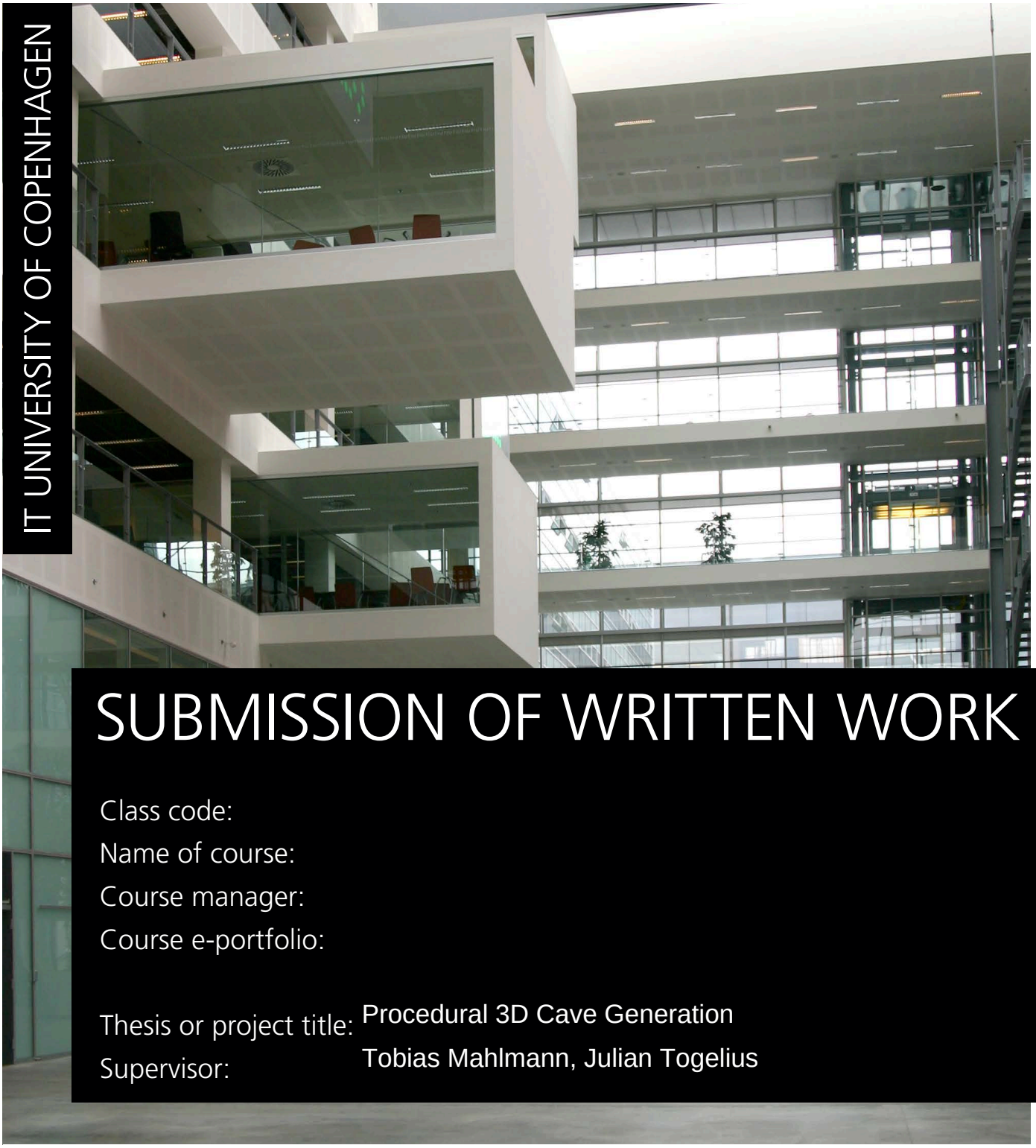
IT UNIVERSITY OF COPENHAGEN



SUBMISSION OF WRITTEN WORK

Class code:

Name of course:

Course manager:

Course e-portfolio:

Thesis or project title: Procedural 3D Cave Generation

Supervisor: Tobias Mahlmann, Julian Togelius

	Full Name:	Birthdate (dd/mm-yyyy):	E-mail:
1.	Benjamin Mark	20-03-1988	bmar@itu.dk
2.	Tudor Berechet	20-04-1989	tdbe@itu.dk
3.			@itu.dk
4.			@itu.dk
5.			@itu.dk
6.			@itu.dk
7.			@itu.dk

Procedural 3D Cave Generation

Benjamin Mark, Tudor Berechet

A thesis presented for the degree of
Masters of Science

Games Technology
IT University of Copenhagen
Denmark
September 2014

Division of work

Tudor Berechet created the overall structure of the GPGPU pipeline, and is generally responsible for modelling the metaball-based detail of the caves. He is also responsible for the isosurface extraction, shading and texturing of the caves.

Benjamin Mark is responsible for the L-systems and modelling the overall flow and structure of the caves. He is also responsible for implementing the tool for interactive evolution of the cave structure, as well as functionality for saving and loading caves for future re-generation.

The authors worked together to make the structuring and detailing methods leverage each other to create a compelling environment. They also worked together to create the initial GPU Cellular Automata based approach to carving out the cave, as well as the creation and distribution of features in the cave, such as stalactites.

Abstract

Procedural content generation is the algorithmical construction of models, textures or alterations of objects in virtual worlds. This field has exploded in recent years, taking advantage of the immense parallel computational power of modern videocards. It is especially popular in the form of real-time content generation for video games, but is also used in 3d graphics, simulation systems, virtual reality, and the movie industry.

Although procedurally generated content is already being used in a sizeable chunk of the mainstream video game industry, procedural caves in 3D space have remained elusive, with only a few unsophisticated approaches to speak of. Our research goal for this thesis is to explore the possibilities for, and develop an approach to, procedurally generating believable and compelling three-dimensional cave landscapes, which satisfy common gameplay requirements, using the GPU[1].

To achieve this goal, we employ the use of L-Systems to construct the structure, chambers, and overall flow of the cave systems. We also implement a series of GPU programming layers which use a metaball ”brush” distorted by various noise functions to three-dimensionally sculpt the cave interiors. We present experimental results which show the relationships between our processes and their ability to produce cave landscapes. Furthermore, features such as stalactites and columns are generated through the use of Cellular Automata and inserted as a step in our GPU programming pipeline.

In order to display the caves as in-game assets (3D meshes), we apply an isosurface extraction algorithm to our point density data. The geometry is then enhanced further using procedural texturing, lighting, and bumpmapping, through shader programming.

Our resulting system is capable of procedurally generating cave networks in real time on low end gaming graphics cards and rendering the resulting geometry as in-game terrain. In addition, our tool allows a certain degree of control over the generation process, which includes features such as confining the caves to a certain volume in the virtual world, guaranteeing full path connectivity, influencing the direction and target of the tunnels, the frequency of the generated landmarks, the amplitude of distortion, and the option to preview, choose and combine different L-system structures.

CONTENTS

I. Introduction

The procedural construction of virtual content, commonly known as Procedural Content Generation (PCG), has become very popular within the fields of 3D graphics, simulation systems, virtual reality, the movie industry and especially the video game industry.
This approach to content creation has both benefits and drawbacks, but before those can be discussed, it is important to define PCG.

