #A, #6 and #9 should match. Perform a Marching Liquid Squares pass for each type liquid being rendered.

Copyright 2014 Timothy Landers

Fig. 11. Landers' "paper" in its concise entirety [8]

2.3 Liquid Simulation

Terrain generation is of course not the only application of compute shaders, but the marching cubes algorithm can also be used for far more than just static objects. A *very brief* (see Figure 11) article by Timothy Landers describes a 2-dimensional version of marching cubes (aptly named *marching squares*), to be used for real-time fluid simulation [8]. Landers expands on the surface boolean value, replacing it with a string of bits. In the article, three bits are used, to represent whether each corner is part of the liquid, the surrounding air, or a solid object with which the liquid is interacting. More bits could be added, increasing the memory required for the algorithm in return for adding extra data which can be used for shading the rendered object differently. This allows for modelling interactions between multiple liquid and solid substances, which could be combined to form much more complex scenes than the shells traditionally produced by marching cubes. For example, different bits could be used to represent different types of terrain, with textures then blended to produce a mountain range with grassy valleys and snowy peaks.

In a 2003 paper by Hong and Kim of Korea University, marching cubes is utilised in a more robust real-time liquid simulation. [6] The majority of this paper focuses on accurately simulating the movements and dynamic interaction between a liquid and bubbles of gas within it, which, while interesting, is not relevant to this project. What is relevant here is the use of the marching cubes algorithm as a visualisation method for the simulated fluids. Crucial here is the concept of *numerical diffusion*, which describes the state in which node values in the cube march grid which lie far from the surface take values other than 0 or 1 (the extremes of the node value scale). Numerical diffusion produces a surface which does not accurately match the intended shape, creating splits in the surface which in this case correspond to bubbles breaking into multiple pieces as in Figure 12.

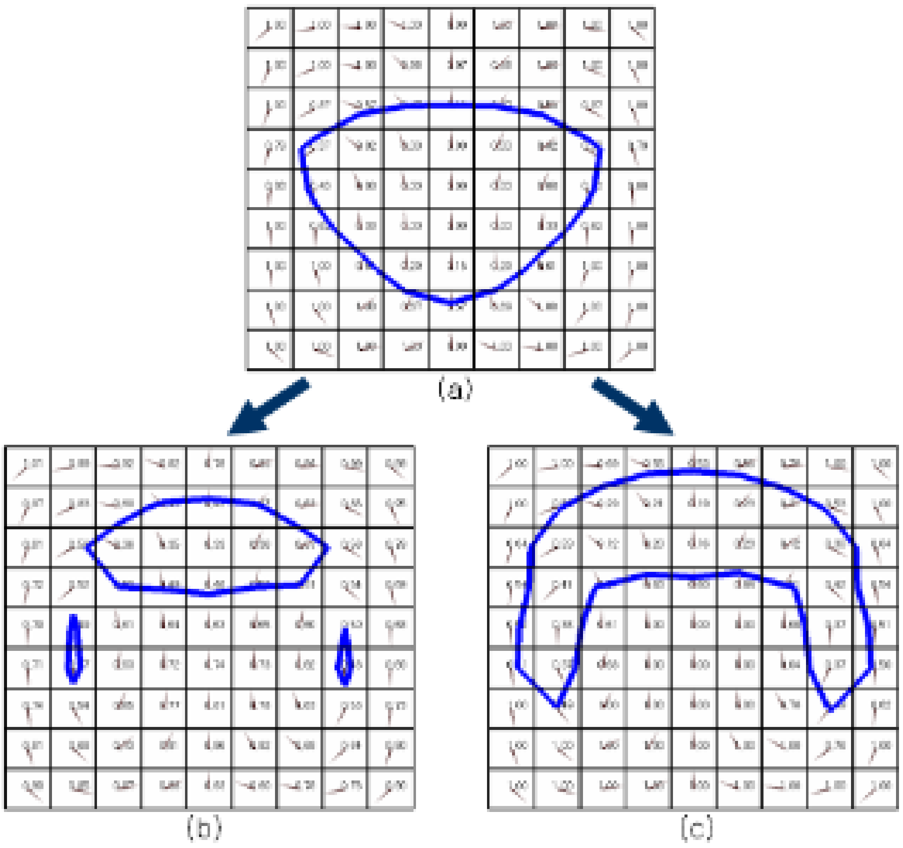


Fig. 12. Errors caused by numerical diffusion [6]

Numerical diffusion causes a major problem for Hong and Kim's fluid simulation, but there is a solution - nodes which lie far from the interface are adjusted to values of 0 or 1, to prevent any ambiguity - however, this results in a loss of mass of the simulated liquid, because as the gas bubbles become less separated they also become larger. The mass of the liquid must be maintained to ensure the accuracy of the simulation, so node values close to the interfaces are scaled by a complex scaling factor based on time and the known mass of the liquid. For terrain generation, numerical diffusion may or may not be a problem - if the result is intended to look realistic, then detached landmasses are of course out of the question, but for a more fantastical or alien world, floating islands may in fact be a benefit to the overall result, as seen in the sky bound temple seen in Figure 13.

