Procedural Generation of Features for Volumetric Terrains using a Rule-Based Approach

Rahul Dey

For Ori.

Abstract

Terrain generation is a fundamental requirement of many computer graphics simulations, including computer games, flight simulators and environments in feature films. Volumetric representations of 3D terrains can create rich features that are either impossible or very difficult to construct in other forms of terrain generation techniques, such as overhangs, arches and caves. While a considerable amount of literature has focused on procedural generation of terrains using heightmap-based implementations, there is little research found on procedural terrains utilising a voxel-based approach.

This thesis contributes two methods to procedurally generate features for terrains that utilise a volumetric representation. The first method is a novel grammar-based approach to generate overhangs and caves from a set of rules. This *voxel grammar* provides a flexible and intuitive method of manipulating voxels from a set of symbol/transform pairs that can provide a variety of different feature shapes and sizes.

The second method implements three parametric functions for overhangs, caves and arches. This generates a set of voxels procedurally based on the parameters of a function selected by the user. A small set of parameters for each generator function yields a widely varied set of features and provides the user with a high degree of expressivity. In order to analyse the expressivity, this thesis' third contribution is an original method of quantitatively valuing a result of a generator function.

This research is a collaboration with Sony Interactive Entertainment and their proprietary game engine PhyreEngine^{TM}. The methods presented have been integrated into the engine's terrain system. Thus, there is a focus on real-time performance so as to be feasible for game developers to use while adhering to strict sub-second frame times of modern computer games.

Acknowledgements

I would like to thank the Centre for Digital Entertainment, EPSRC and Sony Interactive Entertainment for giving me the opportunity to work on some challenging problems. Christos Gatzidis provided invaluable guidance, encouragement and support throughout the entire process. Thanks to Jason G. Doig for volunteering his time to be my industrial supervisor.

The team at Sony, especially Richard Forster and David Lovegrove, were incredibly supportive and allowed me to bounce ideas off of them. They were always happy to help and be my rubber duck whenever I came across a particularly challenging issue.

The research engineers at the CDE have given me some fantastic memories and friendships, and the organisation and support from the CDE project coordinators has been wonderful - special mention to Zoe Leonard and Mike Board.

I would finally like to thank my family - my parents have made countless sacrifices and helped me to get to where I am today. My wonderful wife, Jessica Dey, has shared in all of the ups and downs of the process and has been a rock to me throughout. This would not have been possible without them.

Publications

***Content Generation for Serious Games*, Rahul Dey and Johannes Konert. Book chapter in *Entertainment Computing and Serious Games* (pages 174-188), Springer, 2016.**
Part of this publication is found in Chapter 2.

***Procedural Feature Generation for Volumetric Terrains using Voxel Grammars*, Rahul Dey, Jason G. Doig, Christos Gatzidis. Journal paper in *Entertainment Computing* (volume 27, pages 128-136), Elsevier, 2018.**
This paper is presented in Chapter 3.

***Procedural Feature Generation for Volumetric Terrains*, Rahul Dey, Jason G. Doig and Christos Gatzidis. Poster in *SIGGRAPH*, ACM, 2017.**
An extended version of this poster is found in Chapter 4.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many forms of virtual media, including video games, movies and simulations, need to model large-scale, outdoor environments. Terrains are heavily utilised as a fundamental basis for the construction of realistic virtual worlds for these outdoor environments. Modern GPU pipelines are optimised for processing triangles and, as such, a terrain mesh with triangular polygons is used to render the terrain for the majority of applications. Thanks to current hardware and its computational power, present-day virtual worlds are usually vast, open and detailed. This offers a multitude of experiences to the user as they explore and roam the simulated world.

As virtual worlds get larger, developers utilise a number of designers and artists to fabricate and populate the environment. Undertaking this task in an entirely manual manner is a laborious task and algorithmic assistance can greatly reduce the time taken to construct environments. Procedural content generation describes various methods to computationally create assets, such as meshes, textures and sounds, and is a field of study with a vast array of research. Procedural techniques for terrain generation often consist of generating 2D textures that represent elevation data, termed *heightmaps*. These textures are usually generated using various noise functions depending on the required aesthetic. The elevation data encoded in the heightmap is then used to displace vertices on a 3D grid to form a mesh that represents a terrain. However, as only surface level elevation data is represented, heightmaps are limited in that they cannot create features that consist of concave or subsurface elements found in real world terrains. For a generator to be able to portray concave features the data must be represented in a different way to heightmap-based approaches.

This thesis uses a volumetric data format to model features that are found in natural terrains. Volumetric methods model data using volumetric elements known as *voxels*, which represent a small and discrete point of material in 3D space. This thesis introduces two methods to procedurally act on volumetric data in order to construct terrain features for virtual simulations. The first method adopts a grammar-based approach, where an initial terrain is generated using current techniques and then modified using a set of rules to perform transformations on subsets of the terrain data. The second method introduces three procedural generators to specifically create features found in natural terrains that are difficult to produce using heightmap-based approaches, such as overhangs, arches and caves. The grammar-based method is profiled using CPU timers to determine how fast it performs. The generators are profiled using GPU timers to determine the time taken for their respective algorithms. Furthermore, the generators are evaluated on their expressivity, which governs the variety of features that can be generated using these functions.

Avenues of future work are discussed at the end of the thesis. This includes the use of a more abstract grammar-based solution, further parallelisation of grammars, and machine learning methods that may improve the appearance and aesthetics of generated terrains.

1.1 Company Background

Sony Interactive Entertainment (SIE) is responsible for the development of all PlayStation consoles and technologies. SIE contains a large research and development branch, SIE R&D, that handles all development and interaction with game developers for all PlayStation products, including console SDKs, toolchains, support and training. Furthermore, SIE Euro R&D consists of a number of teams that are responsible for various aspects of the game developer experience including, but not limited to, networking, advanced processing (utilising GPGPU technologies), audio, and game engines. A proprietary game engine used at SIE is PhyreEngine™. PhyreEngine™ has been used in a number of games, including *Hotline Miami*, *Journey* and *OlliOlli*, and is designed as an extensible, cross-platform and highly optimised game engine primarily for use with the various PlayStation platforms. It has also been ported to support Android and iOS devices and is distributed with the full source code to the engine and its associated tools. Game construction can

take place either by solely using the underlying engine source code or by utilising the level editor that covers all aspects of game creation, including level design, asset importing and scripting.

This thesis is associated with PhyreEngine™. At time of writing, the engine was limited in how terrains were created and represented, only being able to utilise a straightforward, heightmap-based approach with constraints on how terrain features could be edited in a game world. It was requested that procedural generation methods be researched and developed to complement the existing functionality, so as to expand the portfolio of tools for terrain creation that was available to game developers.

1.2 Aims and Objectives

Based on the limitations of heightmap-based terrains, this thesis aims to gain insights into the following research questions:

- What procedural generators are appropriate for creating features in volumetric terrains?
- How can these procedural generators be constructed to provide a high degree of user direction whilst ensuring enough variation of results?
- How can the expressivity of a procedural generator of a terrain feature be represented quantitatively?

Our proposed solutions to these questions aim to create practical methods to generate volumetric features found in real world terrains. The overall objective of this research is to integrate these methods into PhyreEngine™ in order to satisfy the research requests from the developers as well as address the current limitations of the engine. As such, the objectives of this research included adding the following features to the engine:

- Design and implementation of volumetric terrain representations.
- Procedural generators for terrain features such as overhangs, arches and caves.
- Tooling for content developers to construct volumetric terrain features.

In turn, the implemented features should exhibit the following desirable traits:

- Performant - The proposed solutions should run at an interactive rate of performance. Content creation should consist of fast feedback so as not to hinder the productivity of the environmental designer.
- User guided - While the solutions should provide a good deal of automation to ease the creation of terrain features, there should be sufficient capability to generate features tailored to the user's specific aesthetic.
- Platform agnostic - Given the cross platform nature of PhyreEngine™ and the flexibility provided by the engine, the presented solutions should be agnostic enough to integrate into a variety of volumetric rendering architectures.

1.3 Contributions

Research, development and engine integration of the aforementioned features corresponds to the contributions of this thesis. These are as follows:

- A platform-agnostic, grammar-based method to generate overhangs and caves for volumetric terrains, including methods and guidance to design rulesets for producing user-directed aesthetics.
- Three parametric procedural generation functions to generate overhangs, arches and caves, respectively.
- A novel method to quantitatively analyse expressivity of procedural generators for volumetric terrain features.

1.4 Thesis Outline

This thesis is separated into a number of chapters. Chapter 2 provides an outline of the current state of the art for voxel terrains, procedural generation and volumetric rendering. Each of these are relevant subdomains related to the problem at hand. Firstly, methods of storing and managing voxel terrains using data structures and algorithms are surveyed. Then, different ways of procedurally generating content are described, and, finally, methods of rendering volumetric elements effectively for use in virtual simulations are

examined. The chapter concludes identifying a niche in the current literature. Chapter 3 introduces a novel procedural generation method inspired by grammar-based solutions that we term *voxel grammars*. This method allows a user to design a set of shape rules in order to generate terrains by defining pairs of input voxel grids (*predicates*) and output voxel grids (*transforms*). Chapter 4 introduces three high-level procedural generation functions to parametrically construct features uniquely found in volumetric terrains. Functions for cliffs, arches and caves are described and implemented on the GPU in order to quickly generate the require voxel data. Finally, Chapter 5 summarises the research, presenting the contributions. It also discusses potential avenues for future research, including extensions to the voxel grammar concept by defining a stochastic, parametric grammar that can be designed to be a higher-level and more intuitive method for designing grammars.

Chapter 2

Literature Review

2.1 Introduction

This chapter describes the current research of procedural generation methods, volumetric terrain and feature creation for terrains. There are many procedural methods used to create terrains and this chapter focuses on methods suited for constructing volumetric terrains. The ways that terrain features are formed geomorphologically is also discussed in order to form a basis for how these processes can be transferred to virtual simulations.

Part of this chapter was presented in Dey & Konert (2016) as part of a survey on current methods of procedural content generation for the domain of serious games.

2.2 Background

This section presents a broad overview of the topics relevant to real world terrains, procedural content generation and volumetric simulation in order to provide the reader with context for the rest of the thesis. It briefly describes the relevant concepts of geomorphology and provides background information for the generation, processing and rendering of voxels.

2.2.1 Geomorphology

The field of geomorphology studies the formation of various types of landforms. This includes the creation of cliffs, natural arches and caves.

Cliffs are steep slopes that are formed by weathering and erosion. They tend to form commonly around coastal regions and further wave erosion can form notches into the cliff face (Huggett 2016).

The formation of both natural arches and caves generally follows a similar process. Both tend to occur in karst terrain using erosion. A terrain can be defined as karst if water dissolves soluble rocks in order to alter the topology of the terrain above or below the water source (Jennings 1971). Natural arches can form via limestone dissolution where a river cuts through narrow bands of limestone. Furthermore, they can also occur when parts of a cave's interiors collapses and forming by the remaining cave ceiling.

Caves form via similar limestone erosion in karst terrain. In order to begin forming, caves require a cavity filled with water that can dissolve rocks so that the fluid can be channelled. Cave formations can be split into two types: vadose and phreatic. The primary difference between vadose and phreatic caves is the location of the water table relative to the cave entrance. Vadose caves lie above the water table, whereas phreatic caves are below the water table and are therefore filled with water (Bretz 1942).

2.2.2 Voxelisation

Modeling and editing the internal structure of geometry requires a volumetric data representation. This can be obtained from polygonal geometry by the process of *voxelisation* (sometimes referred to as *3D scan conversion*). There are a variety of methods to perform this conversion. Furthermore, as GPGPU paradigms have come to the forefront of data processing in recent years, massively parallel designs have heavily optimised voxelisation algorithms by orders of magnitudes.

Different algorithms either perform a *surface* or a *solid* voxelisation. A surface voxelisation generates voxels representing the external surface of the supplied geometry. A solid voxelisation represents the surface as well as all interior voxels. Solid voxelisations can be very useful for intuitive constructive solid geometry.

Chen & Fang (1998) introduces an algorithm for surface voxelisation by using the hardware rasterizer to render view restricted slices of fragments, seen in Figure 2.1. Geometry is rendered multiple times using an orthogonal projection matrix with increasing clipping planes. Fragments that pass the depth test are rendered into the framebuffer, which is in turn copied into a 3D texture. After all slices have been rendered, the texture can be sampled

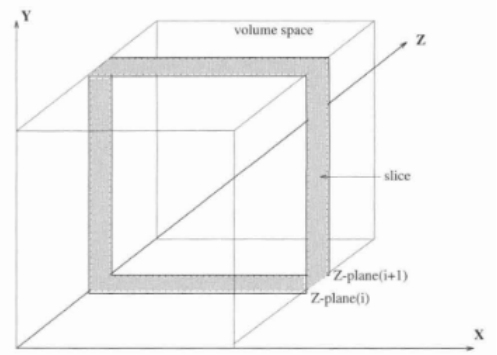

Figure 2.1: Illustration of volume slicing used in Chen & Fang (1998).

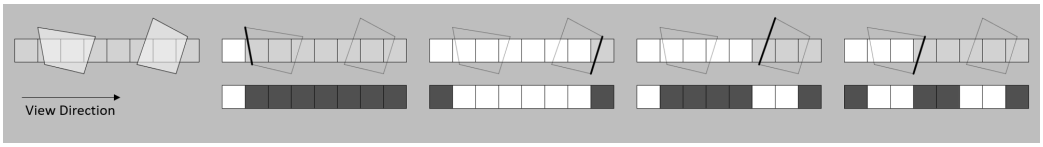

Figure 2.2: Illustration of using XOR to perform solid voxelisation. For each face of the initial state with two shapes (left), voxels are counted to the face and a bitwise XOR operation is used to compute the current bitstream (middle-left, middle, middle-right), until the final voxelisation is computed (right).

as a dense grid of voxels. Li et al. (2005) reduced the total number of render passes of this method by utilising the concept of depth peeling (Everitt 2001).

Parity checks can be used with such slice based methods to determine the existence of internal voxels and result in a solid voxelisation of an object (Hales 2007). A voxel can be determined to be internal or external to an object's surface by accumulating the number of intersections from a ray cast at each voxels position. This can be further optimized to apply an XOR operation instead, such as in Fang & Chen (2000). An illustration of this XOR method can be seen in Figure 2.2. Liao (2008) also uses a similar method by utilising the GPU's stencil buffer in a manner akin to the classic shadow volume algorithm (Crow 1977). However, parity check methods only work effectively with completely manifold, watertight meshes. A scene's meshes should also not be overly intersecting as this can generate voxels that

are false positives.

Karabassi et al. (1999) presented a simple surface voxelisation method that uses 3 depth buffer pairs (one pair for each dominant axis) and renders an object into each one. Whilst this method is fast and straightforward, the main limitation is that it can only effectively voxelise convex meshes.

Eisemann & Décoret (2006) demonstrated a fast binary, surface voxelisation method that can deal with dynamic objects in a scene. Eisemann & Décoret (2008) later extend on this work to provide a single pass, solid voxelisation method. However, their reliance on depth buffers means that the resulting voxelisation can suffer from false negatives due to depth discontinuities.

Forest et al. (2009) create a hierarchy of mipmaps by calculating world space positions of primitives, computing their appropriate voxels and storing the information in a 2D texture. This texture is mipmapped to create a dense octree-like hierarchical structure, which provides the advantage of accelerating raycast operations using less detailed mipmaps.

Thiedemann et al. (2011) removes the dependency on depth buffers in their work and can effectively handle multi-valued voxelisations. This method is limited by its need to fabricate a texture atlas beforehand, and this can become a laborious and prohibitive process as models increase in complexity and polygon count.

Schwarz & Seidel (2010) introduce fast, data-parallel algorithms for both surface and solid voxelisations. They also apply the concept of conservative rasterization to voxels in order to remove the limitation of other voxelisation methods that require watertight meshes. While this is executed in a compute shader on modern GPUs, the introduction of hardware accelerated conservative rasterization may assist in increasing the efficiency of this technique (Akenine-Möller & Aila 2005).

2.2.3 Volumetric Data Structures

Voxels can be used to represent a wide variety of attributes within a virtual scene and have a number of applications in many domains, including medicine and the film industry. Examples include approximating real-time global illumination (Crassin et al. 2009), construction of deep shadow maps (Wyman 2011) and scattering simulations within participating media (Cerezo et al. 2005). They have also been used to create volumetric terrains due to their ability to represent features such as overhangs, caves and arches (Peytavie

et al. 2009).

The main advantage of using voxels is their inherent ability to store information about the internal structure of the geometry. This is particularly useful when the object being modelled consists of non-homogeneous data. For example, a cloud can be modelled as a voxel grid of density values denoting the number of light-interacting particles at uniform sample points of the volume as per Tatarchuk (2015). Surface detail in this instance would not be enough. Similarly, a terrain can consist of many different materials (e.g. topsoil, sand, rock, etc.), which is pertinent information for this research.

However, voxel buffers do possess some inherent disadvantages. The primary drawback is that they are bound by memory limitations as their resolution increases and can be infeasible to store in main memory, particularly as real-time applications require memory for other purposes. Streaming this data from a hard disk is usually too slow for immediate usage, although there are some out-of-core algorithms that handle creation and modification of voxel buffers such as in the work of Baert et al. (2014).

The most naïve form of storing voxel data is to use a dense 3D array of values. While this provides very fast value lookups (O(1)), 3D arrays begin to consume prohibitive amounts of memory at larger resolutions. For example, a reasonably sized 2048^3 dense voxel grid containing a single floating point value per voxel (assuming a 32-bit float on most modern compilers) would consume 32GB of memory. Furthermore, as processing voxel data is a highly parallelisable task, it is important that entire datasets are either fully resident in GPU memory or the amount of streaming where sections of the data are copied from the main memory to the graphics card memory is minimised. This is a slow operation as it introduces CPU-GPU synchronisation points that can stall the GPU and slow down processing significantly.

A straightforward improvement to the dense grid is to introduce the idea of sparseness. A sparse structure only stores values for occupied voxels, whereas a dense structure (such as a 3D array) stores values for every voxel even if empty.

A simple sparse data structure for a voxel buffer is found in (Wrenninge 2012), where groups of voxels are separated into blocks. For example, a grid with resolution 64^3 can be separated into 4096 blocks with resolution 4^3. The process of writing a voxel to this grid works as follows:

1. Compute the block ID for the requested voxel index.

2. Check if the block that will hold the voxel has been allocated memory.

3. If it is not allocated, then allocate memory for the entire block.

4. Write the voxel value in the block.

This can save a substantial amount of memory, particularly when there are large amounts of contiguously empty voxels in the data being voxelised. However, due to the allocation of the entire block, some tweaking is necessary to maintain a balance between performance and memory consumption. Furthermore, if voxels of the dataset are sparsely distributed (e.g. only 1 voxel per block is written to) then there is a large amount of wasted memory allocated to consistently empty voxels.

Recently, there have been advances in data structures used to store volumetric data and these have significantly improved the memory footprint of high resolution datasets (Crassin et al. 2009) (Museth 2013). For example, the sparse property has been extended to include hierarchical data structures such as octrees. Typically, an octree node stores pointers to its child nodes, however, a common thread in many fast octree implementations is the use of pointer-less data structures (Lewiner et al. 2010). Instead of pointers, the children are stored contiguously along with the parent to exploit cache coherent behaviour when traversing the tree. A further optimisation applied to octrees is the use of Morton order encoding to provide fast neighbourhood lookups and traversal (Baert et al. 2014).

Some of the octree implementations have been designed to be highly parallel and work on GPUs as well. Crassin and Neyret (Crassin et al. 2009) construct a two-part sparse voxel octree (SVO). The first part only allocates memory for valid leaf nodes (voxels that contain data) and stores the structure of the tree in contiguous memory for all occupied leaf nodes. The second part stores a brick pool in an array of 3D textures each containing a brick (33 blocks of voxels containing the leaf voxel in the centremost texel and its neighbour voxels in the surrounding texels). Each valid leaf node contains an index into the brick pool and as the bricks are stored in a GPU accessible texture, they can exploit hardware accelerated interpolation. This is particularly useful for their use case of computing indirect diffuse lighting to approximate global illumination.

Laine and Karras (Laine & Karras 2011) also create an efficient SVO. Unlike (Crassin et al. 2009), all parts of the voxel data are on this occasion contained within a single data structure. Laine and Karras use this data structure to store contour data to better approximate the geometry of the surface and the final voxel data is inferred from the non-leaf voxels. Instead

of empty regions, this SVO stores 64-bit child descriptors where 32 bits are used for traversal of the tree. This representation resulted in significant reductions in memory usage as well as fast rendering times assisted by quick traversal times.

Kämpe et al (Kämpe et al. 2013) observed that at higher resolutions of voxel grids, the data required to store an SVO was still prohibitive. They presented a method using directed acyclic graphs (DAGs) that can compress SVOs and significantly reduce memory usage. By utilising a bottom-up approach when processing the SVO, they created a DAG by linking identical subtrees of the octree to the same interior node and thereby reducing the number of links within the original data structure. However, this method was only tested with voxels containing binary data and therefore may not create such savings for multi-valued voxel datasets. A different sparse data structure that can be used to store volumetric data is the *brickmap*. Brickmaps have been used for raytracing, in both offline renderers (Christensen & Batali 2004) and, more recently, real-time rendering (Swoboda 2013). Brickmaps contain a sparse list of *bricks* where each brick contains volumetric information for a small set of voxels. The key difference between an SVO and a brickmap structure used for real-time raytracing is that there are only two discrete levels used in a brickmap, a sparse map for looking up bricks and a finer resolution voxel grid, found in each brick's dataset. It is an efficient data structure that can be quickly rebuilt and is therefore useful for more dynamic scenes. This design however does not save as much memory as a standard SVO.

Recently, Museth (Museth 2013) introduced a new sparse data structure called VDB (Volumetric, Dynamic B+ trees) that offers a number of key advantages to volumetric representations. It is cache coherent and demonstrates fast rates of insertion, traversal and deletion operations. Unlike the SVO which is primarily suited for static volumetric elements, VDB's advantages make it ideal for use with dynamic datasets and animations. The data structure has been open-sourced as part of the OpenVDB library (Museth et al. 2013). The VDB data structure has since been expanded upon to work on GPUs by using memory pools and a hierarchical traversal method in the work of Hoetzlein (2016).

2.2.4 Voxel Compression

One of the primary limitations with volumetric data representation is the memory constraint. Large volumetric datasets can consume a prohibitive amount of memory which can create a storage problem. The likelihood of the dataset residing in main memory is also unlikely when the dataset is sufficiently large, and thus can introduce a detrimental effect on performance due to read-write latency between the main memory and the disk storage medium. In order to mitigate the effects of this, there are a number of ways to compress volumetric data to ensure that it is more manageable to use.

Run-length encoding (RLE) is one simple method of compression that can work well with voxel terrains (Golomb 1966). As many procedural terrains are initially constructed using gradient noise functions, they lend themselves well to this compression scheme. Large numbers of adjacent, occupied voxels can be encoded as a run-length and therefore have the overall size of the voxel dataset reduced.

In large scale voxel worlds, many voxel engines dynamically stream in and out chunks of voxels as the player traverses the terrain. As these chunks will be loaded into main memory, it is prudent to ensure that they have a minimal memory footprint. The LZ4 algorithm (Collet 2013) can be used to dynamically compress and decompress data at a very high speed.

Lossless compression algorithms are important when it comes to medical data, such as MRI and CT scans. Lossy compression on such data is unacceptable as accuracy is highly important. Compression schemes that are lossless and specifically designed for volumetric data can be found in the survey by Komma et al. (2007).

2.2.5 Volumetric Rendering

Volume rendering has a number of applications, especially in the medical field. However, for this research the focus will be on the uses of PCG methods to generate volumetric data and render it in real-time. Investigating methods of art directed creation of volumetric data can be especially useful for content creators in order to achieve a desired visual aesthetic.

Wrenninge (2012) details a modern production quality volume rendering pipeline and highlights the significance of voxels and their applications when rendering 3D participating media such as fog, clouds and smoke. While all important elements of the system are extensively described, the result-

ing system is not designed for real time usage, as it focuses on providing straightforward implementation detail over optimised code.

Jönsson et al. (2012) provide a recent survey of illumination in volumetric data. They focus on only surveying interactive methods that could be better suited to real time scenarios depending on their speed. Due to the nature of volumetric rendering, the authors also observe that the creation of algorithms that work well with the GPU are necessary to result in methods that are sufficiently interactive.

Most 3D applications make use of rasterization in order to render polygonal models (Akenine-Moller et al. 2018). To render volumetric objects they can either be converted to a polygonal mesh and rendered with rasterization, or use a number of different volumetric rendering methods. Surface extraction is the process of converting a volume to a rasterizable mesh and is discussed further in Section 2.2.6.

An alternative to rasterization for volumetric rendering is known as *splatting* (Laur & Hanrahan 1991). This technique is particularly well suited to translucent volumes such as fog or smoke. An alpha-blended screen-aligned sprite - a *splat* - is constructed at the position of each occupied voxel. The splats are sorted by their respective depth values in order to render accurately. However, additive blending can be used in order to prevent the need for this sorting process.