Togelius et al.[70] outline the term by looking at various types of content creation, which they define overall as "The creation of game content automatically using algorithms"[70]. The authors, however, make a distinction between content created by players, both inside and outside of a game, such as levels created in a level editor, and the content created algorithmically either without human input or without the player having the intent to create content.
As an example they use the Civilization games[78], where the deliberate creation of a civilization by the player is not defined as PCG, even though some things, such as the growth of cities, are determined based on the game rules without player input. They argue that because the player's intent is to create a civilization, it cannot be considered PCG. On the other hand, the civilizations generated by the AI in the game can be considered PCG even though it is ultimately still created in response to player actions. This distiction is made as the player does not plan his actions to deliberately create certain AI civilizations. Instead he focuses on creating his own, and is often not aware which of his actions guide the development of AI civilizations.
This definition is, however, not without problems when moving into the realm of mixed initiative[2] PCG. In this case, a designer guides an algorithm to create a piece of content. In some cases such as the tool shown by Smelik et al [8], the content is generated algorithmically, but in direct response to the designers deliberate actions. It is even possible for the designer to edit the level in the same fashion as a level editor, while the terrain around the edited area adjusts to the input by re-generating itself accordingly.
This makes the distinction between what is and is not PCG very vague, and the definition of PCG will vary from person to person. Regardless of this, it is safe to say that any content generation system where the input from a human is used more to mold automatically generated content than to generate the content itself can be considered to be some form of PCG.

With the term PCG defined, we look at some of its benefits and drawbacks. The main drawbacks are the lack of control over the design process and the performance constraints for real-time generation. Even with these negative aspects in mind, a number of benefits can also be drawn from this approach.

[2]A mixture of procedural generation and designer control

Firstly, even though hand crafted content allows the designer to implement their vision completely into the game and controlling every detail to create the experience they have in mind for the player, this process takes a lot of time and can demand an overwhelming amount of assets, even for large studios with many resources. PCG is a viable alternative, as while it does take away some of the control from the designer, it allows to dramatically cut down on the time spent creating environments for a game, and also allows for a larger variety of content, as well as infinite variation of the materials or structure of the content (within certain patterns).

Secondly, procedurally generating most of the content translates into using only a fraction of the storage space that would otherwise be used for artist created assets. Even using PCG only to provide different permutations of the same character model would cut storage space and development time. No Man's Sky[85], developed by a tiny studio, procedurally generates an entire (populated) universe of star systems, planets, landscapes, fauna, flora etc. When a player visits a planet, it gets generated from a small amount of seed data in its entirety, and when they leave it, everything but the seed is discarded.

Thirdly, procedurally generated assets are never final and can always be modified at a later stage in the production with minimal cost. Similarly, they can also be heavily edited just before or during run-time. A developer can embrace and leverage this particular aspect of PCG to create content which changes every playthrough, or it can be changed during gameplay according to the player's actions; whether it is player constructed content (e.g. Minecraft[81]), procedurally built, warped and animated assets and levels (e.g. Krautscape[86]), or procedural characters and dungeons (e.g. Rogue Legacy [76]). This offers a huge potential for replayability, as each play session can be unique.

With the benefits listed above in mind, it would seem that PCG is a good fit for most games. That is, however, not necessarily the case. Every project needs different types of content, and what fits one project will probably not fit another. As an example, the content that is needed for an open world game with high graphical fidelity is not the same that is needed for a roguelike platformer. Where the first game would need detailed models of everything from houses to rocks, the second needs simple 2D sprites and a wide variety of different levels. This means that any content generators for these games would need different properties in order to satisfy the requirements of each game.
Togelius et al.[71] provides a list of some of these desirable properties, such as speed, reliability and controllability. There are, however, usually trade-offs between these properties, and a content generator cannot hope to have them all.

Going back to the example above, a generator for a vast open world game, would need its environments to be believable and relatively diverse. On the other hand the

process does not necessarily need to be fast if the content is generated offline[3]. At the same time, it does not have to guarantee that the range of each procedural function's output dependably generates realistic results (ie. "reliability"). For instance an oddly shaped flower is not as undesirable as an impassable dungeon corridor.
On the other hand, a generator for a roguelike platformer would need to run during gameplay, and it is therefore required for it to be fast. Roguelike games also generally rely on replayability, so a great expressive range is needed while at the same time guaranteeing a certain level of reliability, in that it must conform to a specific set of rules and affordances. On the other hand, it does not necessarily need to look very organic, or be controllable by a designer.
With that in mind, it is very important when creating a content generator, that the requirements for the generated content are fully understood, so it can be made to focus on the properties that are most relevant.

A. Caves in Games

Caves are a fundamental part of terrain for many games, both old and modern. Their use ranges from providing depth to an open world game like Minecraft[81], to being the core environment in games like Descent[83] or Permutation Racer[84].

One common approach to creating caves in video games is for a designer to 3D-model cave sections. These sections are then imported into the game as assets and fit together based on some human or procedural metric (used for caves in Skyrim[110]). This approach can be further enhanced by applying a procedural normalmap displacement to the cave tiles based on worldspace noise lookups (to ensure the distorted tiles seamlessly fit together). Having to use pre-made cave sections does, however, reduce the variety of caves that can be crafted, and makes caves take longer to create and perfect.

Another option is to allow the user to carve out the game world three-dimensionally themselves, as seen in Everquest Next Landmark[87]. This means that in order for a designer (or player) to sculpt a cave, the developers need to undertake the huge task of representing their entire engine through voxels, and storing everything as sculptable, persistent, point cloud data. This is however not often a practical and feasible prospect (and it does not relieve human labour).

Creating caves in a purely procedural manner is a more challenging task than generating most other landscapes, and have not received much coverage in the academic world or in the industry. Unlike regular terrain, caves are enclosed structures, and can therefore not be represented by a height map; which acts by vertically displacing a horizontal plane only, and is the most common approach used on terrain (Smelik et al. 2009[6]). Additionally, it is not trivial to procedurally design the overall structure and direction of the cave systems.

[3]Not generated during gameplay

B. Our Objectives

The main research goal of this thesis is to explore the possible approaches to procedurally generating believable and compelling three-dimensional cave landscapes, which satisfy common gameplay requirements, using the GPU. These caves will not be designed to be standalone gameplay experiences, however, but rather as an addition to a larger world. A possible application of this cave generation method would be to populate a large open world with many and varied caves.

As a solution, we will propose a pipeline for automatic cave generation. The structure of the caves and chambers will rely on an L-System, which along with a random seed will represent the only data required to create, and if desired, save the environment. The voxel data for the cave will be generated at run-time using a series of HLSL Direct Compute[111] layers. The techniques involved in manipulating the voxels in order to model cave geometry and features, such as stalactites, include multiple types of noise, a metaball, and Cellular Automata. The surface of the caves will be extracted from the voxel data, and rasterized into meshes. Further detail will then be added to the meshes using procedural texturing, lighting and bumpmapping through shader programming.

The resulting software will run in real-time on low end gaming graphics cards, provided they support DirectX11 (Direct Compute). The solution will run as an interactive demo built with the free version of the Unity3D[116] game engine. The landscapes of the demo will be explorable using a 3rd-person human character and/or a space ship.
The creation of the caves will also offer a certain degree of customizability to a designer, which include the option to guarantee full path connectivity with few dead ends, confining the caves to a certain volume in the virtual world, influencing the direction and target of the tunnels, changing the frequency of the generated landmarks, the amplitude of distortion, and the option to preview, choose and mate different L-system structures.

As mentioned in the introduction, it is important when creating a content generator to identify which properties are important to achieve. Referring again to the properties mentioned by Togelius et al.[71], our solution will focus especially on believability and expressivity and less on reliability. This means that varied landscapes are an important goal, and therefore that procedural emergence is a big focus. The issue that comes with emergent topology is that it fundamentally lacks reliability. The challenge is then to balance the procedures such that the results remain visually coherent and never cease to suspend disbelief in the portrayed environment. Procedural, "out-of-the-box" and "weird" shapes will exist and are even desired, as long as they do not

break immersion. To maintain believability we will attempt to constrain the generator to a spectrum of shapes identified from a set of natural phenomena.

To continue on the point of expressivity and variation, as seen in the case of Dragon Age 2[77], players notice if similar environments are repeated over and over during gameplay[79]. As our generator is meant to produce many different caves within a pre-existing world, they all need to be differently structured to maintain the illusion of a natural world. A large expressive range allows for this variety in the look and feel of caves.