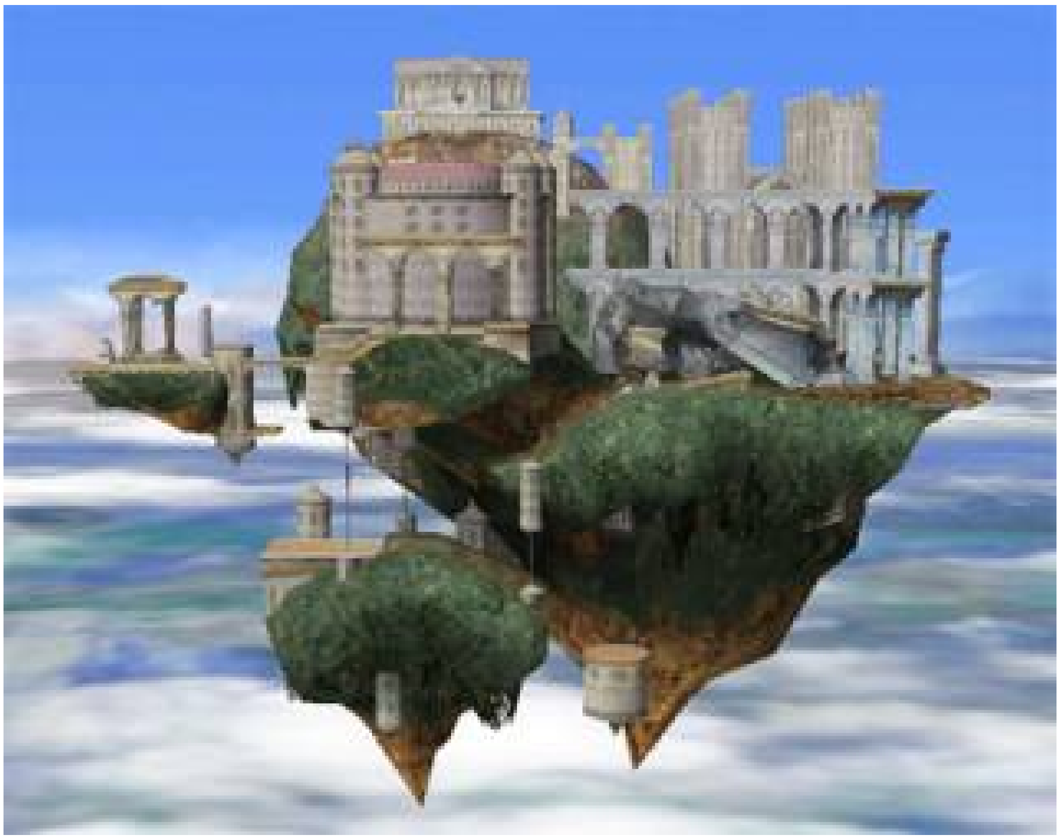


Fig. 13. Floating islands create a striking visual effect in *Super Smash Bros. Melee* (credit: *Nintendo*)

2.4 Generating Useful Noise

The most common form of three-dimensional terrain generators used in modern video games use a heightmap based technique. While this can produce impressive results, the lack of any overhangs or internal features limit the variety of worlds that can be produced in this way. Nevertheless, the techniques used in building up the noise functions behind these heightmaps are entirely applicable to the marching cubes approach to terrain generation - although the noise generated for a heightmap is 2D rather than the 3D noise required for the more complex terrain generated by the marching cubes algorithm, the techniques used to make this noise more complex and realistic are the same.

A commonly used noise function is *Perlin noise*, named for its creator, Ken Perlin. Perlin noise is a *Pixel Stream Editor*, designed to produce realistic approximations of real-world textures, with examples given in the paper for clouds, fire, rock, wood, and crystal, among others. Perlin noise is known to produce natural-looking results due to its continuous nature, while maintaining the ability of each pixel to be computed in parallel, without relying on the results of adjacent pixels. [12]

In a 2015 paper published in the Journal of Compute Graphics Techniques, Ian Parberry explores a surprisingly simple method for generating heightmaps which are incredibly consistent with real-world terrain [11]. Using data gathered from satellite imagery and analysis of a large area of terrain on Earth, Parberry draws the conclusion that most of our planet's terrain follows an *exponential distribution*. This exponential distribution can be achieved with some simple modifications to Perlin noise, which, when multiple octaves are layered on top of each other, produces terrain which could easily pass for a real-world landscape. Hyper-realism is not always the goal in video games, and often a more fantastical world is desirable (hence the desire for overhangs and floating islands afforded by the 3D marching cubes approach) but knowledge of how to imitate reality by methods such as that described by Parberry is a vital starting point for ensuring that virtual worlds are believable, even when they are not realistic.