A further technique of rendering volumes is by using a slice-based approach, where the volume is separated into layers perpendicular to the viewing direction (Milan et al. 2013). This can be well suited for a GPU as each layer can be represented using a two-dimensional texture and the method can take advantage of the blending hardware. However, many layers may be necessary to render a volume with high fidelity; otherwise the final render can have banding artifacts. Many layers would mean higher fill rate requirements and larger consumption of GPU memory, both of which can affect performance adversely.

Signed distance functions (SDFs) can also be used for volumetric rendering by storing functional representations of geometry at points in 3D space (Frisken et al. 2000). The process of rendering an SDF has been executed by sampling a scene at points along a ray from the camera and subsequently testing for intersections with the function. This method is known as *ray marching* (Perlin & Hoffert 1989) and has been popularised by Shadertoy (Quilez & Jeremias 2013).

It has also seen use in production rendering contexts where it can be used

for convincing, real-time, volumetric effects for video games (Hillaire 2015). Utilising points at a fixed distance from each other is the simplest approach, however this is not efficient when a scene consists of some objects close to the near plane of the camera, and other objects much further away. The number of points required to be able to march the entire scene linearly increases depending on the distance of the furthest object from the camera. Furthermore, when scenes consist of thin geometry, the sampling may miss the surface if the sampling distance is larger than the thickness of the geometry.

Sphere tracing is a notable optimisation that provides a simple way of heavily reducing the number of sampling points required for rendering an SDF-based scene (Hart 1996). This works by the sampling point moving along the ray as much as possible. This distance is computed by intersecting a sphere with the SDF and moving the point along the ray by the amount of the sphere's radius. Keinert et al. (2014) have accelerated this process by introducing an over-relaxation method when marching spheres along a ray. This can offer a wide degree of flexibility and can scale well due to its resolution independence.

The work of Wang et al. (2011) decomposes volumes into a hierarchical tree data structure containing partial SDFs at each tree node. Constructing a suitable SDF for complex volumetric features requires either the use of well-developed tooling solutions or mathematically proficient users. Furthermore, constructing robust meshes from SDFs can be a difficult and slow process to perform on GPUs (Swoboda 2012).

2.2.6 Surface Extraction

Modern GPUs are highly optimised to process polygon-based inputs and do not handle the rendering of volumetric data well. Therefore, it is necessary to be able to extract a triangle mesh representation from a voxel dataset used for a terrain, so that the GPU can rasterize the geometry efficiently.

Many techniques that require surfaces to be constructed from volumetric data make use of the oft-cited Marching Cubes algorithm (Lorensen & Cline 1987). The Marching Cubes algorithm takes a 3D grid as input, with density values at each corner of each grid cell and then generates up to 5 triangles across intersecting edges that exhibit a "sign change". A sign change occurs when one corner's density value is a different sign from its adjacent corner.

However, in terms of generating realistic geometry for terrain, the Marching Cubes algorithm has some limitations. The algorithm itself can exhibit

some ambiguous edge cases. In these cases, the algorithm does not know whether the computed isosurfaces of the data fall inside or outside of the geometry's surface. This results in visible graphical artifacts in the output mesh.

Furthermore, it is difficult to recreate sharp edges that can be found in natural terrains in the form of cliff faces without substantially increasing the resolution of the uniform grid that the algorithm processes. Resulting geometry can look too smooth to represent realistic terrain. Additionally, geometry can exhibit a blocky appearance when rendered. This is due to the algorithm generating small triangles at points of the input data where there are large changes in value (e.g. corners and edges of a mesh). The inability to represent such areas effectively leads to inaccurate meshing and jagged edges. Mesh smoothing is a method of ameliorating this artifact, demonstrated by Swoboda (2012), who uses a Laplacian smooth operation to improve the appearance of real time volumetric fluid simulations.

Raman & Wenger (2008) extend the lookup table for the standard marching cubes algorithm to improve upon these limitations. They introduce an extra label for classifying a vertex of a cube cell in an effort to reduce the number of degenerate triangles created by the original algorithm. As a result, this method creates 38 (6561) entries, instead of 256 entries, in the lookup table. Otherwise, the algorithm remains the same. The extended table did result in a substantial reduction of generated triangles, although the surface extraction did take a significantly longer time. Furthermore, the method can modify the resulting mesh's topology and create non-manifold surfaces, with the authors suggesting a post processing step to repair the mesh in these cases.

One method of maintaining a good mesh topology was suggested by Ho et al. (2005). They suggested Cubical Marching Squares. It functions by unfolding each grid cell into six interconnecting squares and marching each square individually. This has reduced the amount of geometric error compared to the original Marching Cubes algorithm and eliminated the requirement for stitching together cracked meshes. Dual methods of surface extraction carry this name as they generate a dual of the mesh produced by primal methods such as Marching Cubes, meaning that topologically each vertex of the dual mesh corresponds to a face of the primal mesh. As such, they offer the ability to better approximate a surface and reduce the appearance of jagged artifacts prevalent in primal methods of surface extraction. One of the earliest dual methods presented was Surface Nets (Gibson 1998). Similarly to

Marching Cubes, this algorithm operated on a cubical grid. However, it only created a single vertex in close proximity to the surface for every grid cell that intersected the surface and sets of 4 neighbouring vertices were linked to form quads. The method then relaxes the mesh by iteratively translating each vertex towards a point equidistant from the vertex and its neighbours, so long as all vertices remain in their initial grid cell. Gibson suggests that the method can be modified and extended upon by utilising different relaxation schemes.

Dual contouring (Ju et al. 2002) has more recently become a popular choice in generating mesh data for volumetric terrains (Cepero 2010). This is predominantly due to generating both smooth and sharp edged meshes effectively. A mesh is initially generated in the same way as the Surface Nets method. The authors apply their method to a set of Hermite data (consisting of positions and unit normals of the surface) and each vertex is placed at a position that minimises a quadratic error function (QEF). As this method can result in non-manifold meshes, the authors have carried out further work to try and guarantee the manifold nature of the resulting mesh (Schaefer et al. 2007).

As surface extraction is a highly parallelisable task, in order for dual contouring to be more performant in the context of real time simulations, it is sensible to transfer its processing to the GPU. Schmitz et al. (2010) observed that there were two qualities of the original dual contouring algorithm that made this difficult. Firstly, minimising the quadratic error of the vertices was not a trivial operation. Secondly, the method requires knowledge of neighbouring vertices to effectively perform the minimisation process. They suggested a particle-based approach, where forces at each vertex of the grid cell were calculated and summed together to move the dual vertex. This removed the dependency on neighbour information and eliminated the requirement of a QEF and thereby produces a far more parallel approach that results in a very significant increase in speed.

Adaptive Skeleton Climbing (Poston et al. 1998) has also been presented as an alternative form of surface extraction. It generates polygons by working on subsets of voxels in a cubical grid, rather than each individual cell. This creates a mesh with substantially reduced polygons than Marching Cubes as the polygons can be much larger in areas of similar values in the grid. However, the main problem with this approach is that it generates a relatively low resolution mesh as its output that still requires a post processing step for artifact-free meshes.

2.2.7 Procedural Content Generation

Procedural content generation (PCG) is the process of creating assets for use in virtual simulations in an automated fashion by way of "limited or indirect user input" (Shaker et al. 2016). For large-scale games and simulations the process of manual content creation can be an arduous affair. An artist or designer must work through a content pipeline in order for a final asset to be created for a virtual simulation. This process includes beginning from initial concept sketches, modelling low level-of-detail models, adding further details for higher-polygon models, creating high-fidelity textures (including light maps for static, baked lighting as seen in Abrash (1997)) and sometimes animating the model. Instead, procedural content generation can be used at several points in the pipeline to reduce the labour cost involved by computationally generating the outputs. Furthermore, a problem that is present when creating assets for large-scale virtual worlds is the cost and effort involved to author enough varied content. Variation is important so as to prevent uniformity when a user is exploring a virtual world and assist in player immersion and realism. PCG methods can help solve this by modifying parameters of a base-level asset in order to generate potentially infinite variations - depending on the range of the parameters and limitations set by the users. Procedural generation has already demonstrated its viability as it has been used in a number of mainstream computer games and some prominent examples include:

- *The Binding of Isaac* - A "roguelike" game that generates dungeons, enemy positions and types (McMillen & Himsl 2011).
- *Borderlands* - An FPS that generates weapons, their associated stats and their abilities (Gearbox Software 2009).
- *No Man's Sky* - A space exploration game that generates the majority of the game universe, including entire planets, vegetation and creatures (Hello Games 2016).

In this section, various PCG methods are reviewed and have been categorised into *Noise-based* and *Rule-based* methods, as well as a sub-section detailing how semi-automated procedural generation can offer assistance to a user.

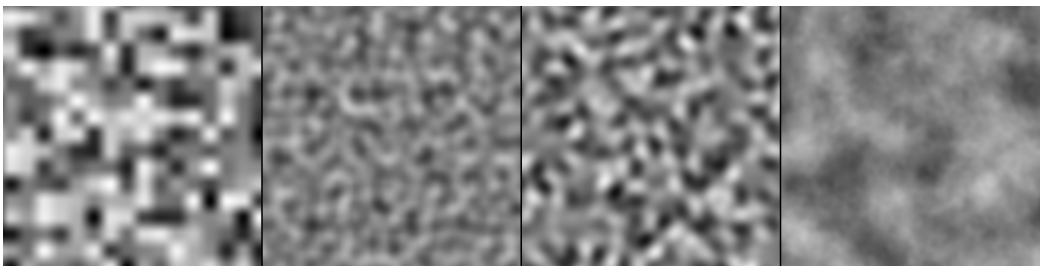

Figure 2.3: Examples of 2D procedural noise (Left to right: Value, Perlin, Simplex, Perlin FBM)

Noise-based

One of the fundamental features of PCG is the use of randomness to introduce variance between generated content. This is primarily achieved by the use of different noise algorithms. Some examples of procedural, 2D noise can be seen in Figure 2.3. The cornerstone of noise that has seen use in many different domains is Perlin noise (Perlin 1985). Perlin noise falls into a class of noise known as gradient noise, where the resulting noise value is an interpolation of other noise values based on pseudo-randomly generated gradient vectors. This results in a smooth-looking noise and prevents sudden changes in noise values when values are expected to be close together.

However, Perlin noise is slow to compute for higher dimensions and exhibits recognisable artifacts in its output in the form of isotropic patterns. Simplex noise (Perlin 2002) was created by Perlin to combat this. Gustavson (2005) explains the implementation of simplex noise further in his work. It works by interpolating gradients on a grid of "simplices", where a simplex is the simplest primitive found in a dimension e.g. a triangle in 2D and a tetrahedron in 3D. It has no obvious isotropic artifacts, offers a continuous and cheap to compute gradient, and has faster performance at higher dimensions ($O(n^2)$, compared to $O(2^n)$ for Perlin).

Other classes of noise include value noise and cellular noise (Ebert et al. 2002). Value noise assigns a random value to each point in a lattice and then performs a linear interpolation between them. Cellular noise randomly distributes a set of feature points and the noise value is computed by calculating the distance from a point to the nearest feature point. Different noise types can be generated in cellular noise by using a range of distance

heuristics (e.g. Manhattan, Chebyshev, Euclidean), as well as selecting the nth closest feature point (Worley 1996).

Noise can also be generated from within the frequency domain. By editing the spectral information of the noise function, the look of the resulting noise texture can be modified to create values with the appearance of different materials (Lagae et al. 2009). However, adjusting the power spectrum of a noise image can be unintuitive. Galerne et al. (2012) realised this and offered a detailed set of examples of different materials produced by specific frequencies. Noise types are particularly useful for the generation of textures. By applying mathematical operators and combining different noise textures, many different material types can be created, including wood, marble and brick. Procedurally generated textures have been used in relevant middleware such as *Terragen 3* (Planetside Software n.d.) and, more recently, *Substance Designer 5* (Allegorithmic n.d.).

Other types of noise can be used to create the illusion of moving fluids. Perlin & Neyret (2001) introduced one such example as flow noise. In gradient noise a time parameter can be used to rotate the pseudorandom gradients to simulate fluid flow. However, this is not particularly applicable to particle systems or geometry as it is a texture-based approach.

In an effort to combat the limitations of flow noise, Bridson et al. (2007) introduced curl noise. This is a very simple extension to add to any noise function, where fluid dynamics are simulated by calculating the curl of a generated vector field. The results of curl noise can be a convincing approximation of real fluid dynamics, which is of particular interest in real-time games and simulations (Swoboda 2012).

Rule-based

However, PCG is not only limited to the creation and utility of different noise types. PCG can also use sets of rules and models. For example, Angelidis et al. (2006) introduce a model for smoke simulation that can be controlled by manipulating currents, adding swirling effects and applying some noise, whilst maintaining the realistic appearance of smoke.

A formal grammar consists of a set of axioms (or rules) that recursively expands an initial state for a number of iterations. This results in a structured and repeatable result, so long as the same initial state and rules are used, and variations can be created by simply altering the initial input state.

Lindenmayer systems, or L-systems, and their extensions have been fre-

Figure 2.4: A "Barnsley fern" generated with a 2D L-System

quently used in order to create smaller objects such as trees and foliage (Prusinkiewicz & Lindenmayer 2012), an example of which can be seen in Figure 2.4. They have since been used to generate road networks for urban environments as well as entire cities (Parish & Müller 2001). Lipp et al. (2009) introduced a multi-threaded version of L-systems that performs faster than the single threaded counterpart as it avoids inter-thread communication and is completely lock free. As such, it can be easily deployed to massively parallel architectures such as GPUs. The advantage of this is that very complex L-systems with many rules can be used to generate results quickly and efficiently.

Another form of grammar being adopted for procedural generation is the shape grammar (Stiny 1980). Shape grammars function much like L-systems and other grammars, where an initial shape is recursively applying rules that govern the shape's transformation, as shown in Figure 2.5. Selection of a rule is dependent on the content of the intermediary steps of the transformation. Wonka et al (Wonka et al. 2003) introduced the concept of a split grammar, an extension to a shape grammar where rules are composed of basic shapes and more detailed decompositions of the basic shapes. They used the grammar to quickly create procedural architecture and the work has since been expanded by Muller et al (Müller et al. 2006*a*). A visual editor to author grammars for procedural architecture was developed by Lipp et al. (2008) that offers content creators the ability to develop their own grammars with minimal effort.

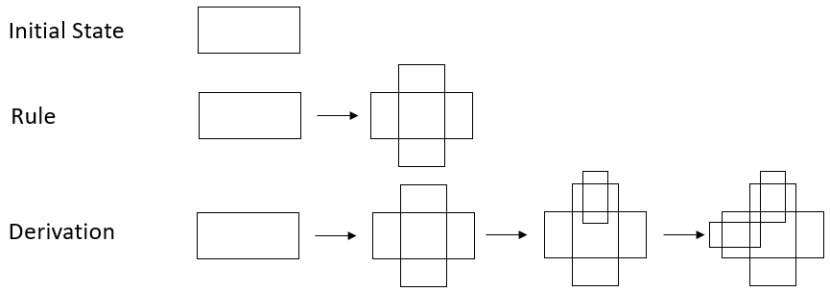

Figure 2.5: Example of a simple shape grammar manipulating an initial state.

Procedural Assistance

Rule-based systems discussed in the previous section can also contribute to methods that combine user direction with variation offered from procedural generation. This is a particularly useful feature of any content creation tool as it provides controls to otherwise arduous tasks. Nowrouzezahrai et al. (2011) developed a tool to allow artists control over volumetric light trails. By controlling the directions of the trails, the light transport and scattering equations are dynamically calculated to provide the required design. These kinds of controls are examples of procedural algorithms being able to assist with manually designed content.

Procedural architecture is also an area where procedural generation is used to assist in creating buildings, instead of completely automating the process. Kelly & Wonka (2011) create a set of procedural tools to enable the modelling of complex features of buildings. Dang et al. (2014) focus on the manipulation of building facades. They do so by manually marking regions of an input facade and, using the data, procedurally create resolution-independent textures whilst maintaining awareness of the structural relationships within the input. Beneš et al. (2011) use the idea of procedural assistance for procedural modelling of tree structures. By allowing a user to manipulate initial guide diagrams, their system uses L-Systems to procedurally generate trees for virtual simulations by using the guides as a basis. This is a further form of art-directable content being given a procedural sheen in order to generate interesting, yet controlled, variants of geometry.

Specifically regarding terrains, Tasse et al. (2014) introduce a tool that

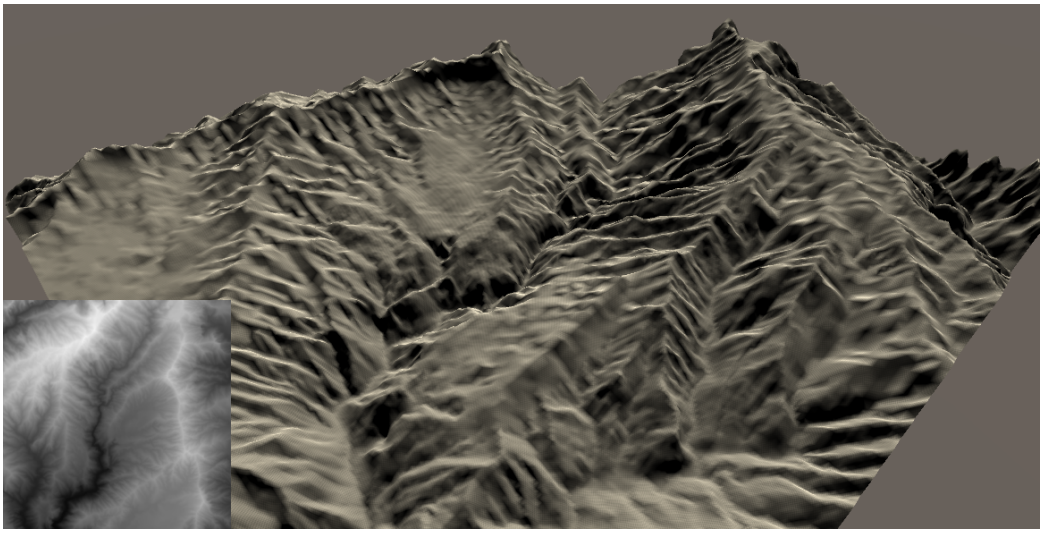

Figure 2.6: Example of heightmap (lower left) and its generated terrain mesh.

allows a user to sketch a terrain from a first-person camera. The sketched lines are then processed and the terrain is procedurally created using the sketched marks as guides.

2.2.8 Terrain Generation

There has been a significant amount of research in terrain generation, particularly when it comes to generating polygonal meshes, as modern GPUs are optimised for rendering polygons. Terrain data can be represented as a polygonal mesh or as a volumetric data set. It should be noted that volumetric terrain is still commonly rendered as a set of polygonal meshes generated by applying various surface extraction methods on the voxel representation of the terrain. This is necessary as volumetric data can become inordinately large at higher grid resolutions. A polygonal mesh that provides an accurate representation of the surface of the data and falls within the constraints of a GPU is important, especially for real-time rendering.

The most common ways of generating terrains use heightmap-based methods (Miller 1986). Heightmaps, in their most basic form, are single channel (greyscale) textures where each texel represents a height of the corresponding point on the terrain surface, as shown in Figure 2.6. Heightmap textures can

be user generated however, as this is an arduous and manual process, procedural methods are utilised to automate the creation of terrains. Heightmaps can be generated by using octaves of noise functions or more complex methods, such as applying filters to a noise texture in order to simulate erosion. A survey of heightmap based methods can be found in (Smelik et al. 2009). Since heightmaps consist of elevation data, their primary limitation is the ability to represent concave features such as overhangs, natural arches and caves.

Heightmaps can either be manually created or automatically generated. Current popular game engines, such as Unity (Unity n.d.), Unreal Engine (Epic Games n.d.) and CryEngine (Crytek n.d.), have sets of brushes used to modify a heightmap manually. Brushes include the ability to raise, lower and level out the terrain to various heights.

Automatic generation of heightmaps can be achieved procedurally using noise as described in Section 2.2.7. Fractal noise created from multiple octaves of Perlin or Simplex noise can achieve satisfactory results. However, as noise creates heightmaps that are pseudorandom in nature, the terrain outputs are too random to be reflective of natural environments without further artistic input.

Recently, Parberry (2015) analysed a large area of real world terrain and observed that the gradient distribution was exponentially distributed. By making a modification to the Perlin noise algorithm, the author found that more realistic terrains could be created. Landscapes modelled purely on Perlin noise distributed large gradients everywhere and therefore lacked in areas that could be identified as landmarks. An exponential distribution of gradients created smoother terrains, whilst adorning the landscape with cliffs and mountainous regions.

Fractal terrains can also be created with other means. For example, Fournier et al. (1982) introduced a method that modifies the grid mesh directly. Fractal midpoint displacement (or commonly known as the diamond-square algorithm) processes a grid of vertices. It then alternates between a "diamond step" and a "square step", where vertices at midpoints of these shapes are displaced by a random value, as shown in Figure 2.7. This algorithm can automatically produce a random looking terrain with little initial effort and while this is satisfactory for many applications, the generated terrains do not vary in terms of features, i.e. the entire terrain seems mountainous with no areas of flat plains. This is not the case for real world terrains, where features such as craggy mountain ranges, hilly plains and

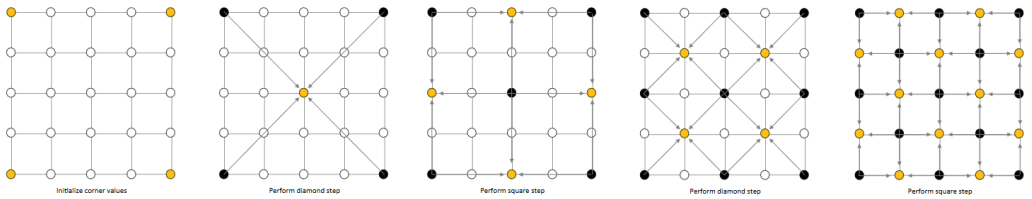

Figure 2.7: Visualisation of the diamond-square algorithm for generating fractals. Image from: *https://en.wikipedia.org/wiki/Diamond-square_algorithm*

smooth shorelines exist together. Furthermore, as this algorithm runs completely automatically, it does not offer the user any control over how the resulting terrain will appear. Any custom modifications to the terrain have to be done by editing the mesh directly. It is important to be aware of such algorithms and the benefits that they can offer, but as this research focuses on user driven procedural generation, similar fully automatic algorithms are not considered to be part of the final system.

2.3 Related Work

This section contains existing work that relates to the domain of terrain generation. Critical assessments of the relevant literature are provided and their solutions to related terrain generation problems are reviewed.

2.3.1 Erosion Simulation

Procedural models that are particularly suitable to terrain generation are various erosion models. In the field of geomorphology there are a number of factors that can affect how a terrain is formed and manipulated over the course of time. Factors include the surrounding climate (e.g. temperature levels, and frequency and intensity of precipitation), the composition of the soil, vegetative cover and topography. As there are a number of factors to consider when creating a computational model of erosion, many existing models tend to select only one or two to simulate.

Some erosion models are directly applied to heightmaps. Anh et al. (2007) use the GPU to model terrain erosion due to water sources depositing sedi-

ments. This is done via a pixel shader that takes two textures that contain relevant erosion parameters. However, the presented approach does seem to be rather slow when simulating the erosion. The shader provided is a reverse version of their model as they observe that pixel shaders cannot randomly access values in the heightmap. With the advent of DirectX 11, this is no longer the case as pixel shaders (like compute shaders) have the ability to randomly access textures. Thus, a modern implementation that better reflects their model and makes appropriate GPU optimisations could be implemented and may be faster on current GPUs.

Beneš & Arriaga (2005) present a method to create mesas (or table mountains) by utilising two separate heightmaps that represent different materials. The first heightmap represents hard material that cannot be eroded, whereas the second represents erodible soft material. During the simulation the softer material is reduced and distributed to neighbouring cells of the heightmap via gravity driven diffusion. This continues until either all of the material has been distributed or equilibrium has been reached with a neighbouring cell. Benchmarks were not provided for the speed of this method. However, the results produced seem realistic for this particular terrain feature and demonstrate the capability of erosion methods on simple heightmap based methods.

Beneš (2006) presents a volumetric method that models hydraulic erosion for terrain. He combines fluid dynamics with sediment transportation. By utilising voxel data both fluid and sediment information can be stored in a single voxel. This can then use the canonical fluid dynamics equations to model the movement of the fluid resulting in sediments being transported around the terrain and eroding certain parts of the terrain over time. It is proposed that only small scale erosion can happen as the amount of data required for storing large volumetric terrains is excessively large, although this may no longer be the case with the use of newer data structures as described in Section 2.2.3, such as sparse voxel octrees.

Another volumetric approach is described by Hudák & Durikovic (2011) where they also take the fluid dynamics route. Instead of creating a voxel grid, they generate a large number of particles with an associated material. The movement of these particles is then simulated using smoothed particle hydrodynamics, which results in realistic, volumetric mass movement of terrain segments. However, this method is far from a real time solution and does require that a surface be extracted from it afterwards.