Apart from believability and expressivity, speed and controllability are also important to some degree. While very quick generation of caves is not required for the generator to fill the role described above, if online generation[4] is required, a high generation speed is necessary to produce the cave environment faster than the player can explore it.

Controllability is also important to some degree, as one type and shape of cave does not fit all purposes. It is, for instance, convenient if a cave can be shaped to fit within certain constraints, such as a hill or mountain range to avoid interfering with the rest of the terrain. Similarly it is desirable to be able to tweak various parameters to achieve just the style of cave that fits a specific application and style of environment.

The last property mentioned by Togelius et al, reliability, is actually not that important for this solution. While a designer should be able to steer the generator towards certain objectives and styles of caves, it is not paramount that these objectives and certain strict topology rules are rigidly adhered to. As caves in the natural world come in all shapes and sizes, it is not necessarily a failure if a cave does not always exhibit all desired traits at the same time. For instance, if long tunnels are generally desired for caves, and a single cave that consists of one big room is generated, it would not be seen by the player as a failure, but rather just as a natural oddity compared to the other caves in the world. In other words, nature would not guarantee us a list of features when we enter a random cave.

C. Contribution

Our proposed solution potentially benefits areas that develop virtual environments, such as video games which involve world exploration or space flight, and uses caves as part of their design.

To date, there has been little or no academic work published on employing the use of Direct Compute shaders (or similar technology such as CUDA[112] or OpenCL[114]) to create 3D voxel-based geometry on the video card. We describe the surprisingly many, vital, and completely different aspects and sub-fields which must be tackled in order to successfully build the whole that makes a procedurally generated landscape.
We highlight the challenges that must be overcome in this field, and show the experimental results from our various techniques. Unlike other solutions, we also provide procedural means of modelling the overall structure of the cave system, and provide a certain degree of customizability for the designer.

Our results provide compelling cave environments with features, rough surfaces and realistic procedural textures, while also satisfying basic gameplay requirements such as connectivity, tunnels, chambers and obstacles.

Furthermore, this thesis presents the existing state-of-the-art PCG terrain generation techniques found both in academia and in the industry, comparing them to our approaches and adaptations for cave structures, as well as comparing to other related work which specifically regards procedural caves.

D. Structure of this Thesis

From here on, the thesis is structured thusly:

Section II contains a literature review of relevant work done in the field of procedural landscape generation, with particular focus towards solving the challenges of cave generation and real-time procedural content creation for video games. The analysed work is primarily from academia, but approaches from the industry are also looked at.
This section also contains an analysis of the natural geologic phenomena responsible for constructing cave structures in the real world. Additionally, the main relevant technological methods and algorithms of approaching PCG, have been researched and compared in the last part of this section.

Section III on page 25 describes our methodology and gives an overview of the structure of our application, outlining our attempts and justifying our decisions, while referring to the approaches and algorithms in section II. The capabilities of our solution are explained, with each major component being analysed in its own subsection. Finally the setup of our user survey is described.

Section IV on page 51 presents the capabilities of our tool, as well as the results of the user survey.

Section V on page 55 concludes the thesis by discussing what was learned from the user evaluation and the viability of our solution. It also outlines the possible directions in which future work can further improve our results.

II. Related Work and Technologies

This section contains a review of the literature on related technologies and areas of research within the field of landscape generation.

[4]Content generation during gameplay

The section starts by reviewing the various known techniques of modelling procedural landscapes and their application, found both in academia and the industry. We relate the methods to the special case of generating caves and the challenges involved, as well as cover methods which regard caves specifically.
Furthermore, we review the geological phenomena involved in the creation of natural cave formations, identifying and isolating the main processes involved and the features they produce.

Secondly, in order to procedurally model the overall structure of the cave network, a grammar system will be used. This section also includes a review of the use of L-systems for structuring procedural content.

Finally, voxels for real-time procedural terrain must be managed within 3D data structures and processed on the GPU. Therefore, we review the available GPU programming methods and toolkits, as well as the documented types of spatial data management structures and their benefits.
A voxel isosurface[91] must then be extracted and displayed as geometry on modern videocards, which implies the use of polygon generation to be fit into the modern 3D graphics pipeline. We examine the different types of voxels used in the industry (what surface information they can and cannot provide), and their relationship to the available isosurface extraction algorithms. In addition, we review the present methods involving procedural shader programming for video games.

A. Noise and Fractal Based Distortion Procedures

1) Heightmaps

Height maps are a common and useful way to represent a 3D terrain, storing the height data in a 2D grid, with a height value associated to each grid point. This grid is then applied to a 2D mesh, and its vertices are distorted according to the height information in the map; figure 1, figure 2, figure 3. As elaborated in the next paragraphs, there are two typical ways of providing the height values for the heightmap distortion: sampling height values from a pre-generated noise texture, and direct vertex displacement according to some (usually fractal) equation (such as Random Midpoint Displacement).

2) Procedurally generated noise

Procedurally generated noise is a very big aspect of PCG. Due to its natural appearing distribution, it represents a dependable resource of infinitely fine detail in areas such as vector field perturbation in fluid dynamics simulations, procedural texture creation, texture lookup perturbation, vertex perturbation (including but not limited to height maps) etc.

The most trivial type of procedural noise would best be exemplified by a 2D texture in which each pixel holds a completely random value between black and white. This

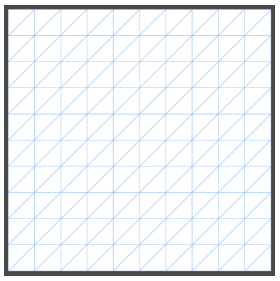


FIG. 1: A 2D plane mesh, prior to the application of the heightmap.

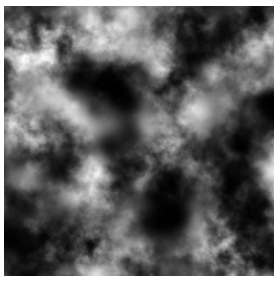


FIG. 2: A heightmap made out of noise, stored as a 2D texture[92]

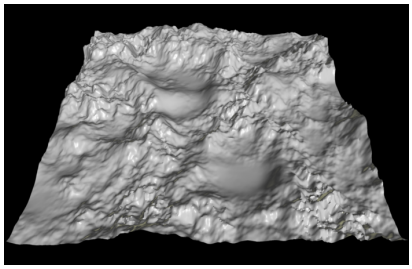


FIG. 3: The result of applying the heightmap to the 2D plane[92]

would essentially be white noise, and would visually resemble television static. Since there is no smooth transition between the random values, there are no noticeable ”interesting” or ”natural” emerging patterns distributed in it, and the random results are not repeatable (it cannot be reliably reconstructed given a certain input), this type of noise is of little usefulness.

Useful noise patterns for PCG require that for a given input, the resulting noise value is always the same, even though the resulting pattern appears to be random (hence pseudorandom). The noise distribution must also be smooth, such that two nearby noise values are relatively close to each other (there must be a gradient transition between noise values).
Ebert et al [35] present a list of ideal properties of noise functions as follows:

- noise is a repeatable pseudorandom function of its inputs.
- noise has a known range, namely, from -1 to 1.
- noise is band-limited, with a maximum frequency of about 1.
- noise does not exhibit obvious periodicities or regular patterns.
- noise is stationary - its statistical properties are invariant to to translation.
- noise is isotropic - its statistical properties are invariant to rotation.

A function which efficiently produces natural gradient noise on modern hardware is a very important resource for PCG. The following are a some of the most common and useful noise functions used for content generation.