A way around the two-dimensional nature of heightmap generation is to use multiple layers of noise, one for the heightmap itself, and then multiple extra noise functions to generate internal features of the terrain. These features could be different types of rock such as mineral deposits which would be textured differently and perhaps carry gameplay mechanics, or empty spaces to form cave systems. An investigation into this layered noise approach can be seen in the 2013 International Conference on Cyberworlds. [13] This approach allows the designer fine control over the terrain, as each layer can be tweaked independently of the other layers, providing a high level of customisability. This method is fast enough for real-time generation but is still far slower than a purely heightmap-based generation system, due to the many layers of noise required and the processing cost of combining the layers into a coherent terrain. The internal complexity of this layered approach can be achieved using the marching cubes approach by combining two factors. Firstly, the 3D nature of the algorithm allows for empty caverns and cave systems as an inherent property of the system. The functionality of adding variation of rock types and mineral veins can be achieved by adding extra data to each node in the cube march grid. This extra data would be a flag corresponding to an enumeration of possible rock types, and when the terrain mesh is generated, any points generated on edges connected to a given node will be assigned this same flag, which can then be used to inform texture blending or any number of other operations to be carried out on the finished mesh.

3 Design and Implementation

3.1 Requirements

1: The program must generate a terrain map using the marching cubes algorithm
   a: The size of the terrain map should be variable in 3 dimensions
   b: The terrain should be variable based on a noise function

2: The algorithm must be implemented with a CPU based approach
   a: The CPU based implementation should be single threaded

3: The algorithm must be implemented with a GPU based approach
   a: The GPU based approach should be tested with a range of compute shader work group numbers and sizes

4: The relative performance of both implementations bust be comparable
   a: Performance should be assessed based on the time taken to generate the terrain map
   b: The performance of each implementation should be compared against itself and the other implementation at a range of terrain map sizes

5: The resulting terrain map should be viewable through a 3D renderer
   a: The renderer should be simple and lightweight to avoid interfering with the performance metrics
   b: The time taken to generate the terrain map should be measurable separately from the time taken to render the result to the screen

3.2 Tools and Frameworks

To maintain consistency with (and transferability to) the gaming applications which would utilise such a terrain generation system, the system was developed using C++ and OpenGL. In terms of the efficiency of the compute shader and general graphics performance, a Vulkan or DirectX 12 implementation may have been more performant, but previous experience in OpenGL ensured a quick set up time of the basic rendering environment so that implementation of the relevant parts of the system could start as early as possible. Tools used included Visual Studio 2019 for the C++ project, and Visual Studio Code for shader editing.

The Graphics Library Extension Wrangler (GLEW) [16] and Graphics Library Framework (GLFW) [10] libraries were used to interface with the OpenGL drivers. These libraries allow easy access to OpenGL function calls in a hardware agnostic manner, improving cross platform compatibility and general ease of development, while remaining very lightweight dependencies.

A noise function was needed in order to generate the node weight values from which the terrain mesh would be generated. Believable results would require a continuous noise function, rather than pure static, and Perlin noise was an obvious choice here, as it is known to generate smooth continuous noise suitable for believable (if not actually realistic) 3D terrain. Examples of Perlin noise can be seen in terrain generators in games such as Terraria [18]. In order to generate this noise, the siv::PerlinNoise library was used [17], as it provided a lightweight, single-header solution for generating multiple octaves of Perlin noise with little setup time. In a real game project, a more complex, bespoke noise function would be required, utilising advanced layering techniques and tweaked to create a unique and recognisable look for the game in question. However, for the purpose of demonstrating and evaluating the performance of the generator developed in this project, pure Perlin noise was sufficient.

3.3 Development Process

Before development could begin on any terrain generation system, a testing environment needed to be set up. This consisted of a basic OpenGL renderer with two shaders pipelines - a $vertex - fragment$ shader draw triangles, with colours dependent on screen-space normals (this was simply to help differentiate between adjacent triangles without requiring a full lighting system), and a $geometry - vertex - fragment$ shader to take in points and draw quads with a grayscale colour. This second shader pipeline was used to visualise the cubemarch grid, with nodes coloured according to their weight. The purpose of the renderer is only to test the output of the terrain generator, it is not intended to represent the final graphical result which may be presented in a finished game.