2.3.2 Terrain Generation

While heightmaps are relatively straightforward and intuitive to create terrains, some features of real world landscapes can be more complicated to generate using heightmaps alone. Features such as caves and overhanging cliffs cannot be represented by single height values and thus call for a different method of creation. One workaround was presented by Gamito & Musgrave (2001), where an existing heightmap was warped using a flow field (a vector field representing velocities at points in a specified space). This resulted in overhanging terrain being generated in a simple manner. However, the main limitation of this method is that a specific flow field needs to be defined before the deformation takes place and this can be a complex process.

Volumetric terrains can effectively and intuitively represent the structures that are not feasible in heightmap based methods, as the data is stored per relevant point in space. This is a substantial advantage as it allows the formation of realistic terrain features necessary to enhance visual fidelity for modern games.

Minecraft (Mojang 2011) is an example of a well-known contemporary game that demonstrates the use of volumetric terrain. Worlds are made of cubes and stored within "chunks" of voxels. A chunk consists of a subset of the entire voxel data and in Minecraft, a chunk's dimensions are 16x16x256 voxels resulting in a total count of 2^{16} voxels per chunk. While Minecraft is not representative of terrain in the real world, other works use similar concepts to produce voxel-based terrains.

Santamaría-Ibirika et al. (2013) demonstrate one such example as they utilise a chunk-based approach to procedurally generate small voxel based terrains. Multiple materials can be added to the terrain as this process is simplified by separating the voxels into chunks. This method is highly user driven as materials are provided by the user at design time and the generator creates the terrain based on a set of input textures that contain the required material data. However, for generating a considerably small volume with a resolution of 50^3 voxels, generation takes over 300ms with their online algorithm. This time is prohibitively slow for real time usage.

Peytavie et al (Peytavie et al. 2009) use a hybrid of a stack-based approach to store different materials and a signed distance field to assist in sculpting the terrain. A benefit of this description is that an erosion approximation can be trivially and efficiently implemented to model more realistic

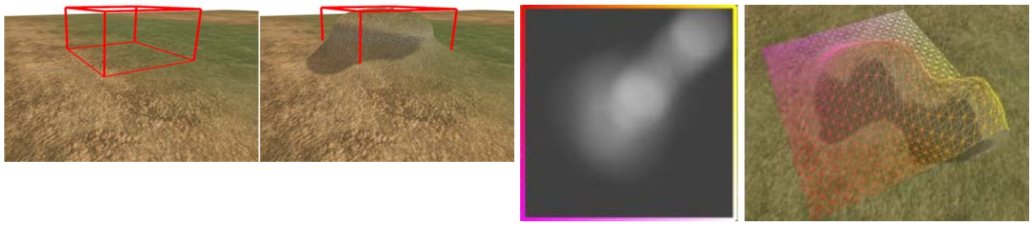

Figure 2.8: Koca & Güdükbay (2014) process of terrain editing, constructing a coarse voxel model (left, middle left) and refining sections using a heightmap (middle right, right).

terrain. Furthermore, the signed distance field representation allows for a number of Boolean operations to be applied to the landscape, which results in intuitive controls for the designer. This means that sculpting and approximating erosion can be trivially and efficiently implemented to model more realistic terrains. The terrain is rendered in real time by the work of Löffler et al. (2011). They generate a level-of-detail hierarchy to ensure that distant parts of the terrain are drawn with less detail and also perform a surface extraction of the terrain data entirely on the GPU. However, this process does require that portions of the resulting mesh be stitched together with neighbour mesh segments which can adversely affect performance. Scholz et al. (2013) improve upon this by developing a hexahedral based method of rendering volumetric terrain that requires no stitching and produces a mesh consisting of triangles with levels of detail.

Another hybrid approach is presented in Koca & Güdükbay (2014), which combines voxels and heightmaps. This approach creates a coarse voxel grid that can represent caves and overhangs and splits the entire voxel grid into patches, as seen in Figure 2.8. The authors then extract the surface to form the polygonal geometry required for rendering. The resulting geometry is subsequently displaced by heightmaps assigned to each patch. The process allows them to represent terrains with volumetric features, whilst maintaining a low voxel grid resolution.

Becher et al. (2017) demonstrated a user-directed method of generating terrain features. User-generated curve-based primitives are superimposed onto existing terrain geometry. These curves are then voxelised in order to produce a signed distance field representation of the terrain which is then rendered. This method requires a separate voxelisation step for the feature

curves. Furthermore, the diffusion based terrain generation is slow for real-time rates.

2.4 Summary

While there is a significant amount of literature surrounding procedurally created terrains, this primarily pertains to heightmap-based terrain generation. There is comparatively little surrounding the generation of terrain features that cannot be modelled with heightmap-based approaches. Whilst there are methods to workaround these limitations to produce specific features such as overhangs (Gamito & Musgrave 2001), a volumetric terrain representation is the ideal method of creating these missing components.

Current procedurally generated volumetric terrains are predominantly reliant on noise-based generation with a modicum of user-directed control. Furthermore, most work focuses on the generation of complete terrains, rather than individual features. There is also sparse work on the use of rule-based procedural generation for terrains, and no research to our knowledge on the generation of volumetric terrain features, such as overhangs, caves and arches, using sets of rules.

Based on this literature review, gaps that can be identified in the current body of terrain generation literature are:

- Rule-based and other procedural methods of generating overhangs, arches and caves for terrains.
- User-directed techniques to parametrically control procedurally generated volumetric terrain features.

The most relevant literature that will inform the remainder of this thesis, along with their respective conclusions and comments, has been highlighted as follows:

Citation: *Gamito & Musgrave (2001)*
Conclusions: Overhangs can be constructed with by warping the highest elevation datapoints of a heightmap using a preconstructed flow field.
Comments: Creation of a flow field is an arduous process. Warping the terrain to create overhangs that fit a specific aesthetic desired by the user is difficult.

Citation: *Peytavie et al. (2009)*
Conclusions: The combination of material stacking and signed distance field rendering allow for intuitive creation of volumetric terrain features.
Comments: Surface extraction of the terrain data requires that meshes be stitched together which can negatively affect performance. Signed distance field rendering may be less performant on less capable GPUs and may be too expensive to fit into a modern real-time simulation graphics pipeline.

Citation: *Santamaría-Ibirika et al. (2013)*
Conclusions: Creation of volumetric terrains with multiple materials, driven by user-defined texture data representing material information for the terrain.
Comments: Prohibitively slow at low-resolution voxel grids means that is not suitable for real-time usage.

Citation: *Koca & Güdükbay (2014)*
Conclusions: Hybrid method using low-resolution voxel grids to represent areas of caves and overhangs. The extracted surface mesh from this representation is then displaced with local heightmaps to generate detail.
Comments: Laborious process of defining heightmaps for specific terrain feature areas means that it is difficult to achieve a user's specific art direction.

Citation: *Becher et al. (2017)*
Conclusions: Intuitive, user-directed method using hand drawn curve-based primitives, which then has diffusion-based methods applied to the resulting mesh to generate topologically realistic terrain.
Comments: Requires the use of a separate voxelisation step. Diffusion based method is slow for low resolution voxel grids, so this is not a particularly interactive method.

Chapter 3

Voxel Grammars

3.1 Introduction

Generation of terrains can be a particularly important process when creating realistic representations of virtual worlds, as found in computer graphics simulations, feature films and computer games with outdoor environments. So far, there has been a considerable amount of research in this domain, which ranges between fully automated and semi-automated methods.

While it is now feasible to create massive virtual worlds, the tasks of designing the terrain, populating the world with content, and, finally, ensuring it does not feel empty or barren, continue to be very time consuming processes. Procedural content generation (PCG) has many applications and has proven valuable to designers due to its ability to algorithmically produce content such as the generation of textures, geometry and animations (Ebert et al. 2002) so it can greatly improve the cost efficiency of populating a virtual environment. PCG will be used in this research to assist designers and shorten the length of time to create large scale landscapes.

Traditionally, terrains are defined by their surface details using a texture-based approach representing a top-down, two-dimensional view, called a *heightmap*. However, the details beneath the terrain surface have a significant impact on how the terrain is formed and its eventual appearance. This research uses volumetric data to represent terrain. This is important as it provides meaning to the details that are not visible to the user. Various factors, such as soil type and material density, govern how terrains are created in the real world. This can be modelled accurately when a voxel-based

approach is utilised. A further advantage of this approach is that both constructive and destructive methods to terrain creation can be adopted without being concerned about real-time polygon mesh editing, i.e. a surface can be extracted from the voxel data after the data has been constructed to the designer's liking.

This chapter proposes a procedural, voxel-based approach to assist users in the generation of key terrain features, such as overhangs and caves. The presented method expands the concept of *shape grammars* to a volumetric space and explains the process employed to create terrain features. We develop specific rulesets that are applied over a voxel dataset in order to create such features on the CPU. We also describe some good practices to be utilised when developing these rulesets. The final terrain mesh is generated at real-time frame rates by using our GPU-based surface nets algorithm. Furthermore, we present timings and memory usage from our results for the generation of the voxel data using different rulesets plus the performance statistics of our GPU surface extraction algorithm.

This chapter was published in the Entertainment Computing journal (Dey et al. 2018).

The proposed method involves aspects of multiple domains, including terrain creation, data structures to manage voxel datasets and rule-based procedural generation methods.

Volumetric representations of terrains enable the creation of features such as overhangs and caves. Peytavie et al. (2009) combines a voxel-based terrain with signed distance fields that allows for a highly user-directed process. However, SDF rendering can be slower on less capable GPUs and may be too expensive to slot into a real-time workflow.

To our knowledge, there is no available literature for a rule-based approach to procedural generation of terrains. However, rule-based procedural generation is prevalent in automatic guided creation of cities (Parish & Müller 2001) and buildings (Müller et al. 2006*b*), utilising concepts of L-systems (Prusinkiewicz & Lindenmayer 2012) and shape grammars (Stiny 1980), respectively. We borrow the concepts of shape grammars in our method to construct rules that can be used to create volumetric terrain features.

3.2 Voxel Grammars

This section describes how the concept of voxel grammars was formulated in this thesis. The individual components that comprise a grammar are detailed, as well as the process of how they are used during the generation phase. Voxel grammars have been inspired from voxel space automata (Greene 1989), where voxels are generated via a set of predefined rules. Greene primarily uses this method to simulate plant growth. However, adding detail to existing geometry is a further application of this approach.

Our method extends the concept of recursively manipulating volumetric data governed by a set of rules to the formation of specific features found in real world terrains. The method operates on a voxel grid that defines the terrain boundary. Each voxel is represented by a density value of the terrain material contained within it. Voxels are generated within the grid to create an initial state. The initial state in this work was constructed by voxelising a heightfield generated with 3 octaves of 2D Perlin noise, as it is a common way of generating plausible procedural heightmap-based terrain. We use multiple octaves at differing frequency values to ensure the initial terrain contains a sufficient balance between low-frequency and high-frequency details, so that it is not too smooth or too noisy, respectively. The populated voxel grid is then derived using a *sliding window* approach, checking whether the subset of voxels within the window satisfies any rule criteria. If a matching rule is found, then its respective transformation is performed on the voxels. The window is subsequently repeatedly offset by a user-defined stride parameter. The derivation process is then repeated for a number of iterations (exposed as a parameter to the user). Furthermore, the user can define a start and end position within the voxel grid to determine which section of the grid the grammar is applied to. The combination of bounding values, the sliding window stride and symbol grid sizes can reduce the state space for a large voxel dataset.

The voxel grammar presented in this chapter can be represented formally as:

$$
\begin{aligned}
G &= f(V, R) \\
R &= \langle s \in S, t \in T \rangle \\
v &\subseteq V \\
V' &= \forall v \in V \begin{cases} t(v) & \text{if } s(v) = True \\ v & otherwise \end{cases}
\end{aligned}
\tag{3.1}
$$

Where the voxel grammar G is a function acting on the set of voxels V with a predetermined ruleset R. The ruleset consists of a collection of tuples containing a symbol function s from the set of all available symbols S, and a transform function t from the set of all available transforms T. v is a subset of the voxels determined by the stride and the bounds of the aforementioned sliding window. For all of the selected subsets of voxels, t is applied to v if s successfully matches the subset of voxels, otherwise v remains unchanged. V' is the resulting set of voxels after valid transformations have taken place.

3.2.1 Rules

The rules within the class of grammars that we have developed consist of three components - *symbols*, *transformations* and *priority* values. Symbols consist of a list of predicates to satisfy and transformations contain a list of mutations to apply to a subset of voxels. This can be seen as akin to a rudimentary programming language to a certain extent, as symbols and transforms are, respectively, analogous to conditional statements and intrinsic functions. Rules also possess a priority value that governs the probability that the rule will be selected for execution in the rule matching stage of the algorithm.

The voxel grammar $\mathbf{G}$ can be further formalised using L-system notation.

$$
\mathbf{G} = \langle V, \omega, P \rangle \tag{3.2}
$$

Where ω is the initial state of the voxel grid and P is a set of rules that transform subsets of voxels in the form (s, t). $s \in V$ is the list of predicates, which we refer to as a *symbol* and t is the *transform* to apply to the subset of voxels. V is the set of all tensors of size $I \times J \times K$ of predicates. The predicates operate over the domain of real values and thus effectively enable the use of a theoretically infinite alphabet. As the values in a voxel grid in our

implementation are stored as a single 32-bit floating point value, realistically the expressive range of the grammar is subject to the size and precision of the voxel values stored by the underlying voxel engine.

3.2.2 Symbols

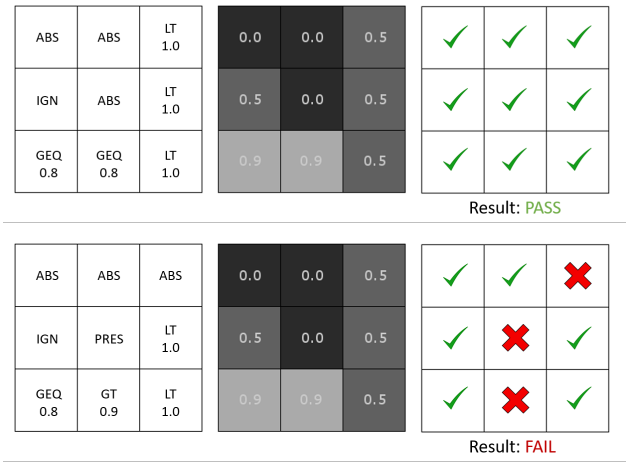

Figure 3.1: Examples of simple two dimensional symbols. Rule symbols (left), input voxels (centre) and results (right)

A rule's symbol is a list of conditions in the form of an $I \times J \times K$ array that determines whether the rule's transformation will be executed. Each symbol entry in the array consists of an *operator* and a *data value*. The operator refers to the type of condition being checked and uses the data value as a comparator to the input voxel value. In order to ensure flexibility, there are a number of operators that have been implemented and their descriptions can be found in Table 3.1. Illustrations of two-dimensional symbols can be found in Figure 3.1.

Symbol Operator	Description
IGN	Ignore - Always returns true.
PRES	Present - Passes if voxel density value is greater than 0.
ABS	Absent - Passes if voxel density value is 0.
EQ	Equals - Passes if voxel density value is equal to the symbol value.
NEQ	Not Equals - Passes if voxel density value is not equal to the symbol value.
LT	Less Than - Passes if voxel density value is less than the symbol value.
LEQ	Less Than or Equal - Passes if voxel density value is less than or equal to the symbol value.
GT	Greater Than - Passes if voxel density value is greater than the symbol value.
GEQ	Greater Than or Equal - Passes if voxel density value is greater than or equal to the symbol value.

Table 3.1: List of symbol operators

3.2.3 Transforms

Figure 3.2: Example of a simple two dimensional transformation. Rule transform (left), input voxels (centre) and output voxels (right)

A transformation consists of a list of manipulations in an array with the same dimensions as the rule's symbol and is only applied to the voxel grid when the symbol's criteria have been fulfilled. Similarly to symbols, transformations also consist of an *operator* and a *data value*. The selected voxel's density is manipulated in a way determined by the type of operator being used plus the data value. Descriptions of the operator types can be found in Table 3.2 whilst Figure 3.2 demonstrates a simple example of a transformation.

Transform Operator	Description
NOP	Does nothing to the value of the selected voxel.
SET	Sets the selected voxel value to the transform value.
ADD	Adds the transform value to the selected voxel value.

Table 3.2: List of transform operators

3.2.4 Rule Priority

As grammars become larger and more complex, there can be times where a set of input voxels can satisfy the conditions for multiple rules. In our implementation, each rule contains a priority value as part of its parameters. When there are multiple matching rules, the priorities are sorted and the rule with the highest priority value is selected. If the priorities match, the selected rule is chosen stochastically from the matches. The greatest priority

is used in order to simplify the creation of grammars, as this allows designers to create rules using a more intuitive approach by introducing more control in the variations, instead of relying on a probabilistic method.

3.2.5 Grammar Construction

During the course of constructing grammars, some observations were made about the effects that certain components of rules had on the resulting voxels. When some rules were matched with the voxel grid, an issue that arose was the repeating, uniform patterns. For natural looking terrains, this symptom is usually undesirable. One method of alleviating this is to introduce some variance to the rules. This can be achieved by creating a rule with the same symbol values and the same rule priority value, but with a transformation consisting entirely of *NOP* operators. This method exploits the mechanisms of our rule-matching system: as the priority value remains the same, when the appropriate symbol is matched, the rule selector will either apply a transformation or do nothing based on a random probability.

When developing grammars, it is also prudent to be wary of overuse of the *IGN* symbol operator. A symbol consisting entirely of these operators should almost always be avoided, as it matches with the entire voxel grid. This can result in severe consequences, where transformations are applied globally to the entire voxel grid which is typically not what the designer intended.

3.2.6 Derivation Process

The final grammar processing method occurs in two stages: *rule matching* and *replacement*. The rule matching stage iterates through the ruleset and checks whether the rule's symbol matches the group of currently selected voxels. Firstly, the rule with the largest dimension symbol is found and a sliding window of this size is passed along each axis. The voxels contained within this window are the currently selected voxels. These are used to compare against the symbol of all rules that have the same symbol dimensions. If there are matches, then the replacement stage occurs. This process continues until all rules have been queried. Pseudocode for this process can be found in Algorithm 3.1. Replacement is simply the process of transforming the set of matching voxels with the selected rule's transformation.

Algorithm 3.1 Pseudocode for deriving the voxel data.

Require: Rules sorted in descending order of symbol dimensions
  for all iterations **do**
    for all *rule* in *rules* **do**
      voxelSubset $\Leftarrow$ Select voxels of size *rule.symbol.dimensions*
      while *voxelSubset* position $\neq$ voxel grid bounds **do**
        if *voxelSubset* satisfies *rule.symbol* **then**
          Add *rule* to *matchArray*
        end if
        Move *voxelSubset* along axis
      end while
    end for
    if *matchArray* $\neq$ empty **then**
      rule $\Leftarrow$ select from *matchArray*
      Execute $rule.transform$
    end if
  end for

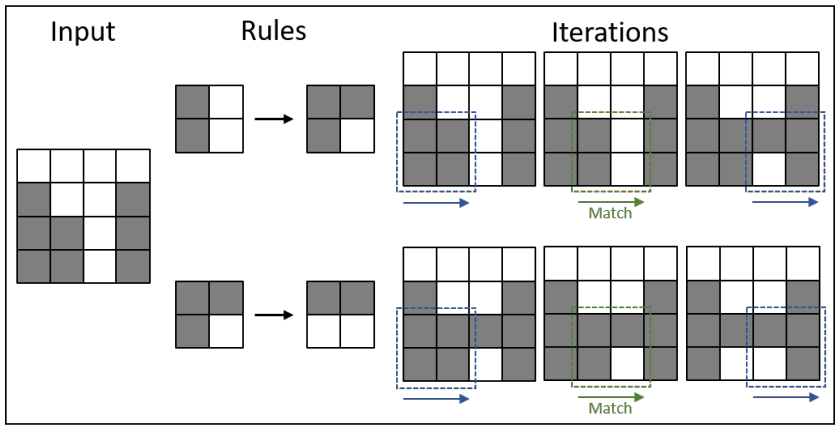

Figure 3.3: Simplified voxel grammar derivation process.

A simplified illustration of a voxel grammar in two dimensions is shown in Figure 3.3, where two rules are matched on a set of input voxels and their respective transforms are used to output a changed set of voxels.

3.3 Method and Implementation

This section discusses how grammars were designed for use with the proposed method. Explanations are provided as to why each rule was constructed. The topologies resulting from cave and overhang formation tend to be very different and, as a result of this, two grammars were constructed with different rulesets to enable the creation of these terrain features. Each rule in both rulesets is given the same priority value as all of their respective symbols are different. Thus, there are no other rules to choose between when the rule selection portion of the derivation process takes place.

Grammars for each feature were constructed by first breaking down the formation of a feature into steps and then generating rules that corresponded to those steps by trial and error. After numerous iterations, resulting rulesets for overhangs and caves can be found in Tables 3.3 and 3.4.

3.3.1 Cliffs and Overhangs

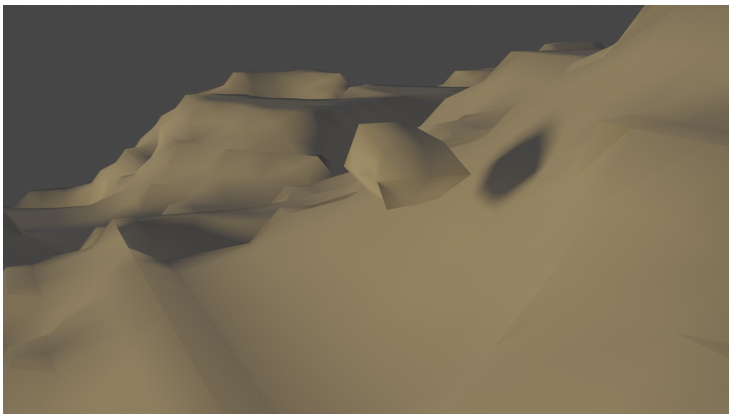

Figure 3.4: Example of impossible floating terrain segment.

A feature found in rich terrains is the formation of naturally occurring

Rule 1						
Size: [3, 3, 1], Weight: 1.0						
Symbol				**Transform**		
GEQ 0.5	LT 0.5	IGN		NOP	NOP	NOP
GEQ 0.5	LT 0.5	IGN		NOP	NOP	NOP
GEQ 0.5	LT 0.5	IGN		ADD -0.5	NOP	NOP

Rule 2						
Size: [3, 3, 1], Weight: 1.0						
Symbol				**Transform**		
GEQ 0.5	GEQ 0.5	IGN		NOP	NOP	NOP
GEQ 0.5	LT 0.5	IGN		ADD -0.5	NOP	NOP
GEQ 0.5	GEQ 0.5	IGN		NOP	NOP	NOP

Rule 3						
Size: [3, 3, 1], Weight: 1.0						
Symbol				**Transform**		
GEQ 0.5	GEQ 0.5	GEQ 0.5		NOP	NOP	NOP
GEQ 0.5	LT 0.5	IGN		NOP	ADD 0.5	NOP
IGN	LT 0.5	IGN		ADD -0.5	NOP	NOP

Rule 4						
Size: [3, 3, 1], Weight: 1.0						
Symbol				**Transform**		
GEQ 0.5	GEQ 0.5	IGN		NOP	NOP	ADD 0.5
GEQ 0.5	GEQ 0.5	IGN		NOP	NOP	ADD 0.5
GEQ 0.5	LT 0.5	IGN		NOP	NOP	NOP

Table 3.3: Ruleset for generating overhangs

Rule 1						
Size: [3, 3, 1], Weight: 1.0						
Symbol				**Transform**		
GEQ 0.5	GEQ 0.5	LT 0.5		NOP	NOP	NOP
GEQ 0.5	GEQ 0.5	LT 0.5		NOP	ADD -0.5	NOP
GEQ 0.5	GEQ 0.5	LT 0.5		NOP	NOP	NOP

Rule 2						
Size: [3, 3, 1], Weight: 1.0						
Symbol				**Transform**		
GEQ 0.5	GEQ 0.5	GEQ 0.5		NOP	ADD -0.5	ADD -0.5
GEQ 0.5	GEQ 0.5	LT 0.5		NOP	ADD -0.5	ADD -0.5
GEQ 0.5	GEQ 0.5	GEQ 0.5		ADD -0.5	ADD -0.5	NOP

Rule 3						
Size: [3, 3, 1], Weight: 1.0						
Symbol				**Transform**		
LT 0.5	LT 0.5	GEQ 0.5		NOP	NOP	ADD -0.5
GEQ 0.5	GEQ 0.5	GEQ 0.5		ADD -0.5	ADD -0.5	ADD -0.5
LT 0.5	GEQ 0.5	GEQ 0.5		NOP	ADD -0.5	NOP

Rule 4								
Size: [4, 4, 1], Weight: 1.0								
Symbol					**Transform**			
LT 0.5	LT 0.5	GEQ 0.5	IGN		NOP	NOP	ADD -0.5	NOP
LT 0.5	GEQ 0.5	GEQ 0.5	IGN		ADD -0.5	ADD -0.5	ADD -0.5	NOP
GEQ 0.5	GEQ 0.5	GEQ 0.5	IGN		NOP	ADD -0.5	ADD -0.5	NOP
IGN	IGN	IGN	IGN		ADD 0.5	NOP	NOP	NOP

Table 3.4: Ruleset for generating caves

cliffs and overhangs. Similar to caves, they are formed due to erosion from water and weathering effects. Assuming an input of a solid, vertical wall of voxels, the ruleset contains one rule to remove voxels towards the base of the wall and another to add some detail. On their own, these rules produce some unnatural looking features (regularly spaced voxels) and some impossible features (floating segments of terrain) as seen in Figure 3.4. In order to mitigate this, another rule was added that acted as a "clean up" action.