The first is the Perlin Noise algorithm (Ken Perlin 1985

[10], improved in 2002 [11], Gustavson[36]). It is a type of lattice gradient noise, and is the most popular and influential (Lagae et al 2010 [37], [38]).
A lattice noise stores a random scalar value at each lattice point in a 3D (or other dimensional) array. Each lattice point has integer coordinates, and if the sampling of noise happens to be at such coordinates, the lattice noise function does a table lookup to find the value to return. Otherwise, for non-integer values, trilinear interpolation is used to calculate a value in-between the lattice points: figure 4. (Wolfe [39])

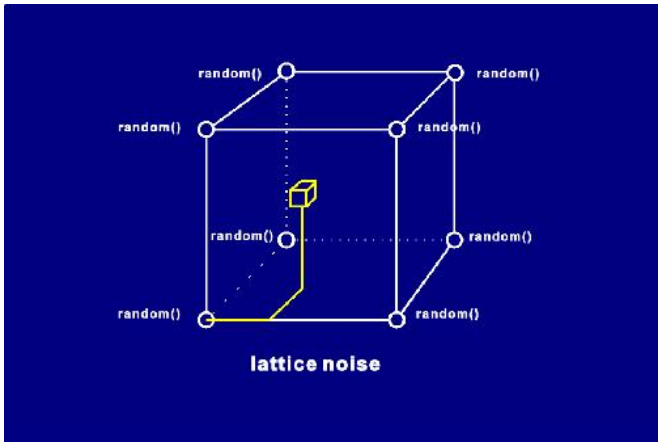


FIG. 4: Visualisation of trilinear interpolation on a lattice [39]

Perlin noise stores a pseudorandom gradient at each integer grid point (grids: 1D, 2D (squares), 3D (cubes), 4D (hypercube)), and interpolates the linear functions with the gradient vectors.
Figure 5 shows an example of a 2D grid, where each vertex (of integer coordinates) stores a pseudorandom gradient vector for the Perlin noise function. The value for a point P, situated in-between grid points, is the dot product of all the neighbouring gradient vectors and the vectors pointing towards P from each of the neightbouring grid points, as seen in figure 6. Bilinear Interpolation is used to calculate the contribution of each of the neighbours neighbour of P.

As previously mentioned, noise values must be pseudo-random, as they need to be repeatable, but at the same time provide enough variation to not have noticeable repeating patterns. Too much variation would result in unknown or unpredictable behaviour, however. The pseudorandom values for the lattices are typically provided as a lookup table, and represent unit-length vectors of successive redirections. Other methods includes the use of various hash functions seeded with the worldspace position, as well as permutation polynomials (bijections). The main goal of these methods being fast performance on either the CPU or GPU. Regardless of method, the gradients need to be reasonably evenly distributed to roughly cover all directions, and are typically unit-length vectors due to the lower performance requirement of calculating their dot product.

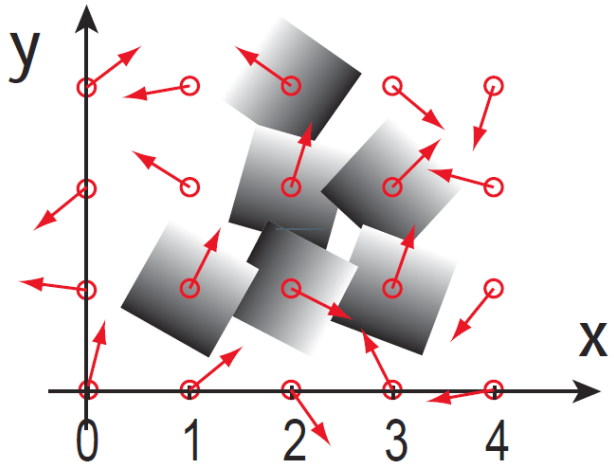


FIG. 5: Visualisation of a 2D Perlin lattice, where each point is a gradient vector[36]

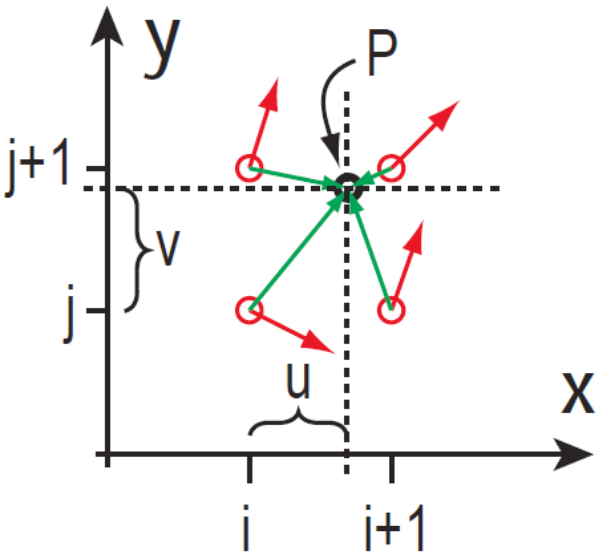


FIG. 6: The contribution of each 2D lattice neighbour point around point P. Red arrows represent the gradient vector, green ones the vectors pointing to P[36]

Simplex noise (Gustavson [36]) is a faster approach to noise which is also developed by Ken Perlin. It uses simplexes instead of cubes as its lattice.
A simplex represents the most compact n-dimensional shape, which in an n-dimensional grid, can fill the entire n-dimensional volume. In 2D, a simplex is represented by an equilateral triangle. In 3D, a simplex is a squished tetrahedron, 6 of which can fill a slightly skewed cube, and in 3 dimensions they would fill a slightly skewed hypercube.

The important aspect here, is that a simplex has as few corners as possible, which becomes a huge advantage when interpolating, especially at higher dimensions: an n-dimensional hypercube would have 2^n corners, whereas an n-dimensional simplex has $n+1$ corners. This reduces the complexity when

interpolating the values for a previously mentioned point P, found in-between lattice points. Interpolating for a hypercube with 2^n corners is $O(2^n)$ complex, whereas for a simplex it is $O(n^2)$.

Simplex noise also no longer requires actual interpolation. Calculating the gradient (normal) of an interpolated value is more difficult at higher dimensions on a cube/hypercube lattice. With a simplex however, the same result (of calculating the gradient) can be achieved by a weighted summation of the simplex' vertex gradient contributions. Due to these performance enhancements, Simplex noise is preferred over Perlin noise, especially when implemented on the GPU and at high dimensions (3D and especially 4D).

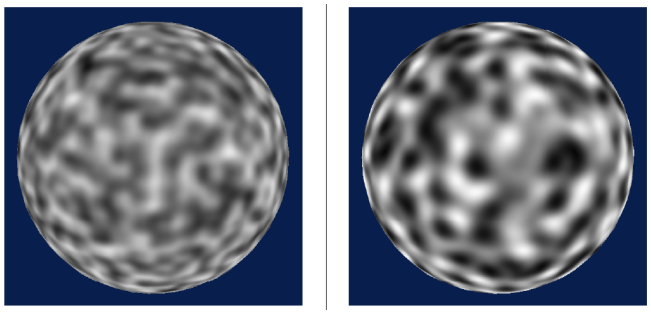


FIG. 7: Comparison image of 3D Perlin noise (left) and 3D Simplex noise, used to texture a sphere[36]

Curl Noise (Bridson et al[40]) is a solution for providing natural-looking fluid-like turbulence fields to particles, without actually simulating any fluid dynamics. It isn't however a type of noise in and of itself, but rather a curling function applied to existing noise.
The curl is a type of derivative which, when applied to a vector field, produces another vector field. It is particularly used to curl layers of 3D Perlin or Simplex noise. To approximate these otherwise more costly derivative calculations, vector potentials (consisting of different noise samples) are subtracted to approximate the same result.

Voronoi noise, or Worley noise (Steven Worley, Ebert et al [35]) is a type of cellular noise which is capable of creating patterns similar to those of a voronoi diagram. Instead of using a regular grid of pseudorandom values, the basic idea behind its generation is to replace this regular grid with scattered random "feature points" in 3D space, as shown in figure 10 on the next page. A voronoi boundaries, which are seen in figure 9, are formed by selecting an arbitrary point P in space and fetching the n nearest scattered feature points to describe the boundary.
The resulting noise can provide a wide range of boundary shapes and roughness depending on the number of feature points and the manner in which they were scattered. The range of results is further increased by combining the Voronoi noise with some other type of noise such as Simplex noise. Figure 11 on the next page shows such a combination, applied to a sphere as normalmapping.