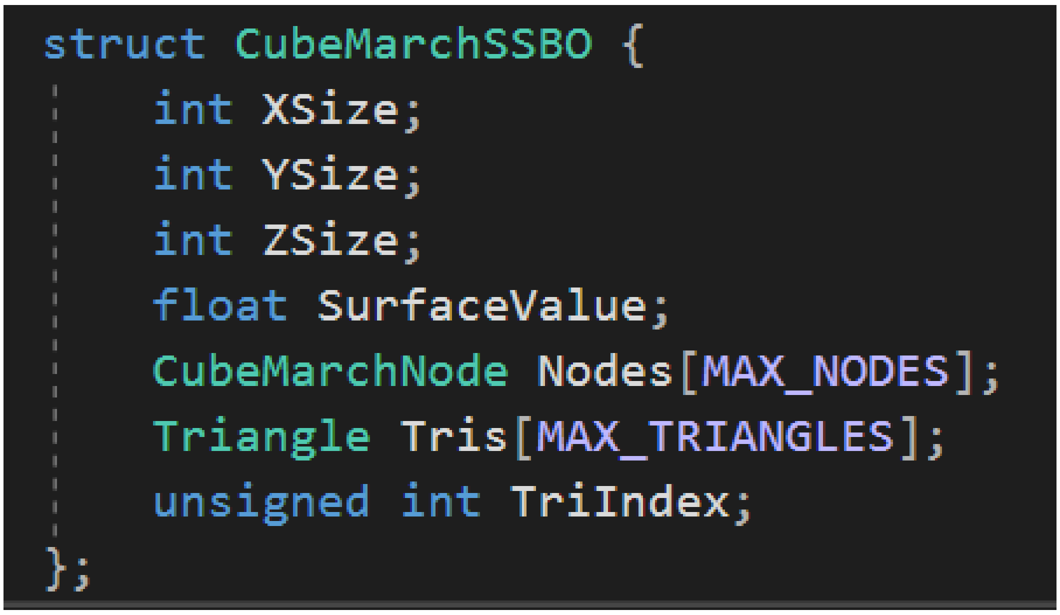
The setup for both implementations is the same, and is handled by the CPU in both cases, as it is not part of the performance analysis and was more straightforward to develop without the use of compute shaders. A GPU based approach to generating the initial point cloud could produce significant performance improvements to the setup process, but the time taken to initialise the node grid was not long enough to pose a problem in this project, and such optimisations were secondary to the purpose of the experiment, as, even in a real-time generation application, the node grid values could be initialised ahead of time. A point grid of arbitrary dimensions is generated (tested up to $100 \times 100 \times 100$), with each node assigned a weight between 0.0 and 1.0 output by the Perlin noise function. The sample space for the noise function can be tweaked by offsetting the entire grid, and by changing the spacing between each node. This point grid can then be rendered with each point represented by a quad with a grayscale value representing its weight - 1.0 will appear white and 0.0 black. This aids in visually understanding the marching cubes process, which is especially useful in debugging the result of the terrain generator during development.

Once the point grid has been generated, it is time to execute the marching cubes algorithm. This is carried out by a GenerateMesh function, which, based on a boolean toggle, will call GenerateMeshCPU or GenerateMeshGPU. This toggle functionality allows the program to switch between the two implementations at runtime to quickly compare the performance of each on the same terrain.

The process of generating a mesh from the point grid is close to the pure marching cubes implementation described in the original paper [9] - points which make up a cube are passed, six at a time, to a SolveCube function, which compares the weight values of these points to a given *surfacelevel* value to produce the surface boolean value for each point. These values are bitmasked together to generate an index which is used to look up the relevant cube state from the 256 possible permutations. The cube could contain up to 5 triangles (15 points) but will exit early if there are no more points in the array (indicated by corner indexes of -1), mitigating the processing time wasted on cubes which lie entirely inside or outside the mesh (i.e., containing none of the surface within the cube). This cube state is an array of integers from 0 to 7, representing the 8 corners of the cube. Adjacent numbers represent the edge between those two corners, and indicate that a point of the generated surface lies on that edge - for example, $\{1, 2, 3, 4, 5, 6\}$ indicates that a triangle should be drawn with points lying on the edges between corners 1 and 2, 3 and 4, and 5 and 6. Linear interpolation is then used, comparing the difference between the weight of the node at each end of the edge to the surface value in order to determine how far along the edge the point should be placed. Once every point within the cube has been calculated, SolveCube returns that array of points, which is appended to the master array until all cubes have been solved and the mesh is complete.