3.3.2 Caves

The formation of caves is governed by the erosion of the different types of rock. Existing fractures already within the terrain structure are eroded by groundwater. In speleological literature, cave topology can be defined as *vadose* or *phreatic* and is determined by the proximity to the groundwater source (Bretz 1942). This research emulates the formation of such types of cave by finding a suitable approximation that can be decomposed into a grammar.

The first rule is designed to create an initial starting point for the cave to be generated, which emulates the initial entry point for groundwater to begin creating caves within a section of the terrain. Gravitational forces and further groundwater erosion are emulated using the second and third rules, which lower the density of the lowest vertical point within a cavity in the terrain. The fourth rule is designed to widen existing cavities in the terrain in order to provide the resulting cave with more internal space, as well as offer some variation to the cavern's internal structure.

3.3.3 Surface Extraction

The polygonal mesh is extracted from the voxel data using the surface nets method (Gibson 1998), where the global energy minimisation strategy used for creating the surface is defined by the centre of mass of each voxel's edge intersections. This is more commonly referred to as the *naive surface nets* variant (Lysenko 2012).

This article utilises a version of the naive surface nets algorithm that executes entirely on the GPU. The algorithm is capable of high parallelism, therefore executing *compute shaders* on the voxel data can be greatly beneficial in terms of performance, as a large number of threads can be launched to work on individual segments of the data concurrently. The method we have

developed executes several compute shaders and pseudocode of each shader is shown in Algorithms 3.2-3.5.

The *ComputeCOM* shader writes to a linear array of 3D position vectors. This array represents the dual mesh of the input voxel grid. Each thread in the shader reads a section of voxel data in 2^3 groups. It then sets the initial position to be the centre of this group and interpolates the position using the direction and densities for each voxel, which is written to the final centre-of-mass buffer.

Construction of the vertex and index buffers is relatively straightforward. Each vertex in the vertex buffer corresponds to each individual element in the centre-of-mass buffer. When an element is added to the vertex buffer, its array index is written to a 3D array of integers that is used as a lookup table. After this process has completed, the construction of the index buffer takes place. The index of the thread in the *ConstructIndexBuffer* shader is used as an index into the lookup table. The thread subsequently queries the neighbours of this index to see if values exist in the table to create a triplet of indices. The triplet is then added to the index buffer.

The vertex normals of the mesh are computed as a separate shader after the GPU buffers have been constructed successfully. The *ComputeNormals* shader uses the index buffer and a cross product operation to calculate the normals for each vertex in the vertex buffer.

Algorithm 3.2 Pseudocode for computing the centres of mass of each voxel

```
function ComputeCOM(voxels):
    mass ⇐ 0
    massCentre ⇐ 0
    massCentres ⇐ {}
    for all v in voxels do
      for all n in v.neighbours do
            mass = mass + n.value
            massCentre = massCentre + (mass × n.position)
      end for
      massCentre = massCentre ÷ mass
      Append massCentre to massCentres
    end for
```

Algorithm 3.3 Pseudocode for computing the vertex buffer of the extracted surface.

```
function ConstructVBuffer(massCentres):
    vbuf ⇐ {}
    vertexLUT ⇐ {}
    for all m in massCentres do
      Append m to vbuffer
      Add index of last element in vbuf to vertexLUT
    end for
```

Algorithm 3.4 Pseudocode for computing the index buffer of the extracted surface.

```
function ConstructIBuffer(vertexLUT):
    ibuf ⇐ {}
    for all index in vertexLUT do
      if index is valid and neighbours of index are valid then
            Append index to ibuf
            Append valid neighbours to ibuf
      end if
    end for
```

Algorithm 3.5 Pseudocode for extracting the surface of the voxel data.

```
function SurfaceExtract(voxels):
    massCentres ⇐ ComputeCOM(voxels)
    vbuf, vertexLUT ⇐ ConstructVBuffer(massCentres)
    ibuf ⇐ ConstructIBuffer(vertexLUT)
```

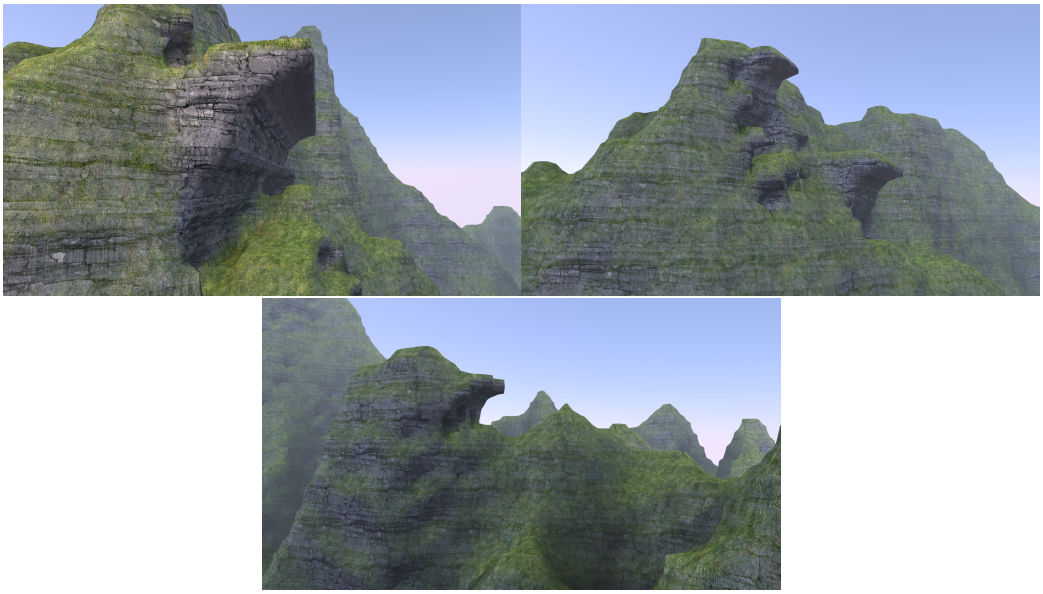

Figure 3.5: Examples of generated overhangs

3.4 Results

This section discusses the results obtained using the proposed method. Its effectiveness is demonstrated by the examples of caves and overhangs generated by the process. The results have been obtained by deriving voxel grids of several resolutions (32^3, 64^3, 96^3 and 128^3). The final surface meshes are rendered using a deferred renderer in *DirectX 11* (Microsoft n.d.) on a PC equipped with a 3.20 GHz quad-core *Intel®* CPU, 16 GB of RAM and an *NVIDIA® Titan X* GPU.

When rendering the mesh, triplanar texturing is used to effectively blend multiple textures (Geiss 2007). The weighting of each texture used at a point on the surface is determined by the dominant axis of the surface normal and the texture coordinate is calculated by the fractional part of the point's world space coordinate.

Examples of overhangs generated with the voxel grammar can be seen in Figure 3.5. The first example demonstrates an overhanging ledge protruding from the terrain with a plateau which appears naturally embedded within the terrain. In a heightmap-based approach a ledge such as this would either have to be stitched to the mesh to make a single mesh, or be rendered as a

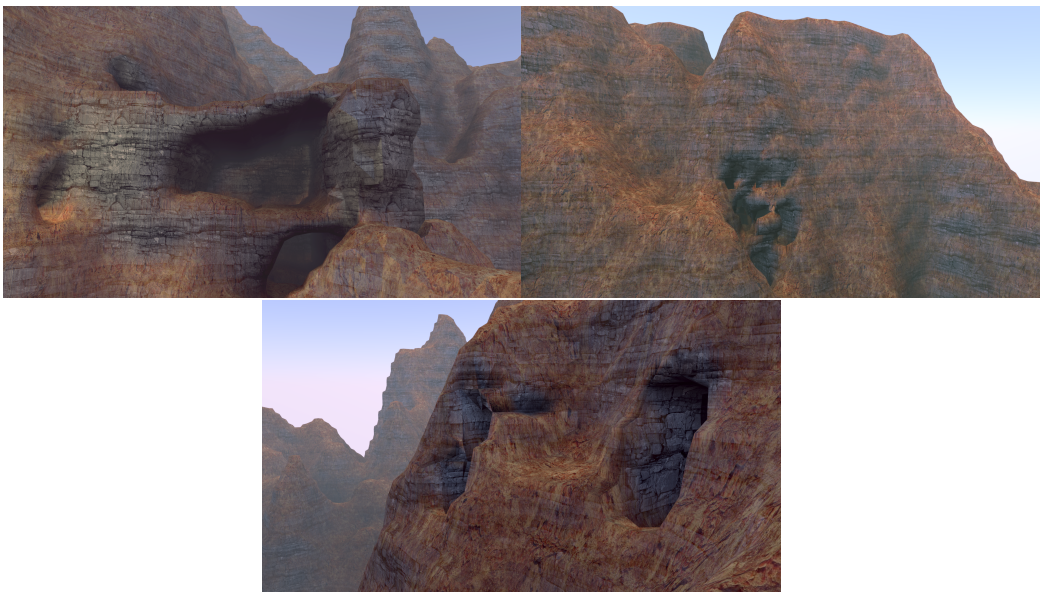

Figure 3.6: Examples of generated caves

separate object. As our method integrates the features directly into the mesh it avoids both of these options, so that the terrain can be treated as a unified entity. The second example shows multiple overhangs being created within a region of space in the terrain by increasing the range of the generation bounding box and offers a flexible option of being able to generate repeating volumetric features within a space. The third example presents another single overhang, this time without a plateau.

Caves generated with our method can be found in Figure 3.6. The first example shows a wide-mouthed cave constructed with our grammar, further demonstrating the benefits of editing volumetric data directly. Achieving this effect in a heightmap-based terrain would be a difficult task, as many surfaces would have to be edited to recreate the concavity presented in this image. The second example shows the grammar being applied to a smaller region of the terrain, where the surface on the side has been eroded to present a small network of caverns. The third image shows two medium-sized caves that have formed next to each other. This has been achieved by modifying the stride of the sliding window in the grammar generation parameters.

Table 3.5 shows the timings to generate cave and overhang examples in different voxel grid resolutions. When timing the data, the voxel grammar

Grid Resolution	Generation		Surface Extraction		Rendering	
	Overhang	Cave	Overhang	Cave	Overhang	Cave
32^3	3.477	9.204	0.125	0.117	0.778	0.777
64^3	32.008	89.059	0.271	0.281	0.915	0.919
96^3	111.331	314.384	0.578	0.604	1.126	1.129
128^3	269.121	770.208	1.072	1.130	1.434	1.441

Table 3.5: Times taken to generate voxels from rulesets, extract surfaces and render the meshes (in ms)

was set to derive the entirety of the voxel grid and the stride of the sliding window was set to 1x1x1 to ensure that the worst-case performance statistics were recorded.

The generation of voxels is the most expensive operation in the process and this is to be expected as it currently does not utilise the GPU. Instead, the voxel grid is derived on a single thread by the CPU and passed to the surface extraction functions. However, even at the largest resolution the grid was derived and voxels were generated in under a second. Generating voxels for overhangs was consistently faster than for caves (between 2.6 and 2.9 times faster). The cave grammar's last rule derives groups of 16 voxels at a time and this would account for the increase in processing time.

Our compute shader variant of the surface nets algorithm has been able to extract the surface of the volumetric data at satisfactorily fast rates. Meshes were created within 1ms and this makes it particularly suitable for dynamic editing of the voxel data, as the meshes can be regenerated at real-time rates. Mesh construction for overhangs and caves both perform at similar speeds as the algorithm is primarily bound by the voxel grid resolution and is independent of the type of features being created in the terrain.

There is little difference between the time taken to render the meshes of overhangs or caves and, as is expected, render times increase as the resolution of the grid becomes larger. This is due to the dual mesh grid required for the surface extraction increasing in resolution and, therefore, increasing the vertex count of the final mesh. However, this could be reduced by utilising various mesh simplification methods since the size of the polygons all over the mesh is uniform. By simplifying the mesh, the triangle count of the final mesh would be reduced and the GPU would have less work to execute. Furthermore, level-of-detail methods can be applied to the mesh that would

make polygons distant from the camera larger, so that the GPU does not spend time on rasterizing many small polygons far away from it that would produce no discernible visual difference. It should also be noted that the surface extraction creates the polygons in the index buffer in a non-deterministic manner. While the resulting mesh is correctly produced, it may benefit from some form of vertex cache optimisation, such as in Forsyth (2006), in order to try and ensure that the data is in a more GPU-friendly format.

3.5 Summary

This chapter of the thesis presented a method for generating complex terrain features using voxel grammars. Furthermore, it presented approaches on how such grammars can be created and what considerations should be taken into account when developing them.

The described methodology is effective in manipulating groups of voxels to create topologies of interesting terrain features using a rule-based approach. Comparing it to other forms of voxel terrain generation, this method provides a semi-automated way of constructing plausible terrain features.

Utilising 3D gradient noise (Perlin 2002) is a simple and fast method of generating volumetric terrains, however it is difficult to achieve a particular aesthetic. If using the fractional brownian motion variant of gradient noise, the controllable parameters are generally limited to the number of octaves, lacunarity, gain and input seeds. This offers the environment designer no further input on how the generated features will look. Furthermore, it is possible to generate floating islands of voxels that are not connected to the remainder of the terrain, which requires a further postprocessing step to fix. Our method offers a designer control over the resulting aesthetic of a terrain feature depending on the created input rules. Fixing floating voxels is also catered for by introducing further rules in the grammar, without the need for a separate postprocessing step.

An alternative method of constructing overhangs for terrains is presented by Gamito & Musgrave (2001) and warps an existing heightmap with a preconstructed vector field. Creation of the vector field is a complex operation and multiple vector fields must be constructed to enable different types of overhangs. Our method is not hindered by the same issues and can use the same ruleset on different input terrain sections to generate overhang variations.

However, the main limitation of this method is the substantially manual nature of designing new grammars. While there are several aforementioned factors that can be taken into account when creating rulesets, the approach requires some considerable effort and experimentation. Symbols and transforms in rules are concretely defined as a set of operators, but it would be beneficial to create rules in more abstract terms to be more intuitive to use. For example, this could be achieved by defining a symbol in terms of the slopes and material types found in a region of voxels, instead of being composed of a specific permutation of voxel densities. Similarly, defining transforms to create a cliff or a cave at a location, instead of operating on individual voxels, would be more useful for generating desired features more quickly. As such, we have reserved future research to create more abstract methods of defining rules in a grammar.

Furthermore, many terrain features such as caves can be formed in a recursive manner, by modifying transform operator values at each recursion. Currently, it is difficult to design grammars that perform such operations effectively. In order to enable this functionality to the voxel grammar approach, rules that execute other rules with parameter deltas would need to be added.

As this approach works on singular density values found within the terrain, this implies the terrain is composed of a single material. However, real terrains consist of multiple materials at multiple densities and ideally these varying densities should be taken into account by adding a check for the material type as a symbol operator.

The presented method is a single-threaded CPU implementation. As the voxel grids the grammar is applied to increase in resolution, this will detrimentally impact on performance. Therefore, it would be prudent to design a GPU-based method of executing the grammar on the voxel grid to maintain high performance in terms of time.

Chapter 4

Feature Generation for Volumetric Terrains

4.1 Introduction

Heightmap-based terrains do not allow for specific features of terrains such as cliffs, naturally-formed arches and caves. Volumetric representations of terrains are not limited in any such way and are gaining traction in practical applications such as procedurally generated computer games. This chapter presents procedural methods for generating features that are found in real world terrains and can be complex to represent using heightmap-based methods.

This work has been produced for integration into *PhyreEngine*^{TM} and proposes a procedural method for each of the three main features found in terrains by directly acting on the underlying volume representation.

This chapter is an extended version of a publication that was presented as a poster at SIGGRAPH 2017 (Dey et al. 2017) and extends prior work with voxel grammars where terrains are generated using a set of transforms (Dey et al. 2018). These transforms are a set of voxel replacements and the features described in this work here have been developed as more intuitive extensions.

Existing methods for creating overhangs have applied vector displacement to output vertices of heightmap-based terrains (Gamito & Musgrave 2001), though this work relies on unintuitive manipulation of the vector field data to achieve desired results. A hybrid approach introduced by Peytavie et al.

(2009) uses a signed-distance field to construct the terrain mesh. While implicit surfaces allow for many operations to be performed on the mesh, generating implicit functions can be computationally expensive. Recently, (Becher et al. 2017) presented the concept of voxelising input feature curves to form the terrain mesh, allowing for the formation of features such as overhangs and arches. However, our method generates a set of local delta values that can be applied directly to the volumetric data, which eliminates the need for a separate voxelisation step.

4.2 Feature Generation

Features such as overhangs, arches and caves are commonly found in naturally formed terrains. The ability to generate them easily for virtual terrains allows for greater fidelity terrains to be used for games and simulations. This section describes the method that we have used to generate overhangs, arches and caves, respectively. Overhangs and arches are functions that generate a set of local voxels that can be additively blended with the existing data to create the desired topologies. The cave generation method is a subtractive function utilising a particle-based approach that produces a set of negative density values removing voxels in the terrain (when blended).

The user begins with either an empty or initialised voxel dataset. They can then add either overhangs, arches or caves which are layered on top of the existing data to ensure the editing is non-destructive. The parameters for the features are set, an optional transformation matrix (translation and rotation) is applied to the feature and the resulting voxel values are submitted to the GPU for surface extraction and rendering. Each feature has user-defined dimensions in voxels (F_{dim}) and the index of each voxel (I) being processed is calculated as the unit cube coordinates of the current voxel (V_{index}) within the feature's bounding box ($I = \frac{V_{index}}{F_{dim}}$).

4.2.1 Overhang Generation

Cliffs and overhangs are formed primarily by erosion around coastal areas. In order to approximate their natural appearance, overhangs in our system are constructed by approximating their topology as a bicubic Bézier surface. A Bézier surface ($S(u, v)$) uses a 4x4 geometry matrix (G) of vectors containing the control points of the surface, and an example surface can be seen

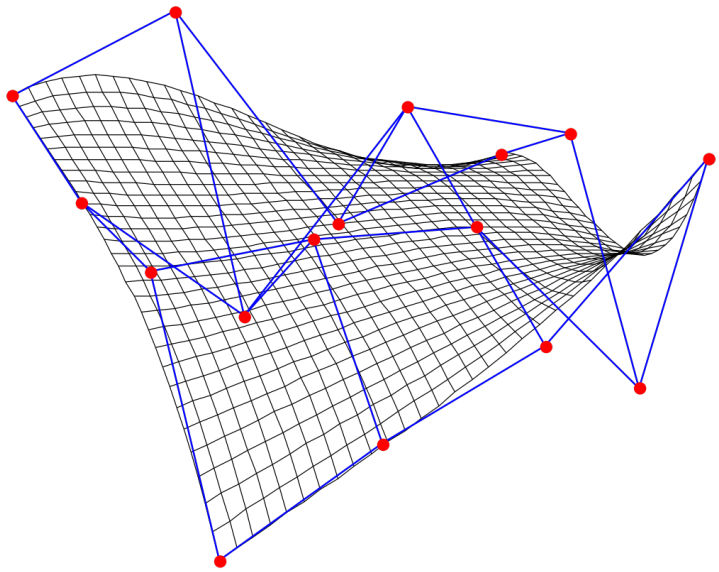

Figure 4.1: Example of Bézier surface with corresponding control points (in red).

in Figure 4.1. The second and third rows of control points act as erosion parameters to the resulting overhang. The final surface function is combined with a parabolic function ($P(u)$) using a user-defined exponent (k) to emulate realistic plateaus and is shown in Equation 4.1, where $u = I_z$ and $v = 1 - I_y$.

$$
\begin{aligned}
U &= \begin{bmatrix} 1 & u & u^2 & u^3 \end{bmatrix} \\
V &= \begin{bmatrix} 1 & v & v^2 & v^3 \end{bmatrix} \\
B &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{bmatrix} \\
S(u,v) &= U \cdot B \cdot G \cdot B^T \cdot V^T \\
P(u) &= (4u \cdot (1-u))^k \\
D_{Cliff}(u,v) &= \begin{cases} 1, & \text{if } I_x < min(S(u,v), P(u)) \\ 0, & \text{otherwise} \end{cases}
\end{aligned}
\tag{4.1}
$$

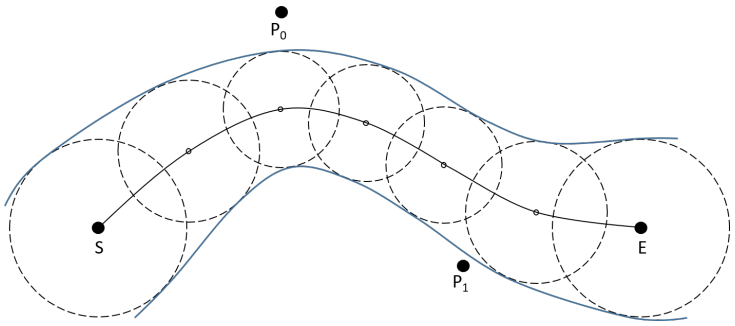

Figure 4.2: Diagram of sphere sweeping along Bézier curve

Along with parameters for the central control points on the surface, which we refer to as the *top* and *bottom* erosion parameters, as well as the parabolic parameters, the overhang generator also has a *lip height* and *base depth* parameter. The lip height refers to amount of the overhang at the lip that is not eroded. The base depth refers to where the bottom of the Bézier surface is positioned and results in dictating the overall slope of the overhang.

4.2.2 Arch Generation

Natural arches can be formed in two primary ways. Firstly, in karst terrain, dissolution between narrow bands of limestone can create breaks underneath top-level rock material. Secondly, arches can be formed by sections of cave ceilings collapsing. After these macroscopic changes have occurred to a terrain, wind and water erosion can introduce additional detail on the arch. Furthermore, the central section tends to be the weakest and thinnest part of the arch, whereas the two ends have thicker deposits of rock.

In order to approximate this in a virtual environment, arches are created by sweeping a sphere along a cubic Bézier spline, as shown in Figure 4.2, where the start point S, the end point E, and the two control points P_0 and P_1 govern the resultant shape. Each end of the arch has optional tapering parameters (radius multipliers (M_i) and exponents (E_i)), as well as interpolants (I_i) to define where the tapering occurs. This results in feasible natural arches simulating greater erosion towards the middle of the arch. At each step of the sweep (t), the radius (r) is updated using the tapering parameters (T_i). This uses the function in Equation 4.2, where V is the voxel

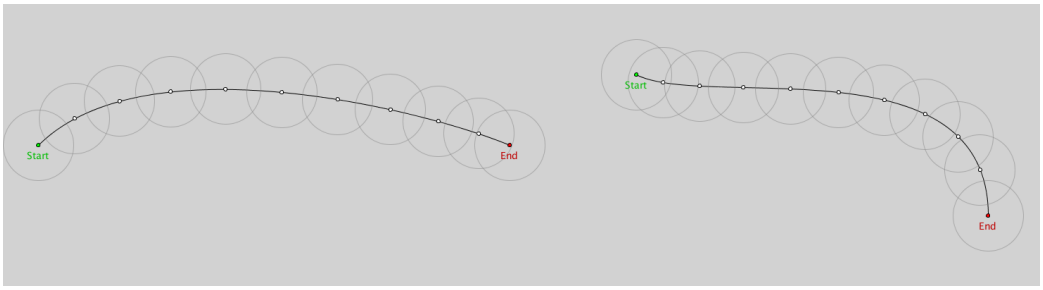

Figure 4.3: Illustration of arches constructed by a sweeping sphere with different start and end points.

being processed and P is the position of a voxel after the radius has been applied at the current step.

$$\begin{aligned} T_i &= r \cdot M_i \cdot (1 - I_i^{E_i}) \\ radius &= r + T_0 + T_1 \\ D_{Arch}(V, P) &= \begin{cases} 1, & \text{if } \|P - V\| < radius \\ 0, & \text{otherwise} \end{cases} \end{aligned} \tag{4.2}$$

From this, the final arch generator has a number of parameters that can be separated into two types: *global* parameters and *control* parameters.

Global parameters consist of both the start and end positions of the arch in the world coordinate space of the virtual environment. They also consist of the number of iterations to sweep along the spline as well as a radius to govern how thick the arch will be. An illustration of the global parameters with different start and end points can be seen in Figure 4.3. The control parameters consist of the interpolants of the control points and the tapering parameters.