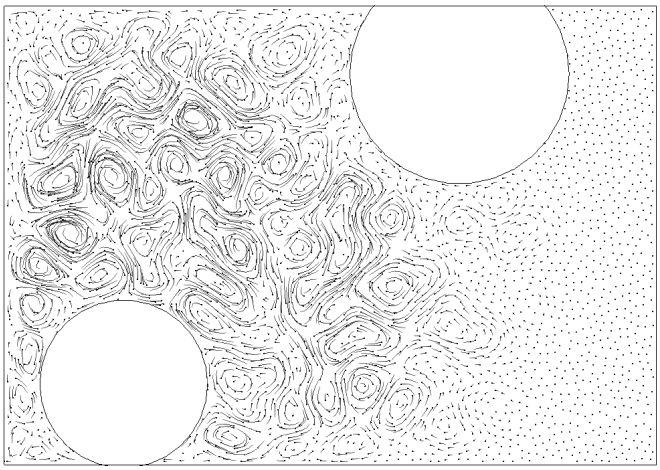


FIG. 8: 2D representation of a vector field perturbed by Curl noise (each perturbation has a weight which allows for the smooth transition between the evenly spaced points on the right and the curl on the left)[40]

Even though Worley noise is capable of producing smooth

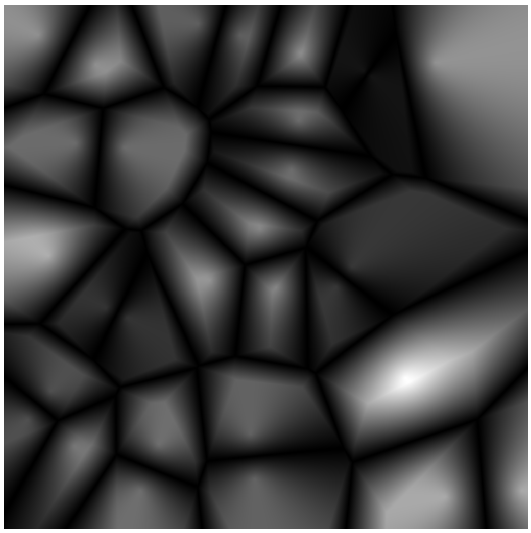


FIG. 9: 2D texture of Voronoi (Worley) noise[101]

spherical voronoi diagram shapes, the scatterings which offer sharp features are typically more useful, as they add roughness or boxiness to an existing terrain or other type of rough surface, and can create boundaries for procedural stone mosaics, as seen on figure 12 on the following page and figure 13 on the next page.

Procedural noise typically takes (multidimensional) spatial coordinates as input, in order to produce a repeatable noise value output for that point in space. This means that the frequency of the noise can be changed by multiplying or dividing the spatial coordinates before sending them to the noise function, and the amplitude can be change by similarly altering the end noise result. Fractal noise (noise with increasingly smaller layers of detail) can be obtained by adding together multiple layers of the same noise function with different frequencies and amplitudes. Since the

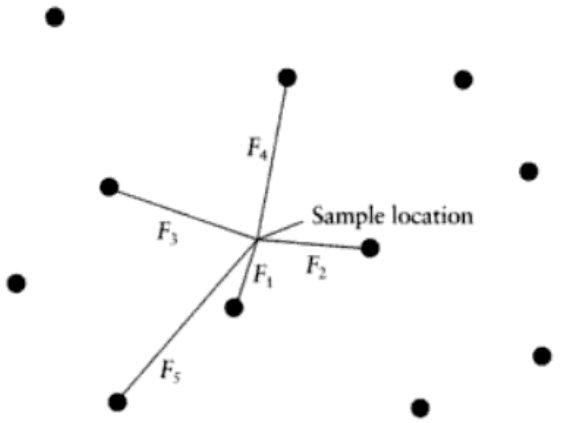


FIG. 10: Scattered 3D points. The distances between the closest points and the sample point, are marked with F_n[35]

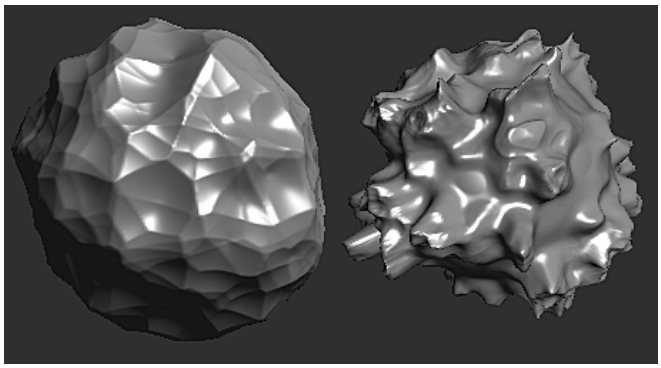


FIG. 11: A sphere distorted by Worley (Voronoi) noise (left), and a sphere distorted by a combination of Worley and Simplex noise[41]

frequencies are typically related by a factor of two, they are called noise octaves.

3) Midpoint Displacement

Random Midpoint Displacement[26] is a popular procedural terrain generation technique developed in 1982 by A. Fournier et al[27]. It recursively subdivides a mesh (most commonly a 2D plane), by adding vertices and displacing the midpoint vertex by a gaussian random amount within a certain range. The terrain is then partitioned into smaller squares, and the perturbation magnitude is reduced for each subdivision. This process is repeated until a desired square size is reached, and produces a distinctly fractal looking terrain. An example is shownin figure 14 on the following page.

B. 3D Terrain Generation Literature Review

Caves are simply a type of terrain, and it is therefore important to look at the methods that are used for generating terrain in general, as well as those that are used to generate caves in particular.

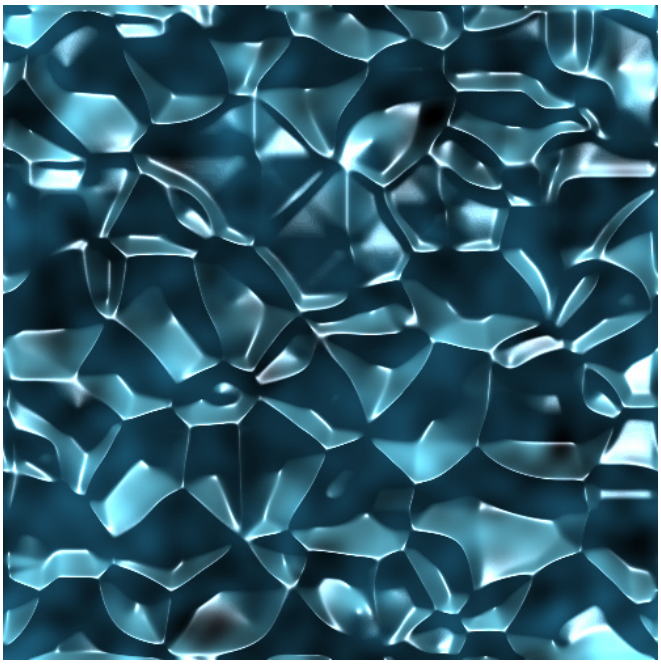


FIG. 12: Version of Worley noise applied as a heightmap onto a 2D plane[93]

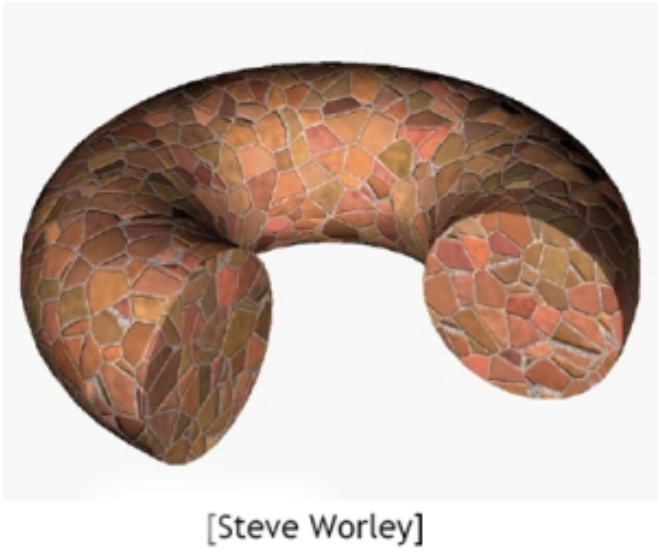


FIG. 13: Procedural stone tiles using Worley noise[35]

1) Heightmap Based Generation

A lot of the research that has been done into terrain generation, has focused on heightmap based approaches (section II-A.1 on page 4), and many of the algorithms that are dedicated to generating heightmap based terrain are based on a fractal approach. A notable example of this is shown by Fournier et al.[12] (figure 15 on the following page), which is one of the early examples of a midpoint displacement algorithm (section II-A.3) being used to generate both 2D and 3D terrain.