The GPU implementation follows a very similar structure to the CPU version, with development being almost as simple as porting the C++ code to GLSL. The SolveCube function from the CPU implementation is almost unchanged here, with the key difference being that each invocation of the compute shader operates on only a subset of the cubes in the grid, rather than the full range. Ideally, to achieve maximum parallelisation, each invocation will operate on a single cube, but the possibility of this criteria will depend on the number of workgroups and invocations available on the GPU in question, and for generating very large maps, it will be necessary to solve multiple cubes per invocation. The grid is divided between invocations proportionally to the "grid" of invocations compared to the grid itself, i.e. if a grid of $256 \times 256 \times 256$ cubes is to be solved using $128 \times 128 \times 128$ invocations, each invocation will solve a sub-grid of $2 \times 2 \times 2$ cubes. As the number of cubes solved by each invocation increases, the benefits of the GPU based approach over the single-threaded CPU implementation decrease, and so in a real time application it would be more beneficial to split the terrain up into chunks as in the GPU Gems 3 implementation.

Some extra considerations need to be made when parallelising the data in this way, mainly in terms of memory management. The point grid, as well as the points output by the compute shader, is stored in an SSBO, the structure of which can be seen in Figure 14. An SSBO was used rather than a UBO because, although a UBO is faster to access, it simply does not offer the capacity required for the large number of triangles being generated by the compute shader. A UBO has a maximum size in the order of kilobytes [5], whereas an SSBO allows access to the entire GPU memory space [4] (in this case, 8GB). As Sebastian Lague points out in a short foray into the marching cubes algorithm [7], multiple compute shader invocations writing to this buffer at the same time will cause crosstalk. Points from different triangles get erroneously interleaved and produce the kind of "triangle soup" seen in Figure 15 where all of the correct points are present, but they're connected up in the wrong order. The solution here lies in *atomic counters*. These counters are persistent across invocations of the compute shader, and each operation on the counter is, as the name suggests, *atomic*. This means that whenever an invocation tries to increment the counter, it must first wait until any other invocation has finished its own operation on the counter, and then the counter state is locked to this iteration until its operation has completed. This creates a blocking scenario, whereby each operation on the atomic counter will lock all other invocations which may also be trying to access the counter. This unfortunately slows performance of the compute shader as a whole but is necessary for preventing crosstalk. The number of operations on the counter is somewhat mitigated by adding points to the mesh a triangle at a time, rather than point-by-point, effectively reducing the number of locking conditions by a factor of 3.



```
struct CubeMarchSSBO {
    int XSize;
    int YSize;
    int ZSize;
    float SurfaceValue;
    CubeMarchNode Nodes[MAX_NODES];
    Triangle Tris[MAX_TRIANGLES];
    unsigned int TriIndex;
};
```

Fig. 14. The SSBO data as it is laid out in memory. $MAX\_TRIANGLES$ and MAX_NODES are changed depending on the size of the grid being generated but are constant in the compute shader.

With the crosstalk situation fixed, the compute shader can produce the same terrain mesh as does the CPU based implementation, when given the same point grid to operate on, by solving every cube in the grid at once rather than one at a time.