4.2.3 Cave Generation

Similarly to arches, the natural formation of caves generally occurs in karst terrain where there is soluble rock to dissolve using a water source (Huggett 2016). The water sources associated with cave formation are usually large and continuous and cut through dense parts of soluble rock. This behaviour can

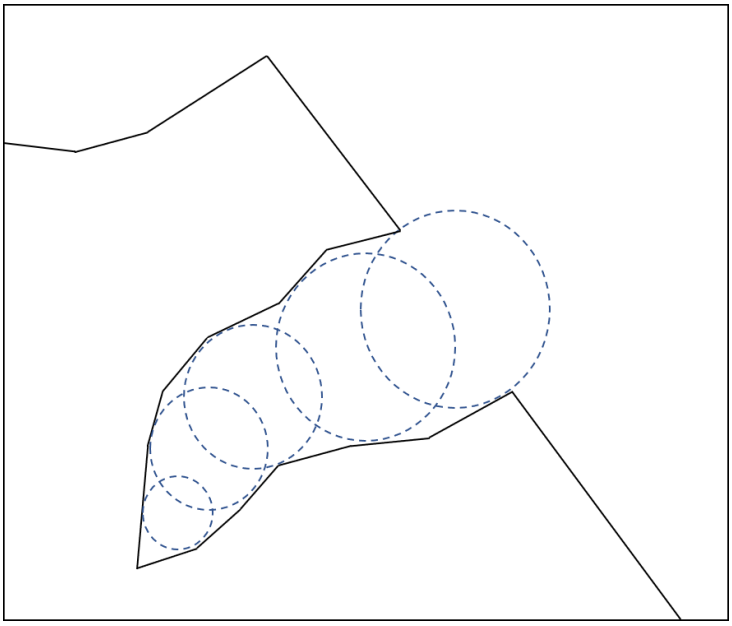

Figure 4.4: Diagram of sphere carving into terrain to make a cave.

be approximated for simulation purposes constructing a subtractive method of manipulating voxels. In order to generate cave-like structures, we adopt a particle-based approach where a sphere is traced in a user-specified direction for a set number of iterations, as shown in Figure 4.4. The particle emulates the behaviour of hydraulic erosion on rock surfaces to produce cave structures. In order to speed up the process, the particle starts at a user-defined value to determine the size of the cave's mouth. A decay parameter (λ) reduces the radius of the sphere for each iteration, thus representing a natural reduction in the amount of erosion occurring. Hydraulic erosion is also affected by gravitational forces and, therefore, a negative vector in the vertical axis is added to the direction (D) at each step (t). As we are modelling an approximation of hydraulic erosion to construct the cave topology, it is necessary for caves with a natural appearance to be affected by fluidic motion. An efficient method of simulating such movement is by generating a velocity vector (U) using curl noise (Bridson et al. 2007), which can then be added to the particle position for each iteration. Curl noise provides a satisfactory approximation of fluid movement and a scalar is used to determine

the amount of curl applied to the resulting cave structure. The combination of these methods efficiently computes a viable estimation of speleological erosion, reducing the need for a complex fluid dynamics system sculpting the internal structure of the cave. The position (P) and radius (r) of the particle are used in a sphere-box intersection test (Ericson 2004) to check whether a voxel is within the sphere and therefore subject to removal, using Equation 4.3.

$$\begin{aligned} P(t) &= P + t \cdot (D + G + U) \\ r(t) &= r - t\lambda \\ D_{Cave}(V, P, r) &= \begin{cases} 0, & \text{if } \|P - V\| < r \\ 1, & \text{otherwise} \end{cases} \end{aligned} \tag{4.3}$$

4.2.4 Expressive Range

An important aspect of procedurally generated content is to enumerate the number of assets that can be generated with the variety of parameters of a generator function. On a system that can generate very large numbers of assets, this can be achieved by evaluating the expressivity of each method parameter, called the *expressive range* (Togelius et al. 2016). As each generator function has a number of real-valued parameters, representation of this multi-dimensional data has been reduced into a single scalar value. This value has been retrieved by using a rotation invariant volumetric shape similarity heuristic, as described in Snape et al. (2016). The dimensionality reduction offered by this method enables us to effectively visualise the variations of the voxel grids used to represent terrain features.

$$\begin{aligned} g(i) &= \text{gradient at voxel } i \\ \phi(i) &= \arccos(\frac{\hat{g}_y(i)}{\hat{g}_x(i)}) \\ \theta(i) &= \arctan(g_z(i)) \\ s &= \sum_i \cos(\phi(i)) + \sum_i \cos(\theta(i)) \end{aligned} \tag{4.4}$$

For each continuous parameter $x \in \mathbb{R}$ in each feature's generation algo-

rithm, we discretise the input value by sampling it at sensible intervals to generate a suitable set of results to demonstrate the expressive range of each algorithm. The range of each parameter for each generator, as well as their sampling intervals, can be seen in Table 4.1, 4.2 and 4.3.

Parameter	**Type**	**Range**	**Interval**
Lip Height	Real	(0, 1)	0.01
Base Depth	Real	(0, 1)	0.01
Top Erosion	Real	(0, 1)	0.01
Bottom Erosion	Real	(0, 1)	0.01
Parabola Offset	Real	(0, 1)	0.01
Parabola Exponent	Real	(1, 4)	0.02

Table 4.1: List of parameter ranges for the overhang generator

Parameter	**Type**	**Range**	**Interval**
Start	Vector	[16, 16, 16]	-
End	Vector	[48, 48, 16]	-
Iterations	Integer	{10, 20, 30, 40, 50}	-
Radius	Integer	(1, 10)	1
Start Offset	Real	1.0	-
Start Peak	Real	(0, 10)	5
Start Taper Interpolant	Real	1.0	-
Start Taper Radius Multiplier	Real	(0, 1)	0.01
Start Taper Exponent	Real	(1, 4)	0.02
End Offset	Real	1.0	-
End Peak	Real	(0, 10)	5
End Taper Interpolant	Real	1.0	-
End Taper Radius Multiplier	Real	(0, 1)	0.01
End Taper Exponent	Real	(1, 4)	0.02

Table 4.2: List of parameter ranges for the arch generator

Parameter	**Type**	**Range**	**Interval**
Start	Vector	[16, 16, 4]	-
Direction	Vector	[0, 0, 1]	-
Iterations	Integer	{10, 20, 30, 40, 50}	-
Mouth Size	Real	(1, 10)	0.2
Curl Scale	Real	(0, 1)	0.01
Decay	Real	(0, 1)	0.01
Gravity	Real	(0, 1)	0.01
Gravity Exponent	Real	(1, 4)	0.02

Table 4.3: List of parameter ranges for the cave generator

4.2.5 Parallelisation

In order to generate results for large features, a high number of voxel positions need to be evaluated by the various generator functions. Performing this on the CPU can take an inordinate amount of time. Since each voxel position is evaluated independently of other voxel positions, this becomes a highly parallelisable task. Thus, the generator functions have been written to use *NVIDIA's* CUDA platform (Kirk et al. 2007) in order to utilise the GPU's parallel capabilities. A CUDA kernel submits work to the GPU via *threadgroups* in blocks of 32 threads for each dispatch (known as a *warp*). A warp executes in lockstep, so minimising branching is necessary to ensure that each warp dispatch operates efficiently, as any thread in a warp that diverges to a branch forces the other threads to wait until the branch has completed executing (Pranckevičius 2018).

Generation of cliffs occurs on a voxel grid where each voxel is individually checked against the parametric Bézier surface and whether it lies within the interior or the cliff or not. A single CUDA kernel with a three dimensional block size is used to quickly iterate through all of the individual voxels.

However, generation of caves and arches differ in their CUDA implementation. These generators utilise the sweeping of a sphere to populate the feature's voxels, resulting in a voxel grid size that is difficult to determine. As such, the CUDA kernels for these generators take a multiple pass approach, meaning that the kernels are dispatched for a number of user-defined iterations. As the sphere is swept along the path of the arch or cave, the voxels that intersect the sphere are sent to the kernel to be filled.

4.3 Results

The results obtained from the different procedural generation algorithms are detailed in this section. Firstly, examples of features are shown with differing parameter sets. Then, the expressive ranges of the generator functions are evaluated in order to show that they can create a diverse set of features from just a few parameters. Finally, the performance is evaluated by comparing the time taken for the CPU and GPU versions of each generator.

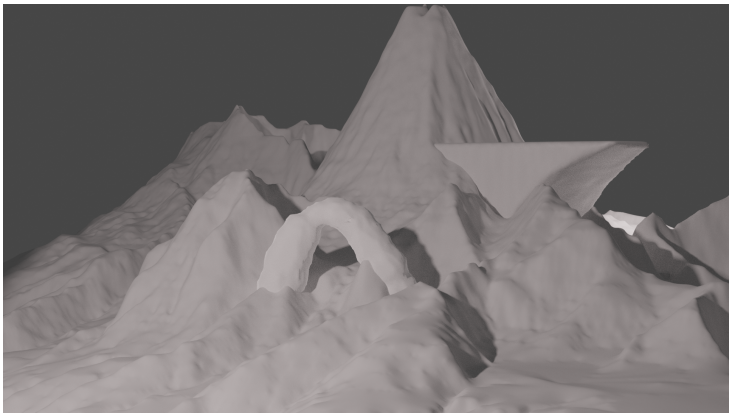

Figure 4.5: Pathtraced render of terrain with an embedded arch and overhang.

4.3.1 Terrain Feature Examples

The examples of terrain features created by the three generator functions visually demonstrate the wide variety of structures that can be constructed, as shown in Figure 4.5. The following sections describe in more detail the effects of modifying the most significant parameters associated with each generator. Furthermore, the use cases that each setting could be utilised for in virtual simulations are explained.

Figure 4.6 shows examples of overhangs, arches and caves at differing voxel grid resolutions. As expected with lower voxel grid resolutions, the fidelity of the resulting mesh is lower than higher voxel grid counterparts, due to the lower number of polygons. However, this could be smoothed using subdivision algorithms that are applied as a postprocessing step on the resulting polygon mesh.

For each result, parameters that were not modified remained at the default value that can be found in Tables 4.4, 4.5 and 4.6 for overhangs, arches and caves, respectively.

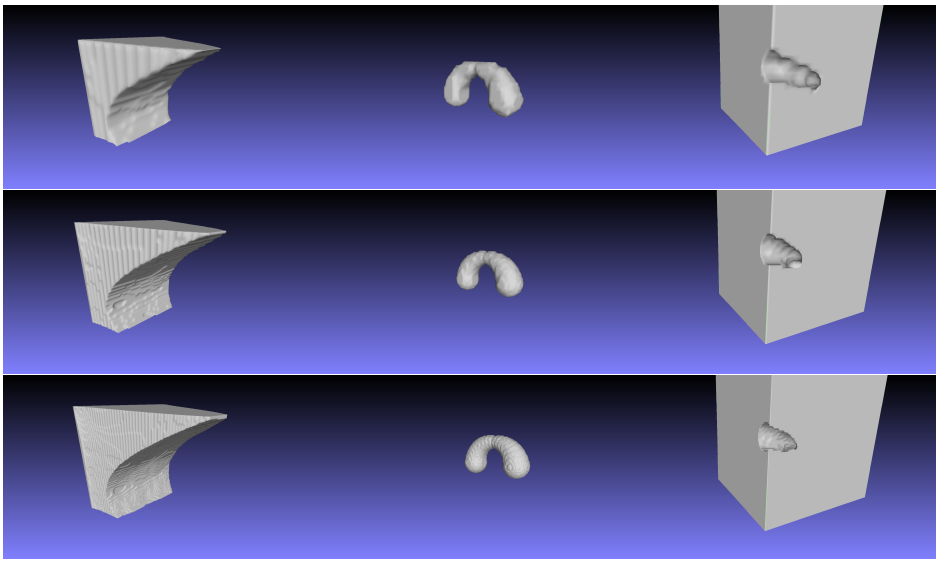

Figure 4.6: Examples of generated overhangs (left), arches (middle) and caves (right) at varying resolutions (Top: 32^3, Center: 64^3, Bottom: 128^3)

Parameter	**Value**
Lip Height	0.0
Base Depth	0.0
Top Erosion	0.0
Bottom Erosion	0.0
Parabola Offset	0.0
Parabola Exponent	1.0

Table 4.4: List of default values of non-varying overhang parameters when generating results for different variants

Parameter	Value
Iterations	50
Radius	5.0
Start Offset	0.5
Start Peak	10.0
Start Taper Interpolant	0.0
Start Taper Radius Multiplier	1.0
Start Taper Exponent	1.0
End Offset	0.5
End Peak	10.0
End Taper Interpolant	0.0
End Taper Radius Multiplier	1.0
End Taper Exponent	1.0

Table 4.5: List of default values of non-varying arch parameters when generating results for different variants

Parameter	Value
Iterations	20
Mouth Size	5.0
Curl Scale	0.0
Decay	0.0
Gravity	1.0
Gravity Exponent	1.0

Table 4.6: List of default values of non-varying cave parameters when generating results for different variants

Overhang Examples

The lip height and the base depth parameters (with examples demonstrated in Figure 4.7 and Figure 4.8) for the overhang generator are both bounded values in the interval [0, 1] that represent a proportional value for the size of the lip and the concavity of the base in the resulting overhang model, respectively. This makes it relatively straightforward for users to create varied overhang types without having to rely on prior knowledge regarding the voxel grid dimensions of the feature.

Examples of the two erosion parameters corresponding to the inner control points on the Bézier surface of the overhang generator can be seen in Figures 4.9 and 4.11. A closer view of the effect of the top erosion parameter can be seen in 4.10. Both parameters are bounded within the interval [0, 1] and are used to approximate eroding behaviours present in real world overhang formation.

The parabola components help represent the overall shape of the overhang surface. The exponent determines how sharp the surface of the overhang is, as seen in examples in Figure 4.13 and a closer top-down view provided in Figure 4.14, whereas the offset provides a way to adjust the parabolic curve for a further aesthetic change (Figure 4.12). The combination of the two can be used to generate overhangs that have top surfaces ranging from a small, rounded shape to a continuous surface that can be used to stitch a group of overhangs together at a similar height.

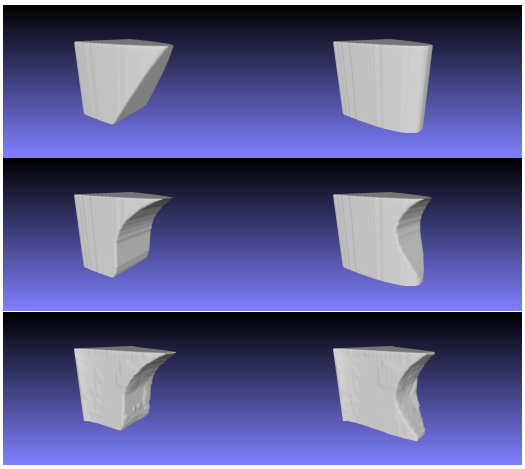

Figure 4.7: Examples of generated overhangs with varying base depth parameters (Left: 0.5, Right: 1.0)

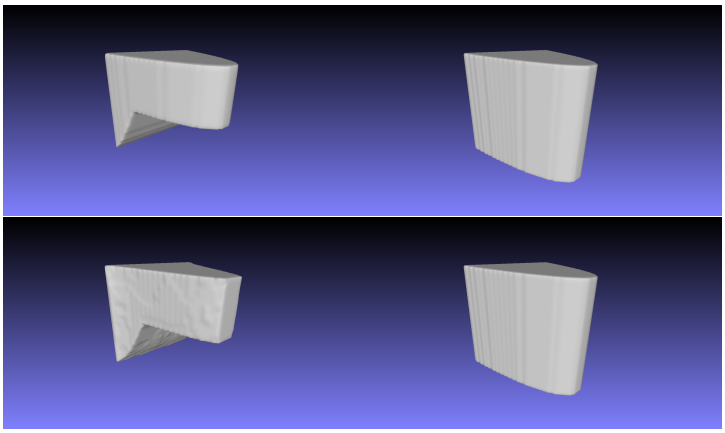

Figure 4.8: Examples of generated overhangs with varying lip height parameters (Left: 0.5, Right: 1.0)

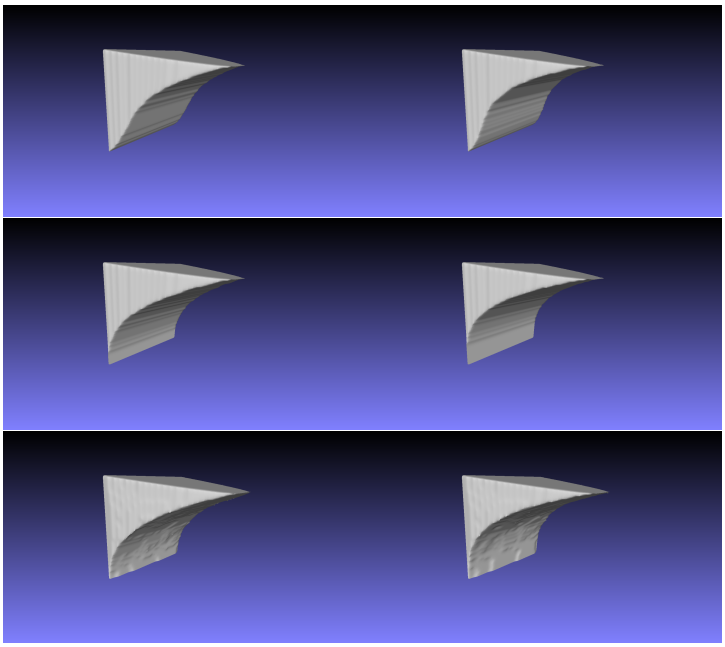

Figure 4.9: Examples of generated overhangs with varying top erosion parameters (Left: 0.5, Right: 1.0)

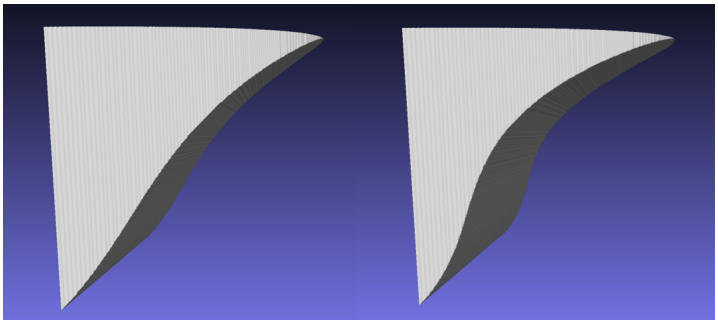

Figure 4.10: Close-up of generated overhangs with varying top erosion parameters (Left: 0.5, Right: 1.0)

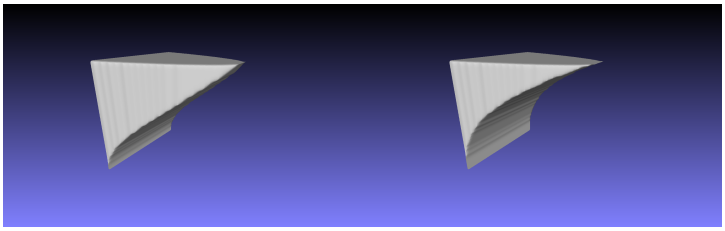

Figure 4.11: Examples of generated overhangs with varying bottom erosion parameters (Left: 0.5, Right: 1.0)

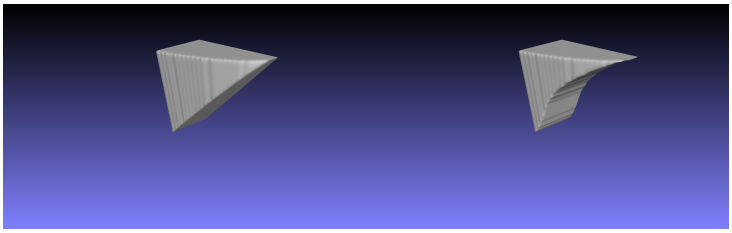

Figure 4.12: Examples of generated overhangs with the parabola offset parameter set to 0.5

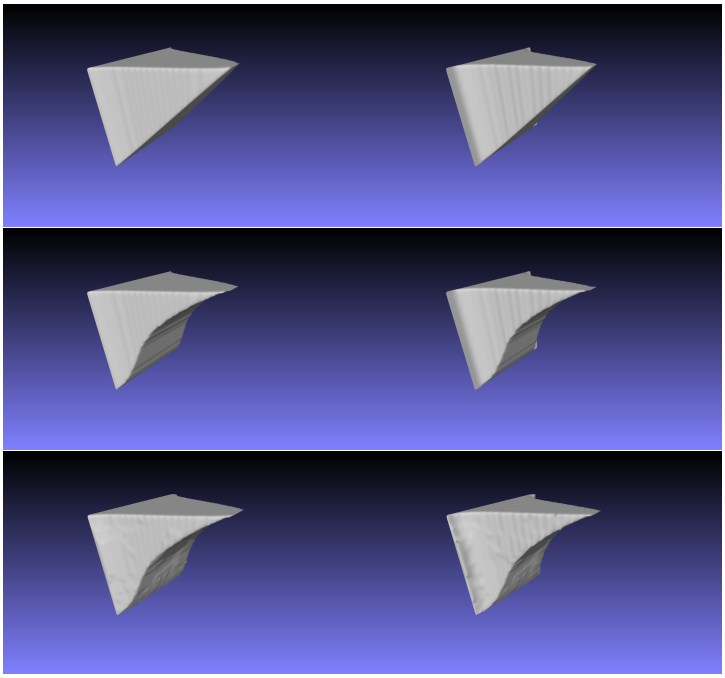

Figure 4.13: Examples of generated overhangs with varying parabola exponent parameters (Left: 2.0, Right: 4.0)

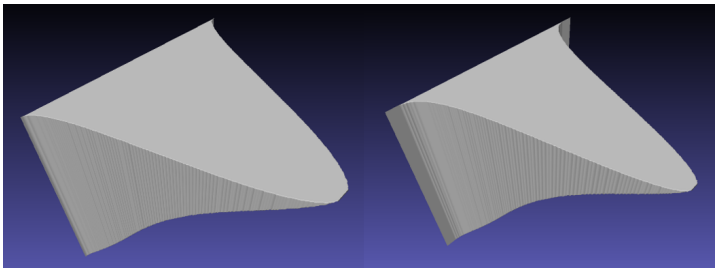

Figure 4.14: Close-up of generated overhangs with varying parabola exponent parameters (Left: 2.0, Right: 4.0)

Arch Examples

The start peak and end peak parameters in an arch generator determine the height levels at two points along the Bézier curve (Figure 4.16). These values can be unbound as long as they are greater than or equal to 0. For our results, we have bound the values between 0 and 10, dictating the maximum height that each peak can be at.

Tapering allows the arch to be presented with a natural falloff of material that simulates the appearance of hydraulic and wind-based erosion, as shown in Figures 4.18, 4.20 and 4.19. Modification of the radius multiplier parameters, bound within the interval [0, 1], enables the designer to vary the resulting aesthetic from a subtly weathered look with a high value, to a stronger eroded look using a value closer to 0, as shown in Figure 4.20. Furthermore, the taper exponent parameters offer further control on the resulting radii along the curve of the arch by increasing or decreasing the strength of the erosion effect, as seen in Figure 4.19.