A different approach to midpoint displacement is shown by Ashlock et al.[34]. Instead of relying on random displacements

FIG. 14: Example of Midpoint Displacement[27]

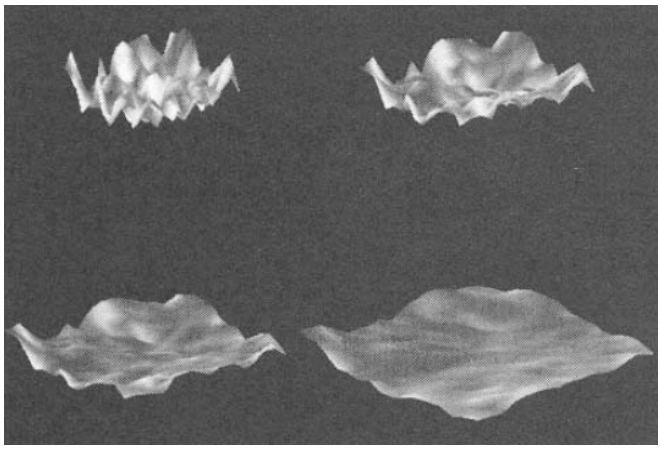


FIG. 15: Example of Midpoint Displacement used to vertically perturb a 2D plane[27]

of the midpoints, they rely on an L-system (which are explored in more detail in section II-C on page 14), to generate a set of displacement values. Instead of representing the L-system in the most common way, like a simple string, Ashlock et al. chose to have each symbol be expanded by a 2D 2x2 set of symbols. Each of these symbols has a fixed displacement value associated with it, which is used to generate the final heightmap. This removes some of the randomness usually associated with midpoint displacement algorithms, and restores some control of the final result. To be able to generate arbitrary terrains, they evolved both the production rules of the L-system and the displacement values in order to be able to generate various structures such as a hill and a crater.

Another approach to terrain generation is shown by Musgrave et al.[13]. This approach uses fractal noise values to generate a heightmap, which is then eroded with a physical erosion simulation.
The noise values are generated by a procedural noise function (section II-A.2 on page 4), in this case Perlin's famous noise function[10], in several octaves to achieve a fractal effect. Each octave has an increased frequency and a decreased amplitude, which makes it have less of an impact on the final noise value. This results in the fractal effect described above. Hydraulic erosion was simulated using cellular automata, where the amount of water and dissolved material that flows out to other cells is calculated based on the local slope of the elevation profile.
The fractal nature of the terrains that are generated with these methods closely mirror the fractal terrain that is often found in nature, where massive features give way to smaller and smaller details as you look closer.

A stochastic approach such as this one does not allow for a lot of control over the terrain generated, however. The user is limited mainly to tweaking various overall parameters in order to change the overall frequency and amplitude of the terrain, but cannot specify which types of terrain the user wants the heightmap to include. To get around this problem, Doran and Parberry[9] suggests another method for generating heightmap based terrain. Their approach is to use various software agents to raise an island out of the sea. These agents each create a specific feature of the terrain, such as beaches, mountains or rivers, allowing for control over the frequency of a specific landform, but not offering any direct control over where they occur; additionally, its performance is not interactive.

Smelik et al [7][8] goes a step further in ensuring control over the generated content. They present a mixed initiative approach to terrain generation, which allows the designer to first sketch a rough outline of the terrain in 2D, which then gets procedurally converted to actual 3D terrain. The designer can sketch several different layers such as mountains, cities, rivers and roads to generate a fully fledged world to use for their game.

While certain methods of heightmap distortion and manipulation can be useful when applied to 3D voxel volumes, the 2D perturbation approach is fundamentally unable to produce caves and overhangs, therefore we shall not focus on it in great detail, nor review any further literature in the field.

2) Fully three-dimensional Voxel Based Generation

While applying heightmaps to two-dimensional meshes offer a compact and easy way of storing terrain data, a voxel based approach offers a much more detailed representation of the terrain at the cost of memory. In a voxel based approach, data about the world is stored in a set of voxels that are positioned in a 3D grid. The main two benefits of this is that it is possible to effortlessly create caves and overhangs, and that data is available about each cube on the world grid. Due to the high memory cost of storing this

data, special data structures such as octrees or k-d trees[16] are often preferred to simple data structures such as 3D arrays.

Voxel landscapes and even flora or buildings can simply be constructed purely through the use of fractals, grammars or noise functions [Joeri VanDerLei[20]], and the results of using noise to make high quality terrain, even with overhangs, have been documented in chapter 1 of the GPU Gems 3 book[21] where a total of ten noise look-ups are used. Specific arches and overhangs are however difficult to produce purely with noise.

It is, however, possible to gain a more advanced real-time control over voxels, as shown by Greeff's MSc paper[14]. It explores ways of generating and storing voxel based terrain, including fractal methods such as midpoint displacement and perlin noise fractals.
A midpoint displacement[26] approach is used to subdivide and perturb single 2D lines across a plane in order to create initial valleys; figure 16. The valleys are subsequently eroded based on virtual water velocity fields calculated from the angles and depth of the initial valleyFigure 17.

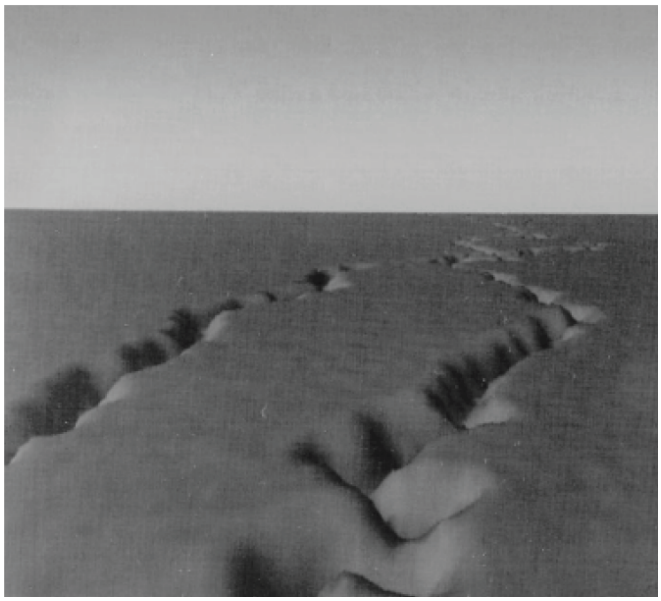


FIG. 16: Visualisation of a valley as a result of midpoint displacement, source Greeff[14]

This solution estimates erosion on landscapes, and generates rivers and mountains. However, to generate more advanced geometry like caves and overhangs, a mixed initiative PCG terrain system is developed, essentially leaving these complex features up to the designer to define. It is argued that a pure fractal or noise approach offers little control over the structure and desired features, and can only provide superficially realistic results.