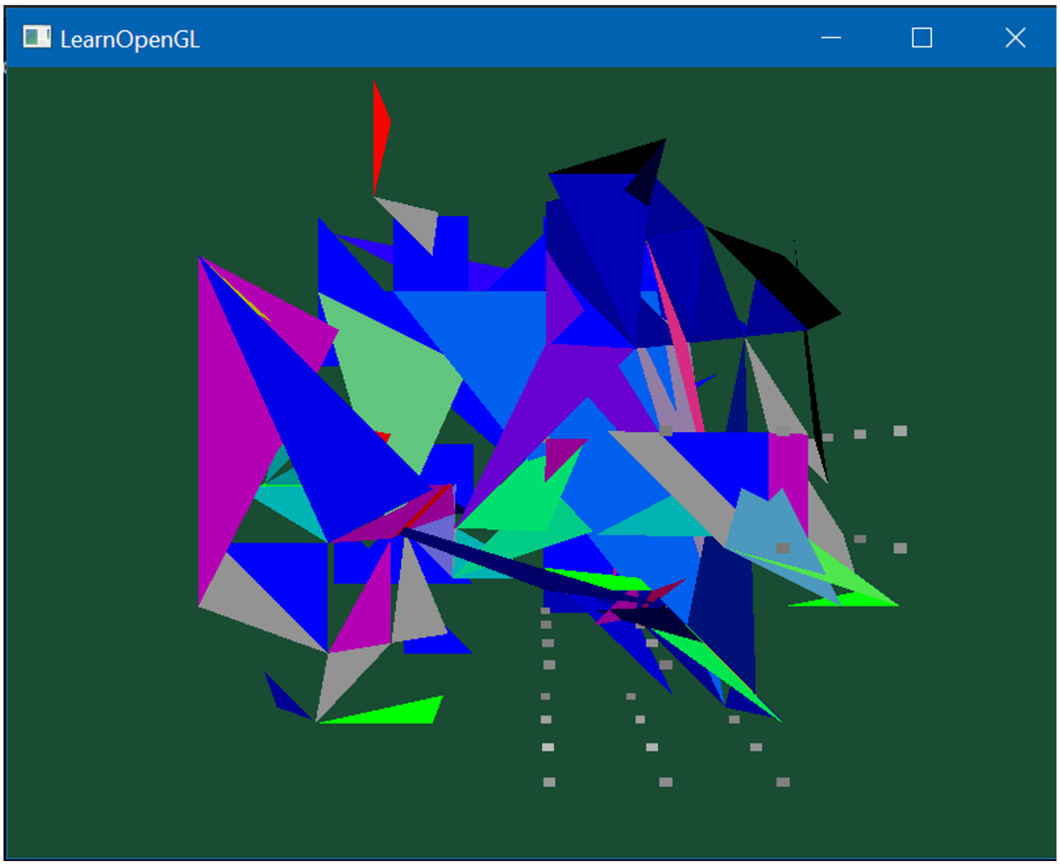


Fig. 15. Something is not quite right with this terrain...

4 Evaluation

4.1 Testing Criteria

This project was tested using the intel i5 6600k, and AMD Radeon RX 480 GPU, with 16GB of system RAM and 8GB of VRAM. The relative performance of these specific components undoubtedly plays a part in the performance results, but the comparisons between the two implementations can be extrapolate to a wider range of systems.

Testing of the generation system would be based only on time taken to complete the execution of the marching cubes algorithm. Usage of both system and GPU RAM was not assessed or compared between the two generation methods. In order to measure the time taken for each generation method, the std::chrono library was used. By simply wrapping each call to GenerateMesh with calls to glfwGetTime, the difference between the system time immediately before and after the terrain generation operation can be recorded and compared to get a rough estimate of the time taken to generate the mesh. This time is not incredibly precise, nor can it be extrapolated to different hardware configurations, but running the generation multiple times and taking an average gives enough data to compare the performance of the two implementations against each other and determine how much more or less performant one is than the other.

4.2 Testing Method and Results

Before comparing the CPU vs GPU based implementations, the GPU implementation was dialled in using different work group sizes to find the optimum balance. This test involved generating the same terrain from a $100 \times 100 \times 100$ grid, with a single cube per invocation, but varying the arrangement of number of work groups vs work group size. The test was first run with a work group size of 1 (with $100 \times 100 \times 100$ work groups), and then the work group size was scaled up as the number of work groups was scaled down. The generation was run 10 times at each work group size configuration, and the average generation time for each configuration output to a file (see Figure 16). This test revealed that a larger work group size produced faster performance, presumably due to the way that the compute shader dispatches invocations within a work group compared to multiple separate workgroups. Following the result of this test, the largest possible work group size was used when comparing the GPU based implementation to the CPU based implementation. This maximum size was $50 \times 50 \times 50$, beyond which OpenGL refused to dispatch such a high number of invocations. The difference in time to generate the terrain was relatively small (1.60 vs 1.96 seconds) but an improvement of roughly 22% is considerable.

Both implementations were tested with the same seed for the Perlin noise generator, i.e. both implementations were generating the exact same mesh in these tests, to ensure that variations in the noise function were not affecting the results. Each implementation was run to generate a mesh from a grid of sizes from $5 \times 5 \times 5$ up to $100 \times 100 \times 100$. Each size was generated 10 times, and the average time for each implementation recorded. The results can be seen in Figure 17.

On small grids (up to $60x60x60$), the CPU based approach actually performs better. This was surprising, as the parallelisation achieved by the GPU was expected to overpower the CPU based implementation outright. However it does make sense - moving data to and from the GPU from system RAM is an expensive operation, and no matter how fast the GPU computes the marching cubes algorithm, generating the mesh will take at least as long as the time required to copy the required data to and from the GPU memory. Once the grid size increases past $60 \times 60 \times 60$, the GPU far outperforms the CPU - the CPU times quickly increase almost exponentially, while the GPU times increase far slower, and at the top end of this test the GPU generated the terrain more than twice as quickly as the CPU. The GPU is still on a worse than linear trend, though, and times upwards of 1.5 seconds are still not suitable for a real-time application, not to mention the fact that in-game generation time would be far higher due to the GPU being utilised for rendering tasks at the same time.