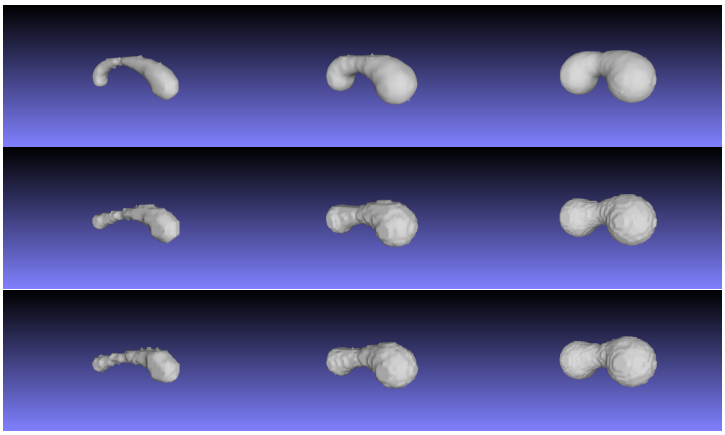

Figure 4.15: Examples of generated arches with varying radius parameters (Left: 2, Center: 4, Right: 6)

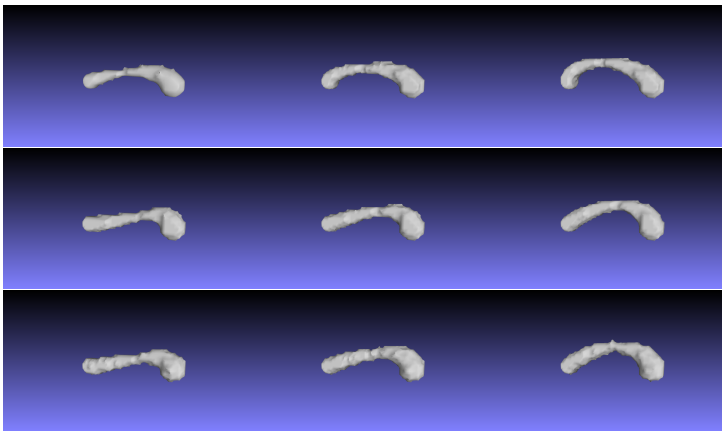

Figure 4.16: Examples of generated arches with varying peak parameters (Left: 0, Center: 5, Right: 10)

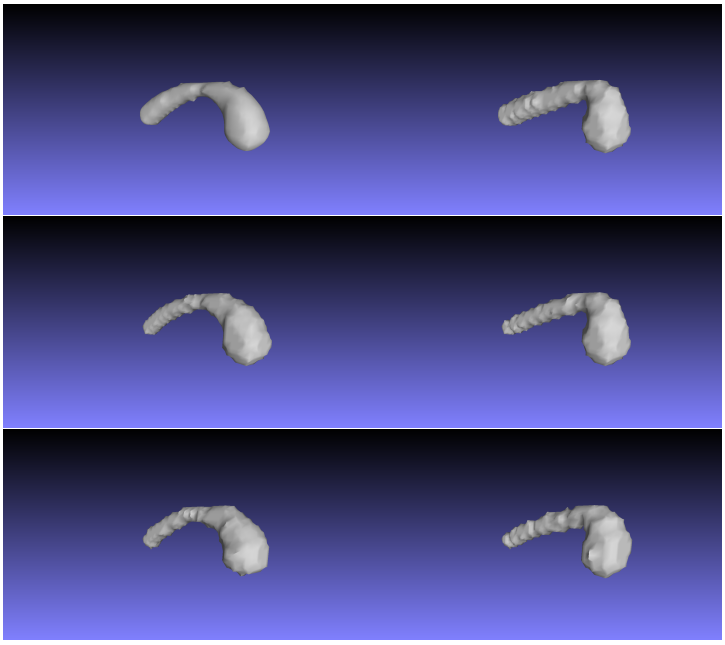

Figure 4.17: Examples of generated arches with varying offset parameters (Left: 0.5, Right: 1.0)

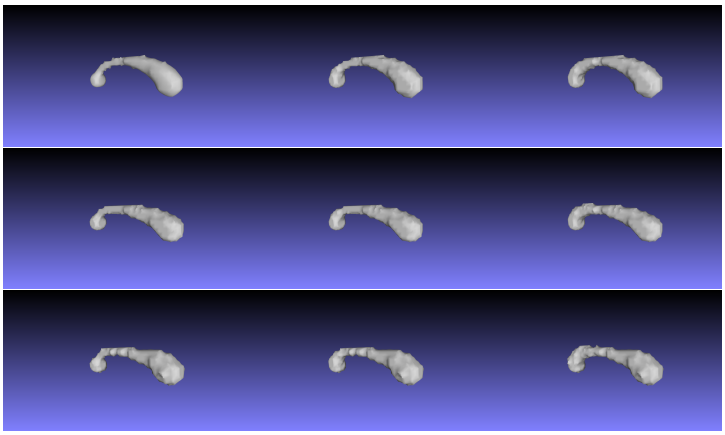

Figure 4.18: Examples of generated arches with varying taper interpolant parameters (Left: 0.0, Center: 0.5, Right: 1.0)

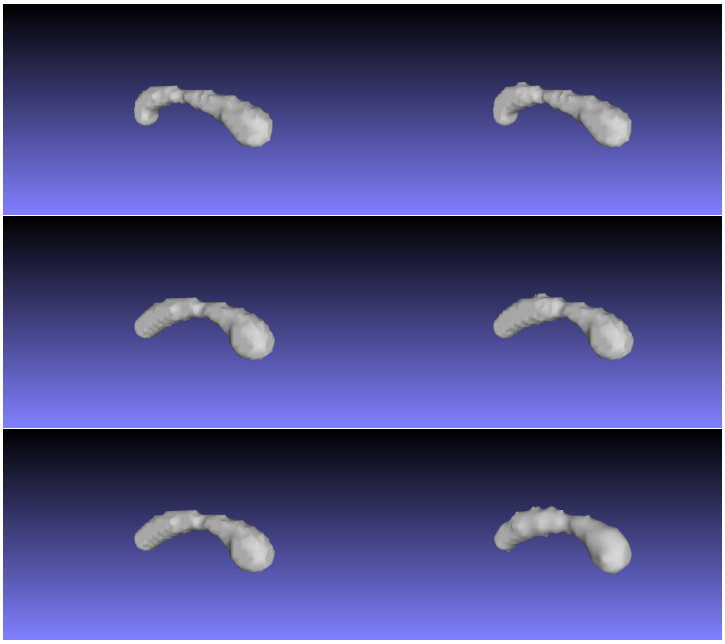

Figure 4.19: Examples of generated arches with varying taper exponent parameters (Left: 2.0, Right: 4.0)

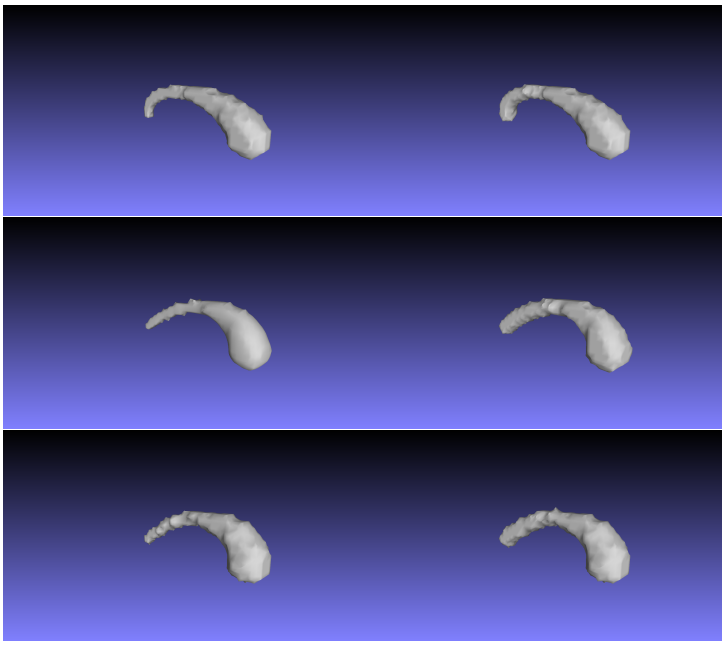

Figure 4.20: Examples of generated arches with varying taper radius multiplier parameters (Left: 0.0, Right: 0.5)

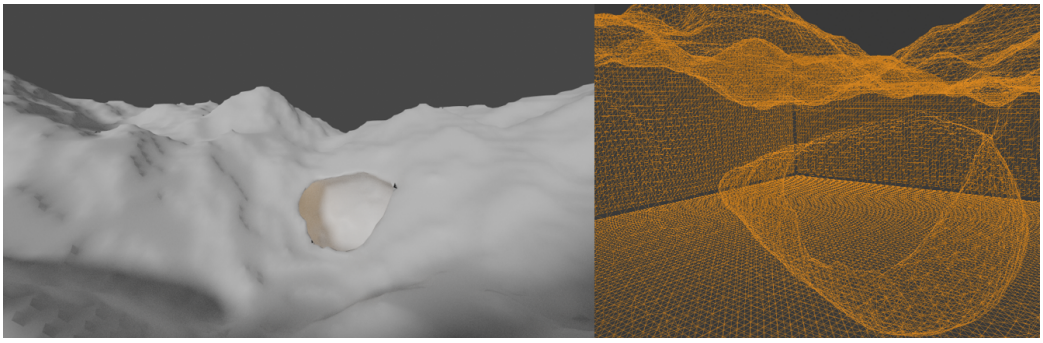

Figure 4.21: Pathtraced render of terrain with an embedded cave and its corresponding wireframe.

Cave Examples

Cave parameters determine the size and path of the sweeping sphere used to carve out the feature from a set of solid voxels, an example of which can be seen in Figure 4.21. The mouth size determines the initial radius of the sphere and varying sizes can be seen in Figure 4.22. Furthermore, the rate at which the sphere's radius decreases is determined by the decay parameter (Figure 4.23). This partly simulates fluidic erosion as the fluid is eventually absorbed by the cave's material. In order to further simulate fluid flow motion when generating the cave, the generator utilises curl noise. The curl scale parameter governs how much effect it has on the resulting path of the sphere. As shown in Figure 4.24, a higher curl scale value results in a more winding path that the sphere sweeps across, whereas a smaller curl scale value results in more subtle movement.

The motion of the sphere can also be affected by a gravitational force (a predetermined vector in the global y-direction). The strength of the gravitational force can be manipulated via the gravity parameter (Figure 4.25) and the rate at which the gravity is applied to each position of the sphere can be controlled by the gravity exponent parameter (Figure 4.26).

These few parameters offer a great deal of control for the designer and can be used to construct many variations of the cave structure in a speedy manner.

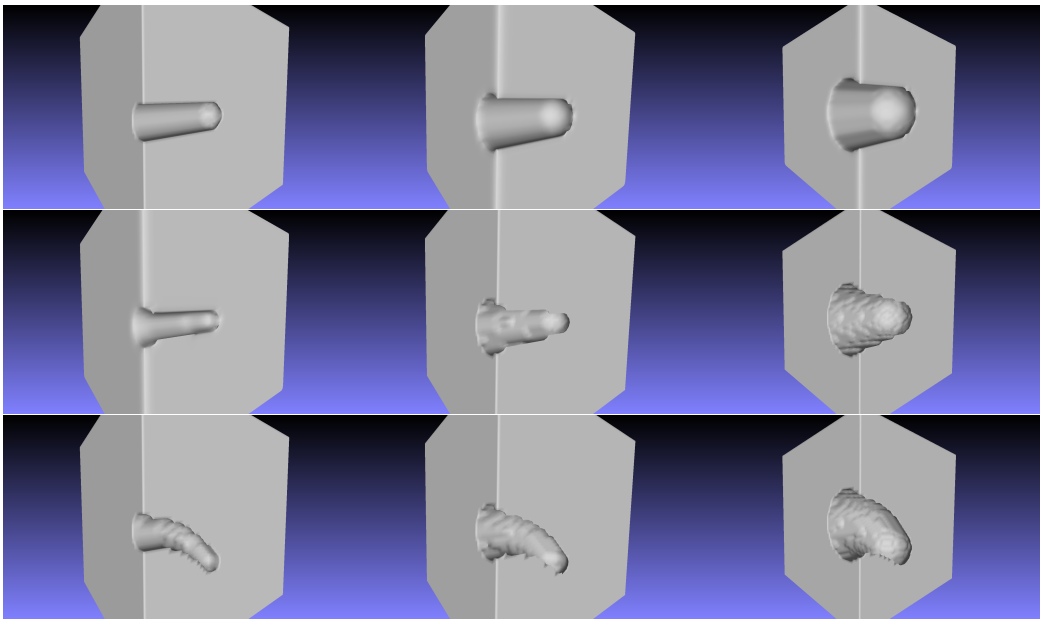

Figure 4.22: Examples of generated cave cross-sections with varying mouth size parameters (Left: 3, Center: 5, Right: 10)

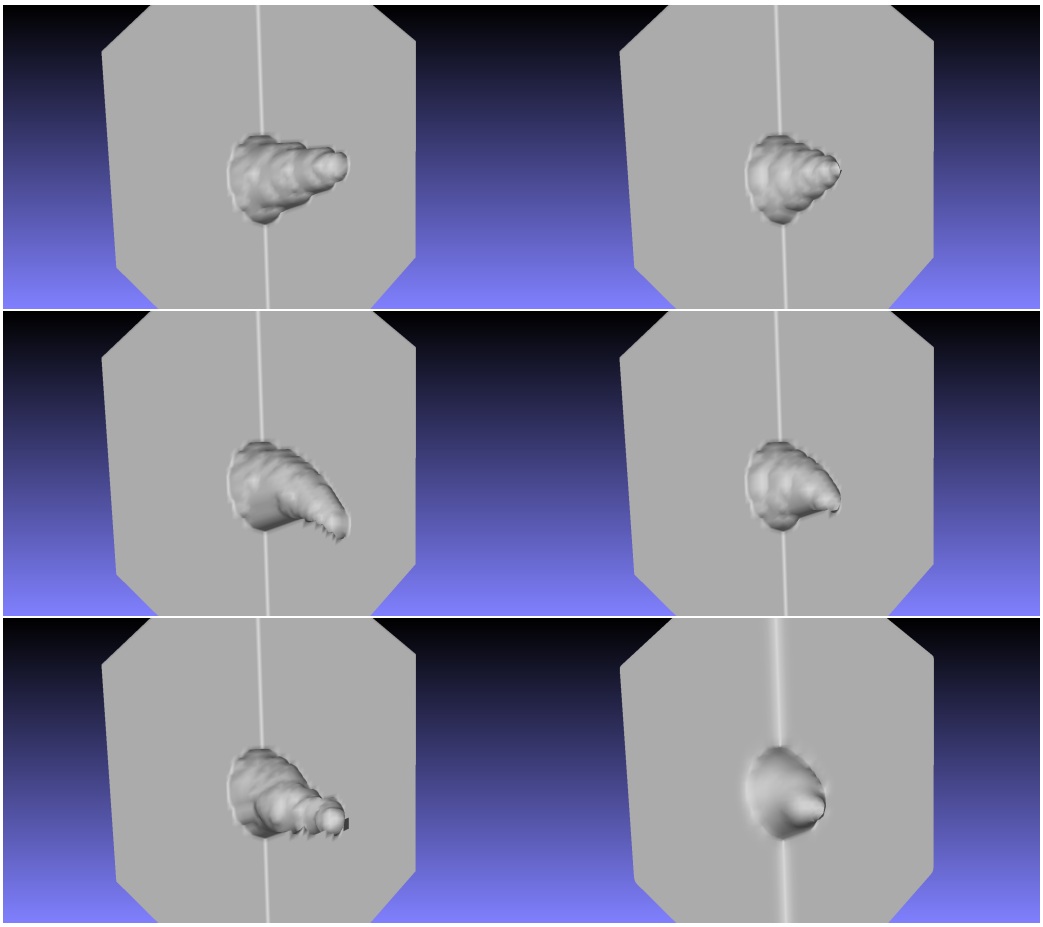

Figure 4.23: Examples of generated cave cross-sections with varying decay parameters (Left: 0.5, Right: 1.0)

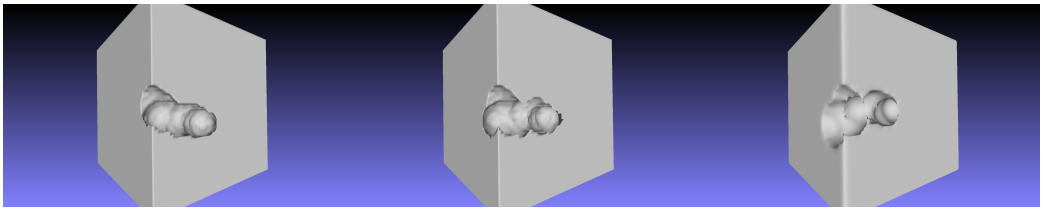

Figure 4.24: Examples of generated cave cross-sections with varying curl scale parameters (Left: 0.2, Center: 0.3, Right: 0.4)

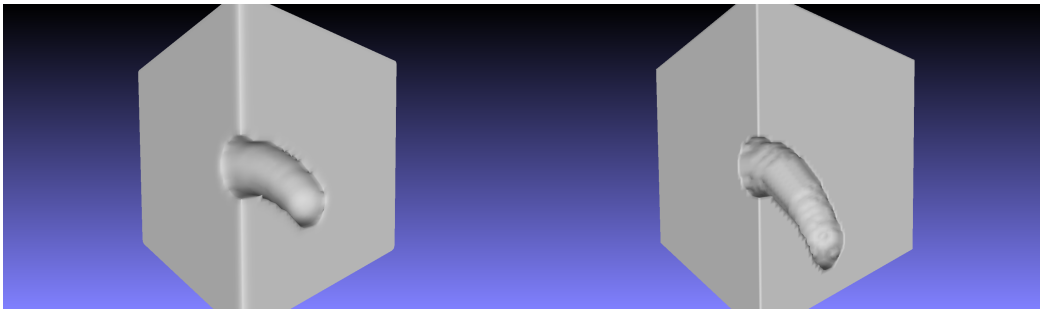

Figure 4.25: Examples of generated cave cross-sections with varying gravity parameters (Left: 0.5, Right: 1.0)

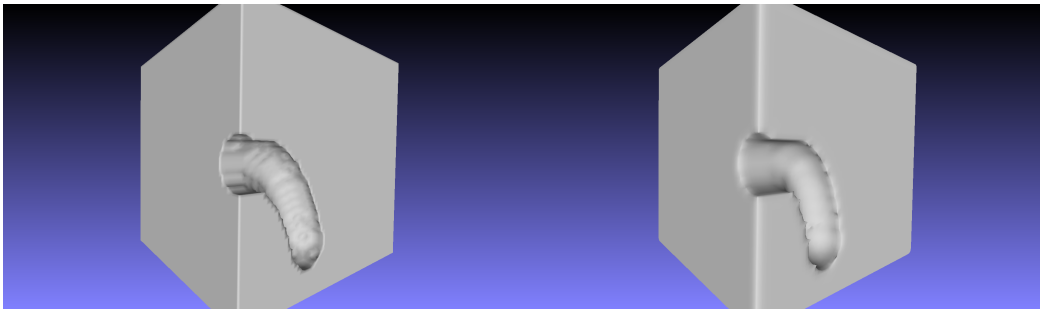

Figure 4.26: Examples of generated cave cross-sections with varying gravity exponent parameters (Left: 2.0, Right: 4.0)

4.3.2 Expressive Range

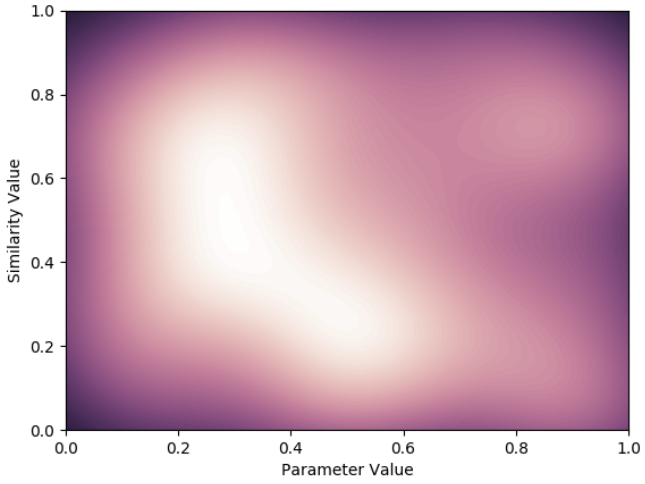

Figure 4.27: Example KDE plot of a uniform distribution of similarity values

In order to quantify the variety of structures that can be generated with our methods, the expressive range of each generator function is analysed in this section. The similarity metric provided by Snape et al. (2016) was used to reduce the dimensionality of each voxel dataset into a single value. For each parameter experiment, the similarity value was normalised and its frequency plotted in a two-dimensional kernel density estimation (KDE) plot. The first axis represents the changing value of the generator parameter being inspected and the second axis shows the similarity value computed for the generated feature. This provided a good insight into the overall effect of changing a parameter to create a resulting structure. An example of a good KDE plot is shown in Figure 4.27, which had a set of 1000 data points randomly generated with a uniform distribution. This example shows that as the parameter value changes on the first axis, the range of values on the second axis maintain an even distribution. The closer the parameter plots look to this example plot, the more expressive the parameter is in generating large variations of features. For each parameter, 100 features were generated at each progressive parameter value at voxel grid resolutions of 32^3, 64^3 and 128^3 voxels.

Overhang Parameter Expressivity

The KDE plots for overhang parameters can be seen in 4.28. The base depth parameter of a generated overhang feature dictates how large the base of the overhang will be. As the parameter value approaches 1, the range of values becomes wider. This indicates that there are more feature variations as the parameter increases in value, which can be justified by the introduction of further voxels as the base depth increases. More voxels being generated implies that more voxels can be modified as part of the generation process.

The lip height parameter determines how large the upper lip of a cliff will be and presents a similar plot to the base depth parameter. This parameter also generates a number of voxels increasing the number of varied ways that the voxels can be modelled when a new feature is created.

The two erosion parameters, top and bottom, control the inner control points on the abstract Bézier surface. Both parameters show an evenly wide distribution of similarity scores for the generated overhangs, demonstrating that both of these parameters are the most expressive and create the most frequent number of varied overhangs.

Parameters for manipulating the parabolic nature of the top of the overhang are governed by the offset and exponent. The parabola offset demonstrates a smaller range of values as the parameter increases. Since the offset only applies to the shape of the surface and not the underlying overhang topology, the parameter affects a smaller number of voxels than other parameters, thus justifying the narrower range of similarity values. Conversely, the exponent parameter has more impact on the shape of the generated overhang and edits a larger number of voxels. Therefore, the exponent parameter has more expressivity due to the number of variations of the overhang that can be produced.

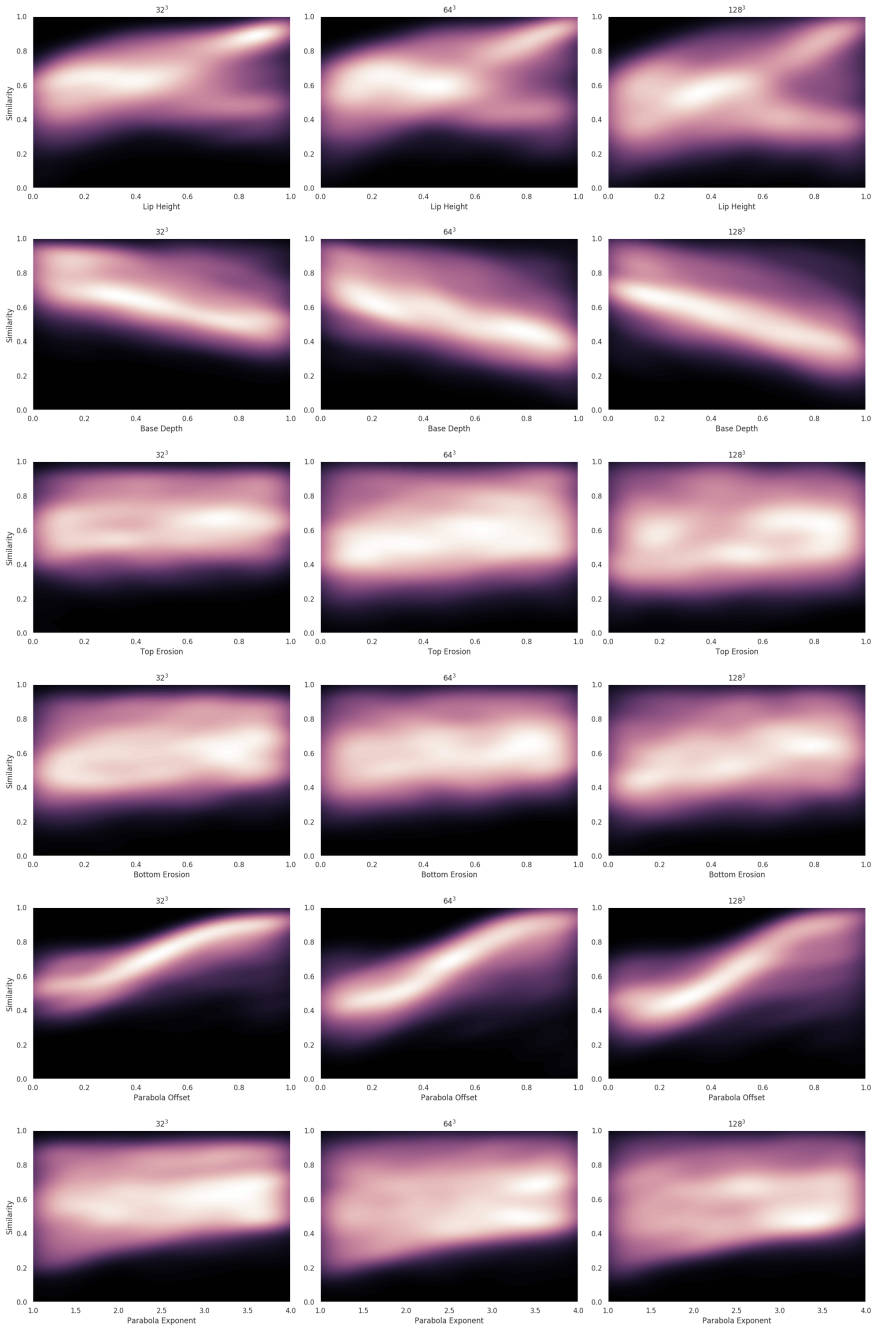

Figure 4.28: Expressive ranges of generated overhangs

Arch Parameter Expressivity

The KDE plots for the arch generator parameters can be seen in Figure 4.29.

The radius parameter for the generator function dictates the size of the sweeping sphere across the Bézier spline used to construct the arch. As the parameter plot shows, the range of similarity values becomes more varied as the value of the parameter becomes greater. This is reasonable behaviour since the number of voxels being manipulated increases as the size of the radius gets larger, and therefore more varied features can be generated.