An original algorithm, named Wires, is developed, which allows a designer to model the overall flow of the landscape,

FIG. 17: Valleys after simulated erosion[14]

or to build a specific landmark at a specific location. The principle is inspired by how an artist or a designer would build the initial rough skeleton of a clay sculpture out of metal wires, to provide an overall structure onto which to apply the finer details.
A designer can draw these "Wires" in 3D space and they will be used as displacement information for the terrain which wraps around them, after which fractal perturbations are applied. This allows for both the creation of new landmarks (figure 18), and the guiding of the flow of and the extrusion of wider structures such as mountain ranges (figure 19 on the next page). A version of voxel-carving Wires is also able to sculpt overhangs such as caves or the inner lining of river beds. The user can describe a wire representing the cross-section view of the start point and end point of a river, and the procedural carving interpolates the values in between, carving a river bed (figure 20 on the following page).

A hybrid rendering system for visualising discrete voxel

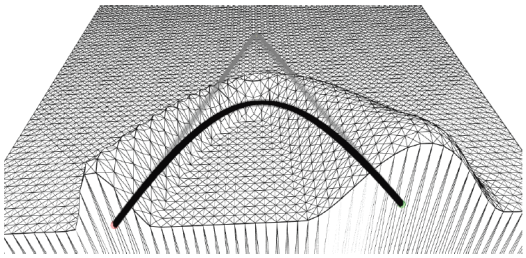


FIG. 18: Visualisation of a 3d spline drawn to create the base for a procedural landmark[14]

data, based on the Marching Cubes algorithm[1], had to be implemented to allow for real-time rendering of the interactive (changing) voxel data.

Another MSc. paper, Cui[22], explores methods of pro-

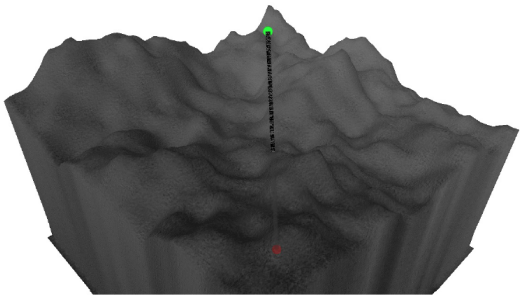


FIG. 19: A Wire with a wide range of influence, extruding a mountain range[14]

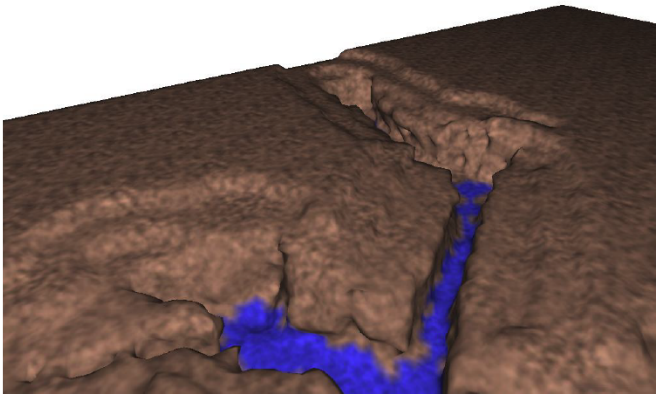


FIG. 20: Mixed initiative carved riverbed segments with overhangs, connected by procedural cross-sections[14]

cessing voxels for the creation of caves with stalactites and stalagmites. It does not focus on modelling tunnel networks for the cave, instead uses primitives such as a sphere for the main structure. The focus is instead on examining different Perlin, Wavelet and Simplex noise patterns and bias values between the ”rock” and ”air” voxels, combining multiple octaves in order to create satisfactory natural-appearing cave walls. It presents experimental results and empirical observations, showing the relationship between the applied functions and the resulting cave structures. The data is stored in an octree structure, upon which a flood-fill algorithm is applied to resolve the problem of floating geometry and air pockets. After the isosurface extraction, a poligonal mesh smoothing technique was used to correct the blockiness of the voxel data, and to patch occasional cracks in the mesh. Figure 21 and figure 22 show Cui's results.

Outside of academia, voxel based approaches are explored more vigorously: Cepero [15], Betts [99], Hansmeyer [100], Alexander[17], as it is a technique that allows for more interesting terrains and features than a heightmap based approach. This is due to its inherent ability to represent caves and overhangs with ease.

As our approach to generating caves uses voxels, this area of terrain generation, and voxel manipulation to model a general overarching structure of the landscape, are of special interest, as many of the problems facing voxel based terrain

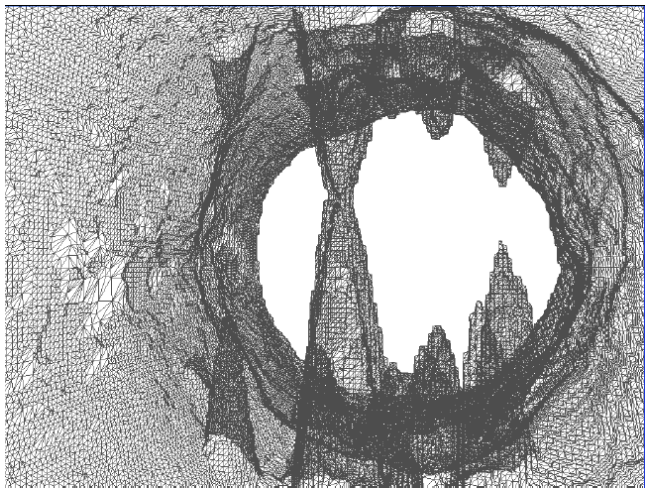


FIG. 21: Wireframe of a sample of Cui's results[22].

FIG. 22: Cave; sample of Cui's results[22]

generation also translate to voxel based cave generation.

3) Other Terrain Generation Methods

Besides purely heightmap or voxel based approaches, there are a few different approaches that try to reach a compromise between the two.

Benes and Forsbach[18] present a representation based on horizontal stratified layers, aiming to simulate terrain evolution and erosion. They argue that a voxel data structure is too slow and memory-consuming, and that a heightmap, although fast, is too imprecise and cannot simulate erosion well. They propose a middle-ground solution, where their entire terrain is represented as a 2D grid, where each cell of the grid holds a sparse 1D array of stratification information. It is reasoned that the layers of different types of material are too thick, and therefore wasteful, for a 3D voxel structure. Figure 23 on the next page shows the layers to be stored in each 2D grid point; each one contains information about terrain density, moisture level, granularity etc.. Together these

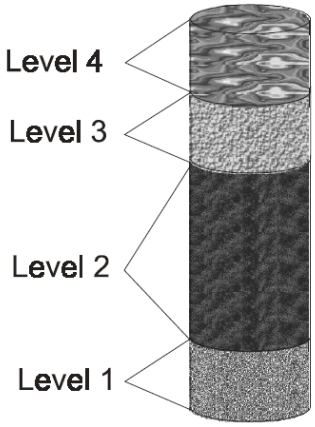


FIG. 23: Example of stratified layers, each of which would be stored in a cell of the 2D grid representing the terrain[18]

layers can have erosion simulation applied to them, and produce realistic results. It is stated that the method can support empty pockets in the layerning scheme, thus creating caves, but their results only show heightmap style shapes (albeit naturally eroded): Figure 24.