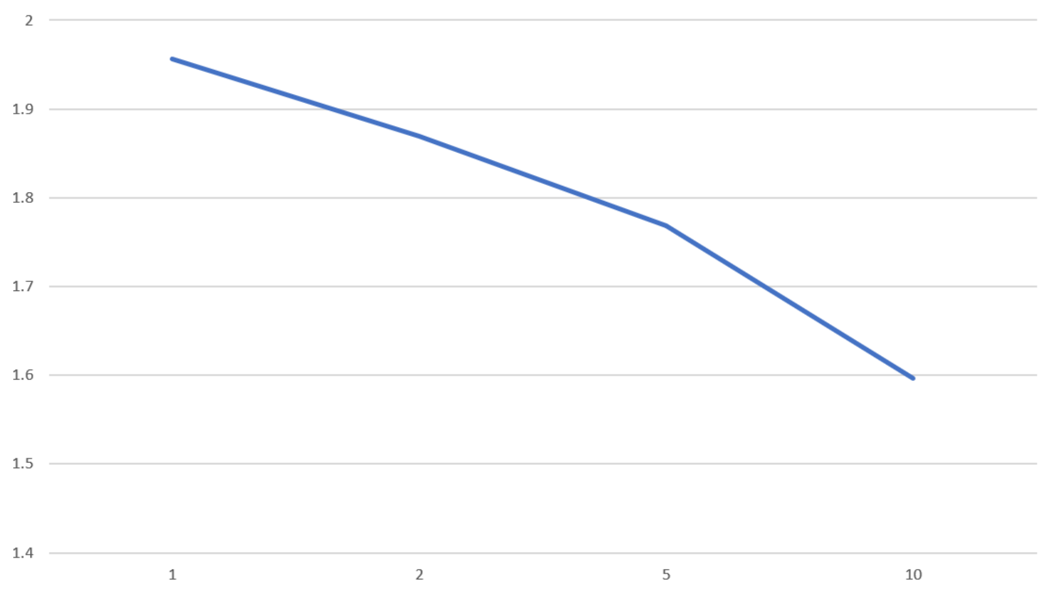


Fig. 16. Time to Generate (seconds) vs Work Group Size

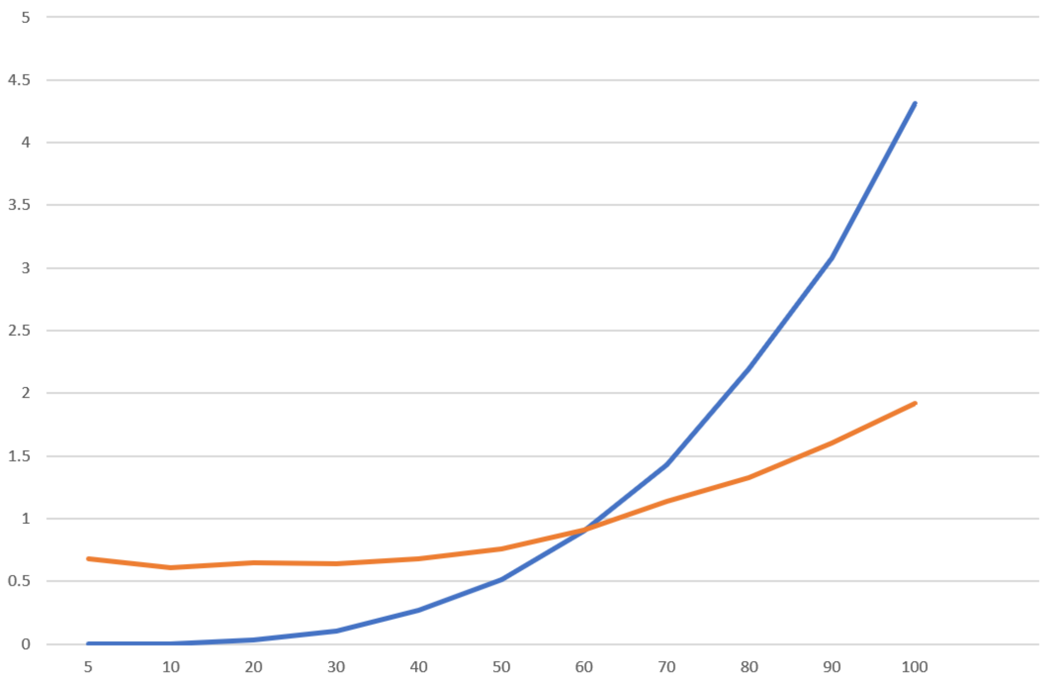


Fig. 17. Time to Generate (seconds) vs Grid Size (side length of cube) - CPU in blue, GPU in orange.