The peak parameter exhibits a very expressive trend. The range of similarity values across the minimum and maximum parameter values stays relatively uniform throughout the plot, and indicates a high degree of expressivity. Similarly, the taper exponent parameter shows the same trend with a uniformly distributed range of similarity values as the parameter increases and is therefore also highly expressive.

The taper radius multiplier shows an interesting trend, where the range of similarity values increases as the parameter increases in value. This can be explained by the increase in the number of voxels being manipulated as the radius multiplier gets larger.

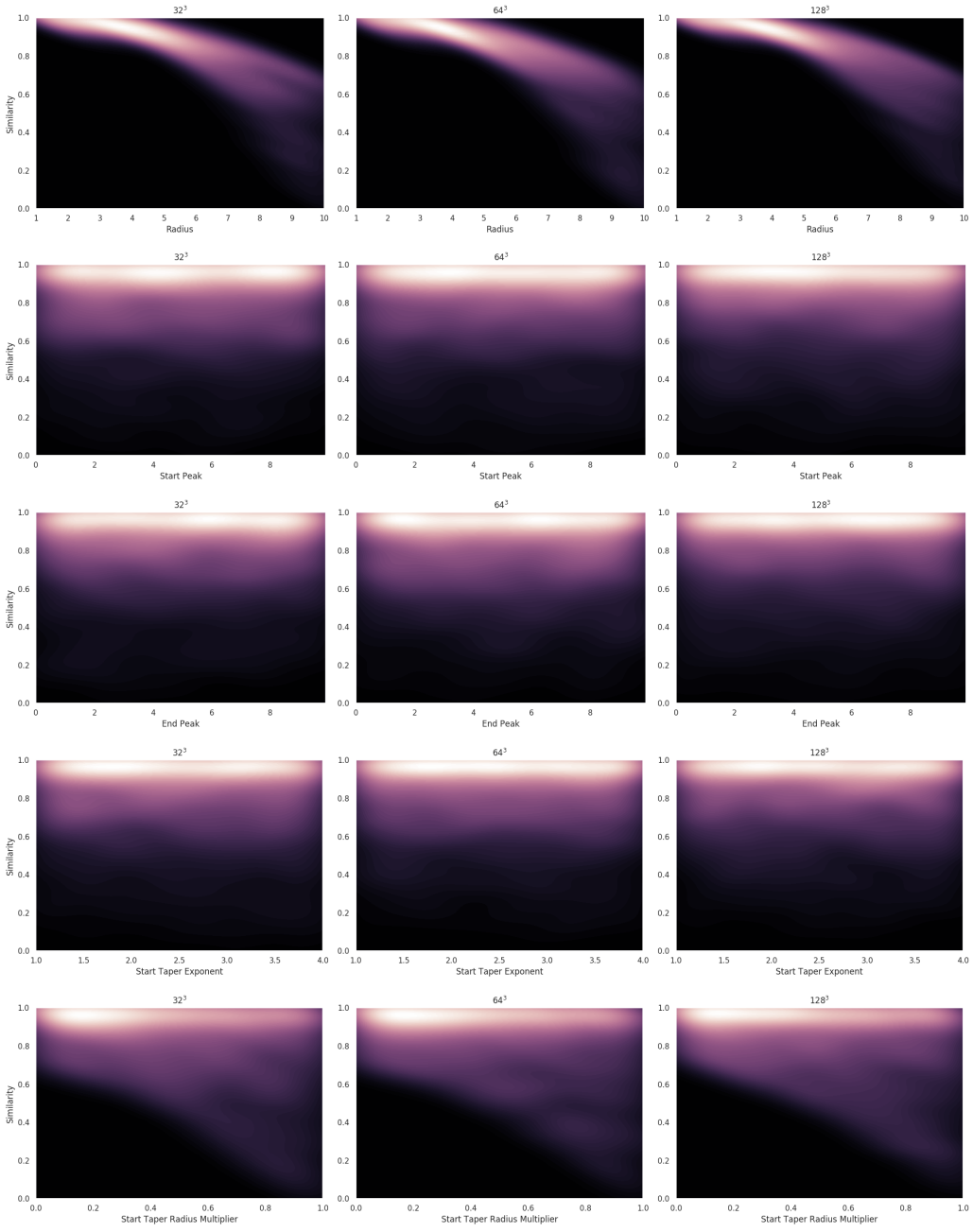

Figure 4.29: Expressive ranges of generated arches

Cave Parameter Expressivity

The KDE plots for the arch generator parameters can be seen in Figure 4.30.

The mouth size parameter for the cave generator determines the size of the sweeping subtractive sphere used to interact with a group of filled voxels. As this behaviour is similar to the radius parameter of the arch generator, a similar distribution plot can be expected. The plot shows that as the parameter value increases, the range of similarity values becomes larger, as hypothesised. This is consistent with the radius parameter for arches since more voxels are manipulated as the radius of the sphere increases.

Conversely, when generating features for the decay parameter, as the value of the parameter increases, the range of similarities decreases. This is due to the parameter's behaviour of reducing the number of voxels that are changed by the generator as the subtractive sphere decays in radius.

Finally, the gravity, gravity exponent and curl scale parameters exhibit very similar distributions. The values stay within the same range as all of the parameter values increase. Given that this range maintains its width, it is implied that these parameters are highly expressive in being able to generate varied caves.

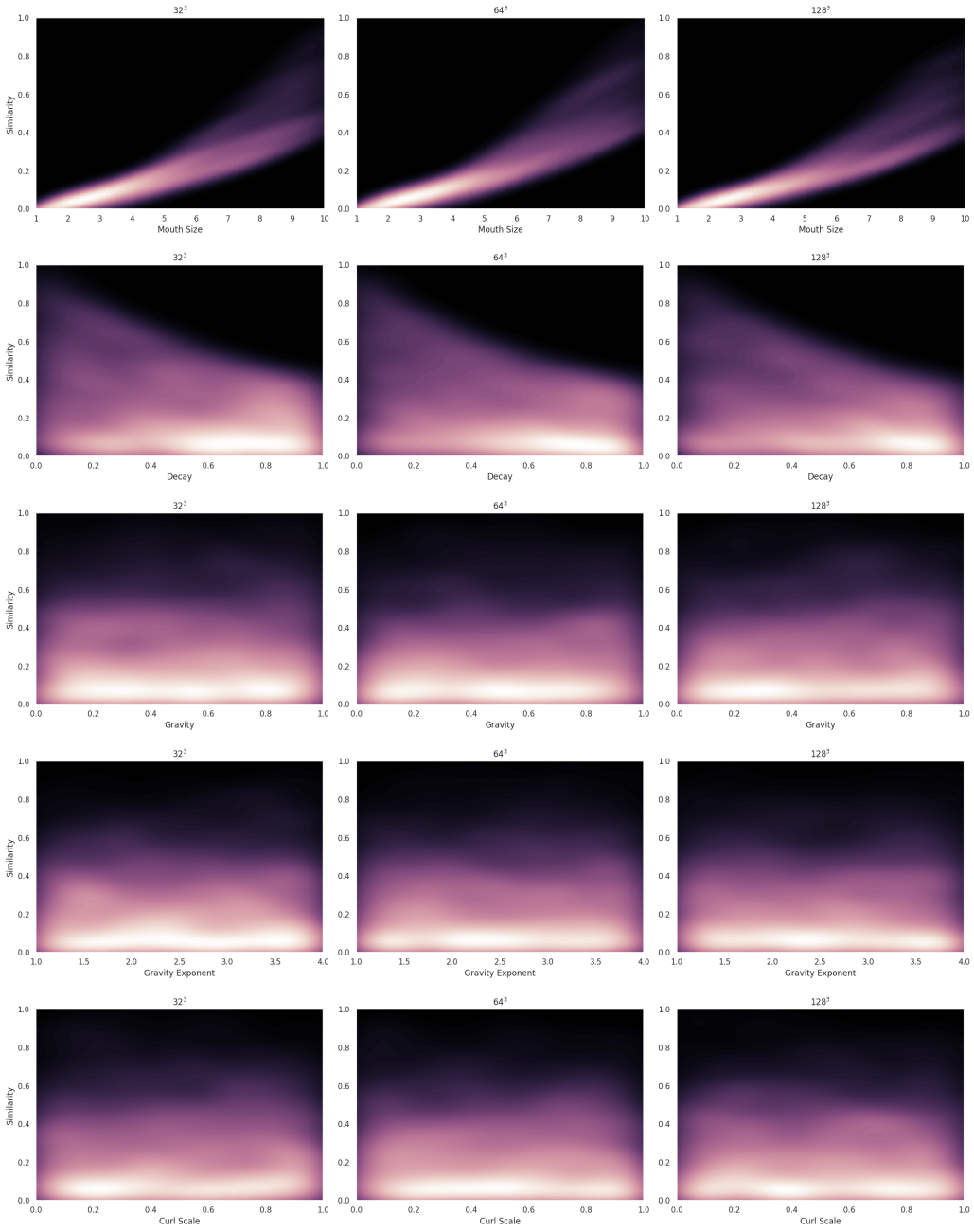

Figure 4.30: Expressive ranges of generated caves

4.3.3 Performance

The times taken for each generator are shown in Tables 4.7, 4.8 and 4.9. Their respective plots are found in Figure 4.31. Timing values across all generators are significantly below 1 second and overhang generation can occur at multiple resolutions under 1.5ms. The rise in timing is expected due to the voxel grid dimensionality being increased exponentially.

Similarly, arch and cave generation times increase exponentially as the dimension of the voxel grid being used increases. At each dimension value, the generators are used with multiple iterations. This governs the number of passes being used to determine the amount of kernel dispatches. Arch and cave generation times increase as the number of iterations along the curve are increased primarily due to the multi-pass approach where multiple kernels are dispatched, instead of a single kernel being executed like the cliff generator.

Overall, the subsecond nature of all of the generator functions allow for fast computation of volumetric features. Overhangs in particular are fast enough for real-time generation at lower dimensions. This is highly useful for fast design iteration and environmental artists can utilise the highly-expressive nature of the generators with fast feedback so that any aesthetic changes that are required can be made very quickly.

Resolution	Minimum	Maximum	Mean	Std. Dev.
32^3	0.00832	0.01341	0.00876	0.00037
64^3	0.02624	0.03555	0.02691	0.00101
128^3	0.13872	0.38179	0.14438	0.01295
256^3	1.06794	1.09104	1.07060	0.00441

Table 4.7: Time to generate cliffs for multiple voxel grid resolutions (in ms)

Resolution	Iterations	Minimum	Maximum	Mean	Std. Dev.
32^3	10	0.98282	3.25808	1.16302	0.20633
	20	1.91162	5.40877	2.36224	0.38349
	30	2.84506	5.92038	3.46095	0.45627
	40	3.79216	5.99610	4.59596	0.50146
	50	4.76378	8.31517	5.66575	0.61607
64^3	10	1.02560	2.44397	1.38609	0.19776
	20	2.05821	5.81616	2.71649	0.44613
	30	2.97837	5.38928	3.85756	0.40980
	40	4.01024	6.48112	5.18394	0.49977
	50	5.13562	8.39584	6.46433	0.63109
128^3	10	1.68934	2.92934	1.92245	0.20689
	20	3.28195	5.04755	3.75437	0.33924
	30	4.87210	7.28326	5.53656	0.46252
	40	6.46694	9.74243	7.32021	0.59390
	50	8.08477	11.00374	9.05837	0.63000
256^3	10	7.07082	8.28365	7.40785	0.17533
	20	13.69488	16.61802	14.34275	0.27386
	30	20.29037	22.51341	21.13733	0.38274
	40	26.94301	29.91786	28.00817	0.39368
	50	33.65401	36.48944	34.88928	0.41374

Table 4.8: Time to generate arches for variable iterations and multiple voxel grid resolutions (in ms)

Resolution	Iterations	Minimum	Maximum	Mean	Std. Dev.
32^3	10	1.00189	1.91501	1.20805	0.19901
	20	1.96326	5.94874	2.45119	0.43211
	30	2.90954	5.42022	3.53451	0.45819
	40	3.85078	6.29328	4.64443	0.53051
	50	4.82592	7.54518	5.60993	0.60207
64^3	10	1.06198	3.31190	1.38802	0.24004
	20	2.06691	5.77680	2.67187	0.43783
	30	3.05738	5.41814	3.90914	0.43086
	40	4.05187	7.46701	5.17292	0.53594
	50	5.21155	8.27187	6.53216	0.62907
128^3	10	1.68278	2.96560	1.92321	0.19268
	20	3.30982	6.49773	3.74602	0.37661
	30	4.91859	7.83203	5.53841	0.44678
	40	6.55389	10.47510	7.31657	0.61292
	50	8.16886	11.21990	9.08094	0.64087
256^3	10	7.11037	8.17370	7.44690	0.16610
	20	13.84954	15.32355	14.41379	0.23596
	30	20.35264	22.51507	21.16449	0.34989
	40	26.97635	29.82602	28.02465	0.43893
	50	33.69434	36.37331	34.99312	0.53714

Table 4.9: Time to generate caves for variable iterations and multiple voxel grid resolutions (in ms)

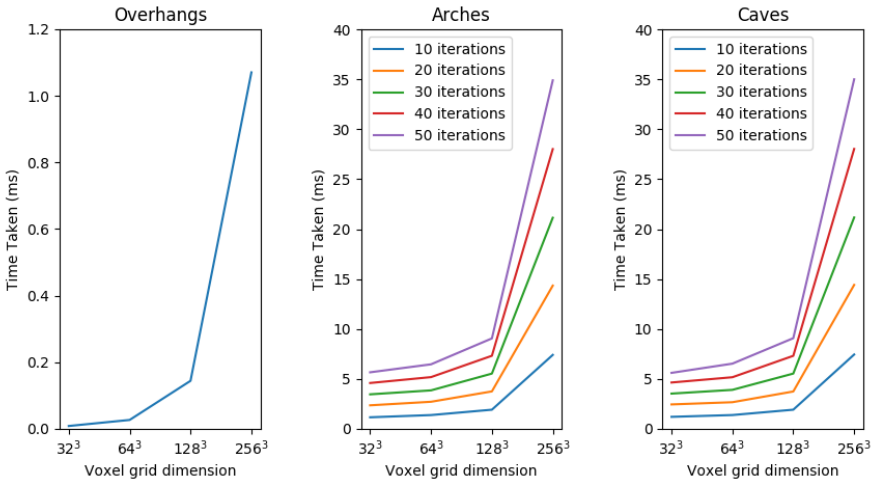

Figure 4.31: Plot of timings for each feature generator

4.3.4 PhyreEngine™ Integration

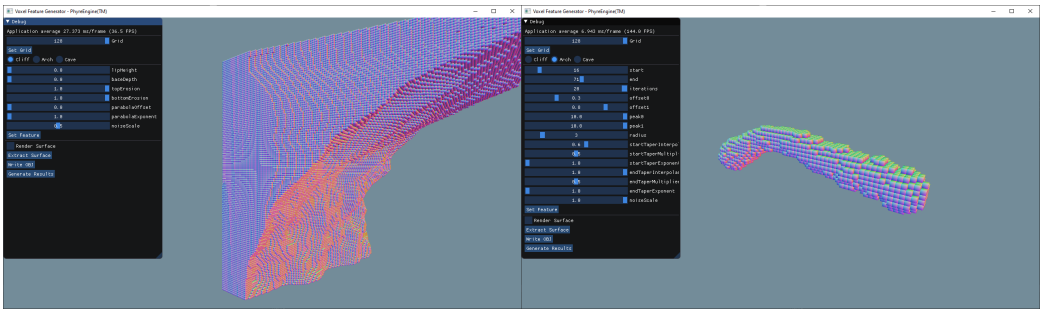

Figure 4.32: Screenshots of feature generator tool.

PhyreEngine™ makes use of a number of abstraction libraries written in C++, including a deferred renderer that is used to draw objects in a game world (Deering et al. 1988). In order to integrate this method into the game engine, classes were created to manage voxel data. A custom render pass was also designed that contained compute shaders that were responsible for populating the voxel buffers, extract the surface and pixel shaders that could render the data as voxels or polygons.

Parameters for each generator were exposed to the user via the use of an immediate mode GUI in a C++ application. As parameters were changed, the voxel buffer and surface was extracted in real-time so that a user has immediate feedback when modifying the terrain feature. Generators' parameters exposed in an immediate GUI interface in C++ which updated the voxel buffer, and then rendered the extracted surface. Example images of the tool can be found in 4.32.

4.4 Summary

This chapter presented three parametric generator functions that can be used to intuitively construct features found in terrains that are difficult to create using traditional heightmap-based approaches. These functions have been shown to generate plausible results by approximating physical erosion processes as computationally simpler mathematical constructs. Furthermore, while realism is desired in most virtual simulations, the functions presented are flexible enough to allow designers to apply their own artistic license if

natural terrains are not required. For example, this can apply to computer games where a particular aesthetic is desired. A wide variety of features can be generated, as has been demonstrably shown in our results.

Previous results for feature generation in Chapter 3 demonstrated construction of overhangs and caves using a grammar-based approach. The feature generators presented in this chapter show an alternative means of creating such features. Whilst voxel grammars provide a higher level abstraction to make features, the feature generators provide a further degree of control for individual features. This permits designers a more direct approach to creating features with a specific aesthetic in mind. One terrain feature that can be represented volumetrically that is not available in the voxel grammar approach is the naturally formed arch. The arch generator fulfils this niche and thus all three major features that cannot be represented with heightmaps can be created.

Compared to similar work for the creation of volumetric terrain features, our method provides performant, art-directable and highly expressive procedural generators.

Warping of heightmaps with a vector field (Gamito & Musgrave 2001) requires a complex pipeline to construct the field in order to achieve a particular aesthetic for overhangs. Furthermore, different vector fields have to be constructed when different overhangs are constructed adding to the complexity of the method. Our method allows for direct and powerful control over the shape, size and parameters associated with natural overhangs. Immediate feedback from our generator function, as well as a limited yet powerful set of parameters, means that many overhang variations that align with an artist's vision can be created in a short amount of time.

Similarly, fast feedback is an advantage of our method as well. The maximum times for a 256^3 resolution cliff, arch and cave using the presented generator functions are *1.09ms*, *36.49ms* and *36.37ms*, respectively. Methods that simulate or approximate erosion to generate features are much slower than the functions we have presented in this chapter, such as Beneš (2006) which takes *several hours*, Anh et al. (2007) which executes in *11.2 seconds*, and Becher et al. (2017) that computes its pipeline in *5.82 seconds*.

Furthermore, while the current implementation of the system uses a simple linear buffer for voxel storage, the generators are framework agnostic. This means that they can be implemented into any existing framework that utilises volumetric data, assuming that there is direct access to the underlying voxels. Such generality is a highly useful property to ensure that the

method can work with many different platforms, engines and frameworks. This differs from other methods such as Peytavie et al. (2009) and Santamaría-Ibirika et al. (2013) that rely on specific volumetric data formats that is required as part of their respective methods.

The generators that have been defined have all been implemented using CUDA and demonstrate the high level of parallelism that is highly beneficial for the speed of feature generation. This has been demonstrated by the profiling carried out for a number of generations.

Chapter 5

Conclusion

Terrains are an essential ingredient to any game or 3D simulation that requires an outdoor environment. Environment designers are given tools to generate terrains and require the capability to populate these terrains with features of varying degrees of fidelity and realism, depending on the desired aesthetics. The amount of effort to construct these features must also be considered when developing methods of terrain generation, and as such procedural methods are frequently utilised to assist designers.

Prevailing terrain methods generally rely on a heightmap-based approach, and there has been a vast array of research into the generation of heightmap-based terrains (Smelik et al. 2009). However, these cannot represent overhangs, naturally-formed arches and caves. Some research has attempted to do so, but has resulted in complex, unintuitive and difficult to control methods (Gamito & Musgrave 2001).

In order to do so effectively, volumetric terrain representations are preferred as they provide the ability to construct features with concave topological structures. However, the literature associated with procedurally generating volumetric terrain features is sparse. This thesis presents two novel methods for volumetric terrain feature generation to contribute to this research gap.

5.1 Contributions and Limitations

The main contributions of this thesis are two novel methods for designing and generating features for volumetric terrains. It also contributes an original

method to quantitatively measure the expressivity of procedural generators for terrain features.

5.1.1 Contribution 1 - Voxel Grammars

The first contribution is a novel grammar-based approach to generate overhangs and caves that has been inspired by shape grammars Stiny (1980) found in Chapter 3. A symbol consisting of predicates that acts on a subgrid of voxels is used to determine whether a subsequent matrix of transformation operators is executed. This has resulted in varied features without having to manually edit existing voxels and can be generated via the creation of a set of rules by the environmental designer.

However, designing rulesets requires sufficient experimentation before the artistic direction of the user can be realised. While the presented method is capable of procedurally generating overhangs and caves from a simple ruleset, the expended effort to initially write a grammar can be a delicate process. Depending on the desired aesthetic, it is important for the grammar to be robust against potential edge cases that can occur. For this reason, it is important to recall the design advice presented in Section 3.2.5.

While each rule could operate on an unlimited number of voxels, predicates and transformations only operated on a single voxel at a time. It became clear that the granularity at which voxel grammars executed was too low a level to be able to create rulesets in a timely manner.

5.1.2 Contribution 2 - Feature Generators

The theorised solution to the granularity issue of voxel grammars was to formulate a method that consisted of parametric functions that could operate on a number of voxels at the same time. This notion resulted in the conception of the feature generators found in Chapter 4, which are the second contribution of this thesis. In order to mitigate the granularity issue highlighted by the voxel grammar approach, a higher-level approach to generating volumetric terrain features was explored, defined and created. This resulted in three procedural generators to construct overhangs, arches and caves. Each generator consisted of a number of parameters that could be defined by the user to generate a particular terrain feature.

A limitation of the generators is that there is no method to effectively combine them currently. This is due to the additive nature of the overhang

and arch generators, as well as the subtractive nature of the cave generator. If these methods are applied to same space of voxels, then whichever function occurred last would have preferential treatment, potentially overriding desirable qualities that were created by the previous functions.

5.1.3 Contribution 3 - Quantitative Expressive Range

The third contribution of our thesis arose from a need to be able to quantitatively define the expressive range of the feature generator functions. This was achieved by applying the heuristic from Snape et al. (2016) to the generated voxels from each generator for each of their parameters. This reduced the dimensionality of a grid of voxels to a single real value, which in turn could be plotted as kernel density estimation plots using combinations of two parameters as axes. The parameter combinations could then be determined to be expressive or not by checking how uniformly distributed the plot points were. To our knowledge, this method of measuring expressive range of generated voxels is original.

5.2 Outputs and PhyreEngine™ Integration

Beyond the research contributions of this thesis, the collaboration with Sony Interactive Entertinment and PhyreEngine™ meant that outputs of this research included the integration of these methods into the game engine. The state of the engine beforehand utilised heightmap-based terrains only and had limited tooling for terrain editing. Prior to any of the methods being integrated into the engine, voxel management and volumetric terrain support had to be added. This was done in the form of a sparse, linear, block-allocated voxel buffer. The interface to this buffer enabled modification of voxels, constructing a GPU-friendly format of the voxel data (i.e. compact, thread-safe and contiguous).

PhyreEngine™ already had support for compute shaders which informed the design decision to pass the voxel data to the GPU. This enabled the surface extraction algorithm and the feature generators to be designed for massive parallelism. This ensured that the generators could be fast enough for real-time interaction by environment designers.

Tooling was designed and constructed to provide designers the ability to interact with the developed methods. For the voxel grammars, a ruleset

uses a textual representation in a JSON file that is loaded by the engine. The boundaries of the voxel grid and the number of iterations the grammar processes for is defined by the user in the tool. The feature generators user interface consisted of sliders to change each parameter of a particular feature, and rendering could be switched between drawing voxels or the extracted surface mesh. These tool decisions allowed for interactive editing and iteration for environment designers.

5.3 Future Work

This research has made substantial progress into offering methods for volumetric terrain feature generation. However, there are still areasof it that could be improved and extended. The realism offered by features constructed from volumetric data is dependent on the resolution of the data itself. Higher resolution voxel grids directly affect time complexity (there are more voxels to generate and/or manipulate) and memory complexity (there are more voxels stored). Therefore, the improvements to the methods focus on ameliorating performance in both of these areas, by utilising parallelisation and compression. Extensions to the work are considered that focus on providing a more intuitive experience for users and further automate the generation of voxels.

5.3.1 Improvements

Parallelisation

In order to maximise performance of the grammar derivation process, it is prudent to consider parallelisation as a potential method of optimisation. Lipp et al. (2009) introduce a method to perform L-system derivations in highly parallel architectures. They do so by first creating an efficient encoding of each production rule in order to maximise throughput on GPUs and then execute multiple passes over the encodings to perform the derivation step. Branching is handled by utilising dynamic dispatch, a modern GPU feature that dynamically generates workloads equal to the number of potential branches. Parallelisation of our grammar can take inspiration from this work by creating encodings for each of the rules and performing multiple passes on the various steps of the derivation process. First, all predicates can be evaluated in parallel using a kernel that determines the predicate's type and applies the condition. Then, each block of voxels that passes the

predicate check can be manipulated using the appropriate transform. This multipass approach could optimise the derivation process of the grammar to work more effectively on large voxel spaces.

Compression and Scalability

The size of the voxel data also needs to be monitored so that it does not grow to too large a magnitude. This would have the ability to optimise the grammars Sparse Voxel Octrees (SVOs) (Crassin et al. 2009) and brickmaps (Christensen & Batali 2004) are two methods optimised for GPU usage that are appropriate for this task. Furthermore, support for terrains consisting of several materials would require multiple density values denoting the amount of each material. Extending this level of detail to a singular voxel increases its size and the work of Dado et al. (2016) could be utilised to compress the data.

The platform-agnostic nature of the methods in this thesis offer the advantage of being able to use any sparse data structure to make this improvement. Larger quantities of voxels are required for higher fidelity of resulting features. Therefore, robust and production-ready technology such as *OpenVDB* (Museth et al. 2013) could be used to achieve increased levels of scalability.

5.3.2 Extensions

Abstract Voxel Grammars

The presented voxel grammar work highlighted complexity issues due to the granularity of its operations. As feature generators were constructed as a higher-level of abstraction to create volumetric terrain features, an intuitive extension to the voxel grammar is further abstraction.

The symbols and transforms of the voxel grammar can be replaced by higher-level predicates and transforms. For example, instead of a symbol that tests every voxel in a subgrid, a symbol that calculates the mean density of the subgrid and evaluate a condition based on this value can replace it. Similarly, transforms that manipulate single voxels at a time can be substituted for generators such as local heightmaps, noise functions (either additively or subtractively) or the feature generators defined in Chapter 4.

This design could be a more intuitive and less error-prone of defining rules to generate features for volumetric terrains.

Machine Learning

A further direction that can be explored is a method to create a more automated process for generating the grammars themselves. Machine learning technologies, such as deep convolutional neural networks (Goodfellow et al. 2016), may suit the underlying training requirements of generating a grammar.

Specifically, generative rules could be deduced by training on existing terrain data found in nature and creating rules that can form features in terrains closer to their natural counterparts. However, given that feature generation takes a number of parameters, this can form a high-dimensional problem which could be resolved using methods from the field of approximate dynamic programming (Powell 2007) and reinforcement learning (Sutton & Barto 1998).

Bibliography

Abrash, M. (1997), *Michael Abrash's Graphics Programming Black Book, with CD: The Complete Works of Graphics Master, Michael Abrash*, Coriolis group books.

Akenine-Möller, T. & Aila, T. (2005), 'Conservative and Tiled Rasterization Using a Modified Triangle Set-Up', *Journal of Graphics, GPU, and Game Tools* **10**(3), 1–8.

Akenine-Moller, T., Haines, E. & Hoffman, N. (2018), *Real-time rendering*, AK Peters/CRC Press.

Allegorithmic (n.d.), 'Substance Designer 5'.
URL: *https://www.allegorithmic.com/products/substance-designer*

Angelidis, A., Neyret, F., Singh, K. & Nowrouzezahrai, D. (2006), 'A controllable, fast and stable basis for vortex based smoke simulation', *Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation* **27**(3), 25–32.
URL: *http://portal.acm.org/citation.cfm?id=1218068*

Anh, N. H., Sourin, A. & Aswani, P. (2007), 'Physically based hydraulic erosion simulation on graphics processing unit', *Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia - GRAPHITE '07* **1**(212), 257.
URL: *http://dl.acm.org/citation.cfm?id=1321261.1321308*

Baert, J., Lagae, A. & Dutré, P. (2014), 'Out-of-core construction of sparse voxel octrees', *Computer Graphics Forum* **33**(6), 220–227.

Becher, M., Krone, M., Reina, G. & Ertl, T. (2017), Feature-based volumetric terrain generation, *in* 'Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games', ACM, p. 10.

Beneš, B. (2006), 'Physically-based hydraulic erosion', *Proceedings of the 22nd Spring Conference on Computer Graphics - SCCG '06* **1**(212), 17–22 PAGE@5.
URL: *http://dl.acm.org/citation.cfm?id=2602161.2602163*

Beneš, B. & Arriaga, X. (2005), 'Table mountains by virtual erosion', *Natural Phenomena* **xx**, 33–39.

Beneš, B., Šťava, O., Měch, R. & Miller, G. (2011), 'Guided Procedural Modeling', *Computer Graphics Forum* **30**(2), 325–334.
URL: *http://doi.wiley.com/10.1111/j.1467-8659.2011.01886.x*

Bretz, J. H. (1942), 'Vadose and phreatic features of limestone caverns', *The Journal of Geology* **50**(6), 675–811.
URL: *http://www.jstor.org/stable/30060299*

Bridson, R., Houriham, J. & Nordenstam, M. (2007), Curl-noise for procedural fluid flow, *in* 'ACM Transactions on Graphics (TOG)', Vol. 26, ACM, p. 46.

Cepero, M. (2010), 'From Voxels To Polygons'.
URL: *http://procworld.blogspot.co.uk/2010/11/from-voxels-to-polygons.html*

Cerezo, E., Pérez, F., Pueyo, X., Seron, F. J. & Sillion, F. X. (2005), 'A survey on participating media rendering techniques', *Visual Computer* **21**(5), 303–328.

Chen, H. & Fang, S. (1998), 'Fast Voxelization of Three-Dimensional Synthetic Objects', *Journal of Graphics Tools* **3**(4), 33–45.

Christensen, P. H. & Batali, D. (2004), An irradiance atlas for global illumination in complex production scenes, *in* 'Proceedings of the Fifteenth Eurographics Conference on Rendering Techniques', EGSR'04, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, pp. 133–141.

Collet, Y. (2013), 'Lz4-extremely fast compression'.

Crassin, C., Neyret, F., Lefebvre, S. & Eisemann, E. (2009), Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering, *in* 'Proceedings of the 2009 symposium on Interactive 3D graphics and games', ACM, pp. 15–22.

Crow, F. C. (1977), Shadow algorithms for computer graphics, *in* 'ACM SIGGRAPH Computer Graphics', Vol. 11, ACM, pp. 242–248.
URL: *http://portal.acm.org/citation.cfm?doid=965141.563901*

Crytek (n.d.), 'CryEngine 3'.
URL: *http://cryengine.com/*

Dado, B., Kol, T. R., Bauszat, P., Thiery, J. M. & Eisemann, E. (2016), 'Geometry and attribute compression for voxel scenes', *Computer Graphics Forum* **35**(2), 397–407.

Dang, M., Ceylan, D., Neubert, B. & Pauly, M. (2014), 'SAFE: Structure-aware facade editing', *Computer Graphics Forum* **33**(2), 83–93.
URL: *http://doi.wiley.com/10.1111/cgf.12313*

Deering, M., Winner, S., Schediwy, B., Duffy, C. & Hunt, N. (1988), 'The triangle processor and normal vector shader: a vlsi system for high performance graphics', *ACM SIGGRAPH Computer Graphics* **22**(4), 21–30.

Dey, R., Doig, J. G. & Gatzidis, C. (2017), Procedural feature generation for volumetric terrains, *in* 'ACM SIGGRAPH 2017 Posters', ACM, p. 64.

Dey, R., Doig, J. G. & Gatzidis, C. (2018), 'Procedural feature generation for volumetric terrains using voxel grammars', *Entertainment Computing* **27**, 128–136.

Dey, R. & Konert, J. (2016), *Content Generation for Serious Games*, Springer International Publishing, Cham, pp. 174–188.

Ebert, D. S., Musgrave, F. K., Peachey, D., Perlin, K. & Worley, S. (2002), *Texturing and Modeling: A Procedural Approach (The Morgan Kaufmann Series in Computer Graphics)*, Morgan Kaufmann.

Eisemann, E. & Décoret, X. (2006), 'Fast scene voxelization and applications', *Proceedings of the 2006 symposium on Interactive 3D graphics and games - SI3D '06* (May), 71.
URL: *http://portal.acm.org/citation.cfm?doid=1111411.1111424*

Eisemann, E. & Décoret, X. (2008), 'Single-pass GPU Solid Voxelization for Real-time Applications', *Proceedings of Graphics Interface 2008* pp. 73–80. URL: *http://dl.acm.org/citation.cfm?id=1375714.1375728*

Epic Games (n.d.), 'Unreal Engine 4'.
URL: *www.unrealengine.com*

Ericson, C. (2004), *Real-time collision detection*, CRC Press.

Everitt, C. (2001), 'Interactive order-independent transparency', *White paper, nVIDIA* **2**(6), 7.

Fang, S. & Chen, H. (2000), 'Hardware Accelerated Voxelization', *Computers & Graphics* .

Forest, V., Barthe, L. & Paulin, M. (2009), 'Real-time hierarchical binary-scene voxelization', *Journal of Graphics, GPU, and Game Tools* **14**(3), 21–34.

Forsyth, T. (2006), 'Linear-speed vertex cache optimisation'.

Fournier, A., Fussell, D. & Carpenter, L. (1982), 'Computer rendering of stochastic models', *Communications of the ACM* **25**(6), 371–384.

Frisken, S. F., Perry, R. N., Rockwood, A. P. & Jones, T. R. (2000), Adaptively sampled distance fields: A general representation of shape for computer graphics, *in* 'Proceedings of the 27th annual conference on Computer graphics and interactive techniques', ACM Press/Addison-Wesley Publishing Co., pp. 249–254.

Galerne, B., Lagae, A., Lefebvre, S. & Drettakis, G. (2012), 'Gabor noise by example', *ACM Transactions on Graphics* **31**(4), 1–9.

Gamito, M. N. & Musgrave, F. K. (2001), Procedural landscapes with overhangs, *in* '10th Portuguese Computer Graphics Meeting', Vol. 2, p. 3.

Gearbox Software (2009), 'Borderlands', http://www.borderlands.com/. [Online; accessed 03-January-2019].

Geiss, R. (2007), 'Generating complex procedural terrains using the gpu', *GPU gems* **3**, 7–37.

Gibson, S. F. (1998), Constrained elastic surface nets: Generating smooth surfaces from binary segmented data, *in* 'International Conference on Medical Image Computing and Computer-Assisted Intervention', Springer, pp. 888–898.

Golomb, S. (1966), 'Run-length encodings (corresp.)', *IEEE transactions on information theory* **12**(3), 399–401.

Goodfellow, I., Bengio, Y., Courville, A. & Bengio, Y. (2016), *Deep learning*, Vol. 1, MIT press Cambridge.

Greene, N. (1989), 'Voxel space automata: Modeling with stochastic growth processes in voxel space', *Siggraph* **23**(3), 175–184.

Gustavson, S. (2005), 'Simplex noise demystified', *Linköping University, Linköping, Sweden, Research Report* .

Hales, T. C. (2007), 'The Jordan curve theorem, formally and informally', *American Mathematical Monthly* **114**(10), 882–894.
URL: *http://www.jstor.org/stable/27642361*

Hart, J. C. (1996), 'Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces', *The Visual Computer* **12**(10), 527–545.

Hello Games (2016), 'No man's sky', http://www.nomanssky.com/. [Online; accessed 03-January-2019].

Hillaire, S. (2015), 'Physically based and unified volumetric rendering in frostbite', *SIGGRAPH Advances in Real-Time Rendering course* pp. 570–610.

Ho, C. C., Wu, F. C., Chen, B. Y., Chuang, Y. Y. & Ouhyoung, M. (2005), 'Cubical marching squares: Adaptive feature preserving surface extraction from volume data', *Computer Graphics Forum* **24**(3), 537–545.

Hoetzlein, R. K. (2016), Gvdb: raytracing sparse voxel database structures on the gpu, *in* 'Proceedings of High Performance Graphics', Eurographics Association, pp. 109–117.

Hudák, M. & Durikovic, R. (2011), 'Terrain Models for Mass Movement Erosion', *EG UK Theory and Practice of Computer Graphics* pp. 1–8.

Huggett, R. (2016), *Fundamentals of geomorphology*, Routledge.

Jennings, J. (1971), 'An introduction to systematic geomorphology. volume 7, karst'.

Jönsson, D., Sundén, E., Ynnerman, A. & Ropinski, T. (2012), State of The Art Report on Interactive Volume Rendering with Volumetric Illumination, *in* 'EG 2012 - State of the Art Reports', Vol. 1, pp. 53–74.

Ju, T., Losasso, F., Schaefer, S. & Warren, J. (2002), 'Dual contouring of hermite data', *ACM Transactions on Graphics* **21**(3).
URL: *http://portal.acm.org/citation.cfm?doid=566654.566586*

Kämpe, V., Sintorn, E. & Assarsson, U. (2013), 'High resolution sparse voxel DAGs', *ACM Transactions on Graphics* **32**(4), 1.

Karabassi, E.-A., Papaioannou, G. & Theoharis, T. (1999), 'A Fast Depth-Buffer-Based Voxelization Algorithm', *Journal of Graphics Tools* **4**(4), 5–10.

Keinert, B., Schäfer, H., Korndörfer, J., Ganse, U. & Stamminger, M. (2014), Enhanced Sphere Tracing, *in* A. Giachetti, ed., 'Smart Tools and Apps for Graphics - Eurographics Italian Chapter Conference', The Eurographics Association.

Kelly, T. & Wonka, P. (2011), 'Interactive architectural modeling with procedural extrusions', *ACM Transactions on Graphics* **30**(2), 1–15.
URL: *http://portal.acm.org/citation.cfm?doid=1944846.1944854*

Kirk, D. et al. (2007), Nvidia cuda software and gpu parallel computing architecture, *in* 'ISMM', Vol. 7, pp. 103–104.

Koca, Ç. & Güdükbay, U. (2014), 'A hybrid representation for modeling, interactive editing, and real-time visualization of terrains with volumetric features', *International Journal of Geographical Information Science* **28**(9), 1821–1847.

Komma, P., Fischer, J., Duffner, F. & Bartz, D. (2007), Lossless volume data compression schemes., *in* 'SimVis', pp. 169–182.

Lagae, A., Lefebvre, S., Drettakis, G. & Dutré, P. (2009), Procedural noise using sparse gabor convolution, *in* 'ACM Transactions on Graphics (TOG)', Vol. 28, ACM, p. 54.

Laine, S. & Karras, T. (2011), 'Efficient sparse voxel octrees', *IEEE Transactions on Visualization and Computer Graphics* **17**(8), 1048–1059.

Laur, D. & Hanrahan, P. (1991), Hierarchical splatting: A progressive refinement algorithm for volume rendering, *in* 'ACM SIGGRAPH Computer Graphics', Vol. 25, ACM, pp. 285–288.

Lewiner, T., Mello, V., Peixoto, A., Pesco, S. & Lopes, H. (2010), 'Fast generation of pointerless octree duals', *Eurographics Symposium on Geometry Processing* **29**(5), 1661–1669.

Li, W., Fan, Z., Wei, X. & Kaufman, A. (2005), Flow Simulation with Complex Boundaries, *in* 'GPU Gems 2', Vol. 2, Addison-Wesley Professional, pp. 747–764.

Liao, D. (2008), 'GPU-accelerated multi-valued solid voxelization by slice functions in real time', *Proceedings of The 7th ACM SIGGRAPH International Conference on Virtual-Reality Continuum and Its Applications in Industry - VRCAI '08* **1**(212), 1.
URL: *http://portal.acm.org/citation.cfm?doid=1477862.1477886*

Lipp, M., Wonka, P. & Wimmer, M. (2008), Interactive visual editing of grammars for procedural architecture, *in* 'ACM SIGGRAPH 2008 papers on - SIGGRAPH '08', Vol. 27, p. 1.
URL: *http://portal.acm.org/citation.cfm?doid=1399504.1360701*

Lipp, M., Wonka, P. & Wimmer, M. (2009), Parallel generation of l-systems., *in* 'VMV', pp. 205–214.

Löffler, F., Müller, A. & Schumann, H. (2011), 'Real-time Rendering of Stack-based Terrains', *Vision, Modeling, and Visualization* .
URL: *http://www.informatik.uni-rostock.de/ schumann/papers/2010+/VMV2011.pdf*

Lorensen, W. E. & Cline, H. E. (1987), Marching cubes: A high resolution 3D surface construction algorithm, *in* 'ACM SIGGRAPH Computer Graphics', Vol. 21, ACM, pp. 163–169.
URL: *http://portal.acm.org/citation.cfm?doid=37402.37422*

Lysenko, M. (2012), 'Smooth Voxel Terrain (Part 2)', https://0fps.net/2012/07/12/smooth-voxel-terrain-part-2/. [Online; accessed 30-August-2016].

McMillen, E. & Himsl, F. (2011), 'The binding of isaac', http://www.bindingofisaac.com/. [Online; accessed 03-January-2019].

Microsoft (n.d.), 'DirectX 11'.
URL: *https://www.microsoft.com/en-gb/download/details.aspx?id=6812*

Milan, I., Joe, K., Aaron, L. & Charles, H. (2013), 'Volume rendering techniques', *Book Randima Fernando, GPU Gems NVIDIA. http://http. developer. nvidia. com/GPUGems/gpugems- ch39. html. Last accessed* pp. 667–672.

Miller, G. S. (1986), The definition and rendering of terrain maps, *in* 'ACM SIGGRAPH Computer Graphics', Vol. 20, ACM, pp. 39–48.

Mojang (2011), 'Minecraft'.
URL: *https://minecraft.net/*

Müller, P., Wonka, P., Haegler, S., Ulmer, A. & Van Gool, L. (2006*a*), 'Procedural modeling of buildings', *ACM Transactions on Graphics* **25**(3), 614.

Müller, P., Wonka, P., Haegler, S., Ulmer, A. & Van Gool, L. (2006*b*), 'Procedural modeling of buildings', *ACM Transactions on Graphics* **25**(3), 614.

Museth, K. (2013), 'Vdb: High-resolution sparse volumes with dynamic topology', *ACM Trans. Graph.* **32**(3), 27:1–27:22.

Museth, K., Lait, J., Johanson, J., Budsberg, J., Henderson, R., Alden, M., Cucka, P., Hill, D. & Pearce, A. (2013), Openvdb: an open-source data structure and toolkit for high-resolution volumes, *in* 'Acm siggraph 2013 courses', ACM, p. 19.

Nowrouzezahrai, D., Johnson, J., Selle, A., Lacewell, D., Kaschalk, M. & Jarosz, W. (2011), 'A programmable system for artistic volumetric lighting', *ACM Transactions on Graphics* **30**(4), 1.
URL: *http://portal.acm.org/citation.cfm?doid=2010324.1964924*

Parberry, I. (2015), 'Modeling Real-World Terrain with Exponentially Distributed Noise', *Journal of Computer Graphics Techniques* **4**(2), 1–9.

Parish, Y. I. & Müller, P. (2001), Procedural modeling of cities, *in* 'Proceedings of the 28th annual conference on Computer graphics and interactive techniques', ACM, pp. 301–308.

Perlin, K. (1985), An image synthesizer, *in* 'Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques', SIGGRAPH '85, Association for Computing Machinery, New York, NY, USA, p. 287–296.
URL: *https://doi.org/10.1145/325334.325247*

Perlin, K. (2002), Improving noise, *in* 'ACM Transactions on Graphics (TOG)', Vol. 21, ACM, pp. 681–682.

Perlin, K. & Hoffert, E. M. (1989), Hypertexture, *in* 'ACM Siggraph Computer Graphics', Vol. 23, ACM, pp. 253–262.

Perlin, K. & Neyret, F. (2001), Flow noise, *in* '28th International Conference on Computer Graphics and Interactive Techniques (Technical Sketches and Applications)', SIGGRAPH, p. 187.

Peytavie, A., Galin, E., Grosjean, J. & Merillou, S. (2009), 'Arches: a Framework for Modeling Complex Terrains', *Computer Graphics Forum* **28**(2), 457–467.

Planetside Software (n.d.), 'Terragen 3'.
URL: *http://planetside.co.uk/products/terragen3*

Poston, T., Wong, T. T. & Heng, P. a. (1998), 'Multiresolution Isosurface Extraction with Adaptive Skeleton Climbing', *Computer Graphics Forum* **17**(3), 137–148.
URL: *http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.27.1075*

Powell, W. B. (2007), *Approximate Dynamic Programming: Solving the curses of dimensionality*, John Wiley & Sons.

Pranckevičius, A. (2018), 'How does a gpu shader core work?'.

Prusinkiewicz, P. & Lindenmayer, A. (2012), *The algorithmic beauty of plants*, Springer Science & Business Media.

Quilez, I. & Jeremias, P. (2013), 'Shadertoy beta'.
URL: *https://www.shadertoy.com/*

Raman, S. & Wenger, R. (2008), 'Quality isosurface mesh generation using an extended marching cubes lookup table', *Computer Graphics Forum* **27**(3), 791–798.

Santamaría-Ibirika, A., Cantero, X., Salazar, M., Devesa, J., Santos, I., Huerta, S. & Bringas, P. G. (2013), 'Procedural approach to volumetric terrain generation', *The Visual Computer* **30**(9), 997–1007.

Schaefer, S., Ju, T. & Warren, J. (2007), 'Manifold dual contouring', *IEEE Transactions on Visualization and Computer Graphics* **13**(3), 610–619.

Schmitz, L., Scheidegger, L. F., Osmari, D. K., Dietrich, C. A. & Comba, J. L. (2010), 'Efficient and quality contouring algorithms on the GPU', *Computer Graphics Forum* **29**(8), 2569–2578.
URL: *http://doi.wiley.com/10.1111/j.1467-8659.2010.01825.x*

Scholz, M., Bender, J. & Dachsbacher, C. (2013), 'Level of Detail for Real-Time Volumetric Terrain Rendering', *VMV 2013: Vision, Modeling & Visualization* **24**(17), 211–218.
URL: *http://diglib.eg.org/EG/DL/PE/VMV/VMV13/211-218.pdf*

Schwarz, M. & Seidel, H.-P. (2010), 'Fast parallel surface and solid voxelization on GPUs', *ACM Transactions on Graphics* **29**(6), 1.
URL: *http://dl.acm.org/citation.cfm?id=1866201 http://portal.acm.org/citation.cfm?doid=1882261.1866201*

Shaker, N., Togelius, J. & Nelson, M. J. (2016), *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, Springer.

Smelik, R. M., De Kraker, K. J., Tutenel, T., Bidarra, R. & Groenewegen, S. A. (2009), A survey of procedural methods for terrain modelling, *in* 'Proceedings of the CASA Workshop on 3D Advanced Media In Gaming And Simulation (3AMIGAS)', pp. 25–34.

Snape, P., Pszczolkowski, S., Zafeiriou, S., Tzimiropoulos, G., Ledig, C. & Rueckert, D. (2016), 'A robust similarity measure for volumetric image registration with outliers', *Image and Vision Computing* **52**, 97–113.

Stiny, G. (1980), 'Introduction to shape and shape grammars', *Environment and Planning B: Planning and Design* **7**(November), 343–351.

Sutton, R. S. & Barto, A. G. (1998), *Reinforcement learning: An introduction*, Vol. 1, MIT press Cambridge.

Swoboda, M. (2012), 'Advanced Procedural Rendering in DirectX 11', http://www.gdcvault.com/play/1015455/Advanced-Procedural-Rendering-with-DirectX. [Online; accessed 05-January-2017].

Swoboda, M. (2013), 'Real Time Ray Tracing Part 2', http://directtovideo.wordpress.com/2013/05/08/real-time-ray-tracing-part-2/. [Online; accessed 02-April-2016].

Tasse, F., Emilien, A. & Cani, M.-p. (2014), 'First Person Sketch-based Terrain Editing', *Graphics Interface ...* **2014**, 217–224.
URL: *http://hal.inria.fr/hal-00976689/*

Tatarchuk, N. (2015), Advances in Real Time Rendering, Part I, *in* 'ACM SIGGRAPH 2015 Courses', SIGGRAPH '15, ACM, New York, NY, USA.

Thiedemann, S., Henrich, N., Grosch, T. & Müller, S. (2011), Voxel-based global illumination, *in* 'Symposium on Interactive 3D Graphics and Games', ACM, pp. 103–110.

Togelius, J., Shaker, N. & Nelson, M. J. (2016), Evaluating content generators, *in* N. Shaker, J. Togelius & M. J. Nelson, eds, 'Procedural Content Generation in Games: A Textbook and an Overview of Current Research', Springer, pp. 215–224.

Unity (n.d.), 'Unity Game Engine-Official Site'.

Wang, L., Yu, Y., Zhou, K. & Guo, B. (2011), 'Multiscale vector volumes', *ACM Transactions on Graphics* **30**(6), 1.

Wonka, P., Wimmer, M., Sillion, F. & Ribarsky, W. (2003), 'Instant architecture', *ACM Transactions on Graphics* **22**(3), 669.

Worley, S. (1996), A cellular texture basis function, *in* 'Proceedings of the 23rd annual conference on Computer graphics and interactive techniques', ACM, pp. 291–294.

Wrenninge, M. (2012), *Production Volume Rendering: Design and Implementation*, CRC Press.

Wyman, C. (2011), 'Voxelized shadow volumes', *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics - HPG '11* p. 33. URL: *http://dl.acm.org/citation.cfm?doid=2018323.2018329*