5 Conclusions and Further Work

5.1 Project Outcomes

As expected, the GPU was able to generate terrain far faster than the CPU on larger grids but was surprisingly slower for very small grids. Neither implementation was sufficiently fast for generating terrain in real-time during gameplay at large grid sizes, meaning procedural generation of high-fidelity landscapes would still need to be performed during a loading screen in between gameplay sections, rather than as the player explores the game world. If a far lower-fidelity appearance was acceptable, for example in a game using a low-poly art style, terrain could still be generated in real time using a far smaller grid size per tile using the CPU. Generating a tile of terrain at a $10 \times 10 \times 10$ grid size, for example, only took 0.003 seconds, or 3 milliseconds on the CPU, which could feasibly fit into the 16ms target for 60fps gameplay without causing any stutter. The unexpectedly poor small-scale performance of the GPU rules it out entirely for real-time terrain generation, as it would take 600 milliseconds - 200 times as long - to generate that same $10 \times 10 \times 10$ tile. If terrain is to be generated during a loading screen, however, the GPU based approach could drastically reduce those load times compared to the CPU based approach, as it performs well over twice as fast when scaling the grid size up past $100 \times 100 \times 100$ per tile.

5.2 Improvements and Further Work

Some optimisations to the Marching Cubes algorithm have been mentioned in this paper which were not applied to the project itself. If this system were to be integrated into a full game project, then some of these optimisations would be considered.

Perhaps the simplest optimisation is in the vertices output by the GenerateMesh function - the mesh is made up of individual triangles, meaning that where multiple triangles meet, vertices are duplicated. The rendering time of the mesh could be improved by checking each time a vertex is added to the mesh if a duplicate vertex already exists in the mesh at that position. Duplicate vertices can then be skipped, and an index added instead to correspond to the existing vertex in the mesh. This would speed up rendering of the terrain but would not significantly improve generation time of the system, and so was not necessary to implement in this experiment.

A potential optimisation which would significantly improve the generation time is the re-use of solutions to adjacent cubes outlined by Lorensen and Cline back in 1987 [9]. As shown back in Figure 9, for any given cube (except for those at the corners and edges of the grid), nine of the twelve edges have already been solved in a previous cube. Implementing a system for re-using these previously solved edges should improve single-threaded performance of the algorithm by a factor of three. However, due to its reliance on the cubes being solved in a specific and predictable order, the parallelised nature of the GPU based implementation makes this "trick" impossible to apply. Improving the CPU performance by such a high factor would improve the system's prospects as a real-time system for low-fidelity game worlds, but the GPU would still vastly outperform the CPU with larger scale maps.

References

[1] M. Garda. Neo-rogue and the essence of roguelikeness. *Homo Ludens*, 1(5):59–72, 2013.

[2] R. Geiss. *GPU Gems 3*. NVidia Corporation, 2007.

[3] Khronos Group. Opengl wiki: Compute shader. Last Accessed: 14/05/2020.

[4] Khronos Group. Opengl wiki: Shader storage buffer object. Last Accessed: 16/05/2020.

[5] Khronos Group. Opengl wiki: Uniform buffer object. Last Accessed: 16/05/2020.

[6] J. Hong and C. Kim. Animation of bubbles in liquid. *EUROGRAPHICS*, 22:253–262, 2003.

[7] S. Lague. Coding adventure: Marching cubes, 2019. Last Accessed: 15/05/2020.

[8] T. Landers. Marching liquid squares, 2014. Last Accessed: 20/05/2020.

[9] W. Lorensen and H. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH*, 21, 1987.

[10] C. Lowy and various contributors. Glfw. Last Accessed: 10/05/2020.

[11] I. Parberry. Modeling real-world terrain with exponentially distributed noise. *Journal of Computer Graphics Techniques (JCGT)*, 4(2):1–9, 2015.

[12] K. Perlin. An image synthesizer. *SIGGRAPH Comput. Graph.*, 19(3):287–296, July 1985.

[13] A. et al. Santamaría-Ibirika. Volumetric virtual worlds with layered terrain generation. *International Conference on Cyberworlds*, pages 20–27, 2013.

[14] Planetside Software. Terragen. Last Accessed: 13/05/2020.

[15] System Era Softworks. Astroneer. Last Accessed: 04/05/2020.

[16] N. Stewart and various contributors. Graphics library extension wrangler. Last Accessed: 10/05/2020.

[17] R. Suzuki. siv::perlinnoise. Last Accessed: 10/05/2020.

[18] J. Tippets. 3d cube world level generation. Last Accessed: 12/05/2020.