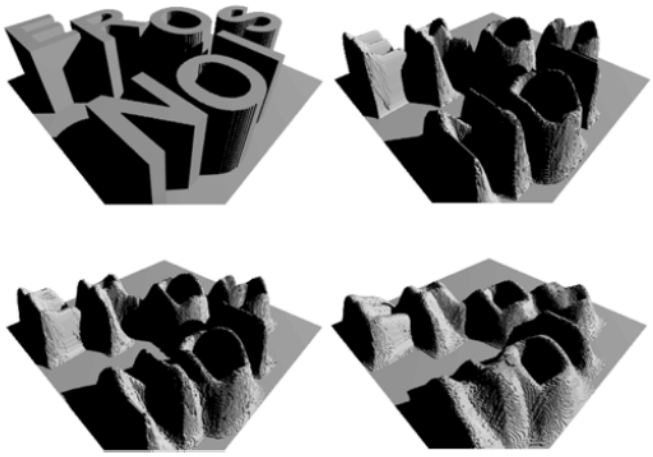


FIG. 24: Example of erosion simulation applied on letter shaped 3D models[18]

Another alternative which is documented in Joeri VanDerLei's survey[20], is Gamito and Musgrave's[19] approach to creating overhangs, hoodoos, and stratification, which do not self-intersect, without having to rely on noise. The approach relies on setting points on a mesh with an existing heightmap, and displacing them along a vector field. Apart from waves and overhangs their results included fairly impressive canyon structures with rock stratification: figure 25.

They describe how to apply these distortions directly on adaptively refined and tessellated[113] meshes. The major drawback of this approach is that warping a mesh in a non-heightmap way will invalidate all information regarding the normals of the surface, and will require the calculation of the gradient function (derivative to get the tangent, then cross product to get the normal) of whichever displacement function

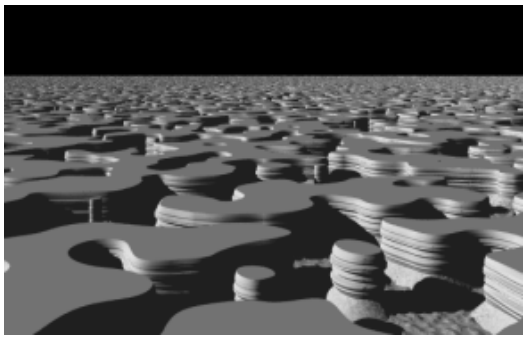


FIG. 25: Procedurally modelled canyon landscape[19]

was applied to the vertices; this process is very resource heavy and difficult to implement in real-time.

Yurovchak [44] procedurally structures caves out of primitive geometry (capsule, sphere, cone) connected in a graph. The spheres represent chambers whereas the capsules and cones represent tunnels and stalactites respectively. The interesting part of this approach, is that this primitive geometry is stored in the form of metaballs (meta-objects) which act upon a voxel structure, as opposed to stitching primitive meshes together and then distorting them with a heightmap.

A metaball [45] is a spherical energy field, typically applied to point-cloud data to exert some sort of perturbation. They are commonly used together in clusters, as multiple metaball primitives can be combined into a more advanced energy field shape.
A metaball has a radius of influence which radiates outwards from its centre, and gets weaker and weaker until it reaches its boundary of influence. Figure 26 shows these radiation boundaries in a group of metaballs. Figure 27 on the next page shows an extracted mesh, modelled from a group of metaballs.

FIG. 26: A group of three metaballs visualised as several layers of energy / influence. Combined, they create a more complex energy field shape[45]

In Yurovchak's approach, the cave-structure graph has one metaball sphere for each chamber, and a capsule for each

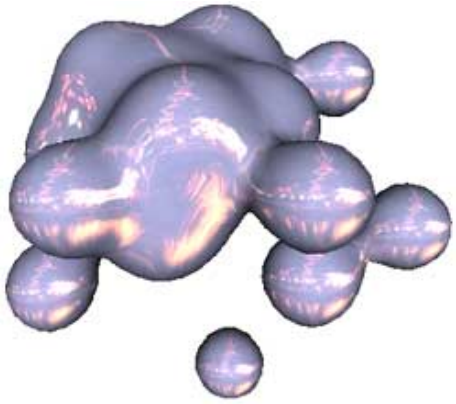


FIG. 27: Mesh extracted from voxels modelled by a group of metaballs[45]

tunnel. Instead of using a group of metaballs, he distorts the energy field of the one metaball, using layers of Perlin noise, in an attempt to recreate virtual cave features and landmarks. Figure 28 illustrates this concept from an outside view of the cave.
Stalactites are placed by picking coordinates of random numbers within each cave chamber. A Perlin noise value is then computed for each random coordinate point, and the density value of this noise is used to decide whether a stalactite will actually be spawned or not. Deciding based on Perlin noise values is important because it creates clusters of stalactites (if the noise frequency is low enough, there is a gradient of similar values next to each other). If the location is chosen, the volume function is iterated through, up and down, to find the cave ceiling and floor, and place a stalactite and stalagmite respectively.

To the finished cave mesh model (which is built from voxel data with the Marching Cubes algorithm), is added a procedural material which provides some colouring and a procedural bumpmap which simulates the rough rocky texture of a cave surface figure 29.

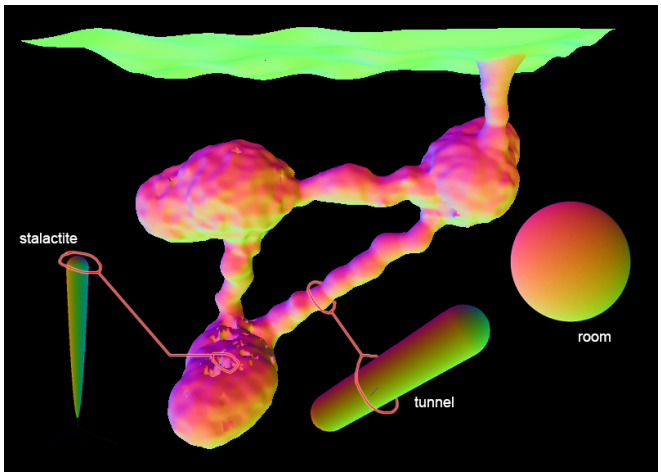


FIG. 28: Sphere, capsule and cone primitives arranged in a connected graph under the ground surface. Each primitive is distorted by noise[44]

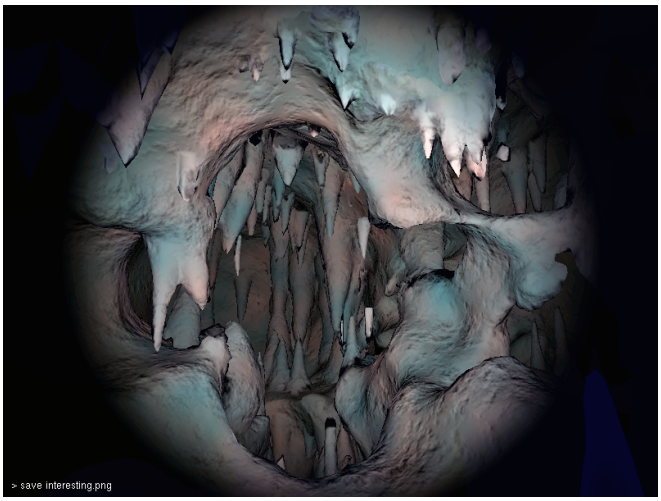


FIG. 29: Example of Yurovchak's finished cave[44]

Because Yurovchak's approach uses a single distorted meta-primitive to create a cave chamber with cave landmarks, a rather high amount of distortion is needed. This results in fairly cluttered chambers. A large amount of noise coupled with a low voxel count, results in floating geometry (a lower voxel density results in lower Marching Cubes precision and therefore thin geometry does not get extracted). This was alleviated by increasing the voxel density. Increasing voxel density however, results in a high poly mesh which would be too resource demanding to render unless dynamic LOD is implemented, or the draw distance is lowered to compensate.

Peytavie et al[23] presents a hybrid voxel terrain system, capable of arches and overhangs, which simulates erosion by attributing different material types (such as sand and rock) to voxel clusters, as well as supporting user-sculpting of the terrain. They present a compact volumetric discrete data-structure used for storing the different materials of the landscape as illustrated in figure 30 on the following page. They use a grid-based discrete model for simulating terrain stabilization, after which they reconstruct the implicit, high detail model as explained in figure 31 on the next page. Additionally, to enhance the illusion of a physically eroded terrain, they propose a procedural technique for generating rock piles using a tiling technique instead of physics simulations. The main focus of their work however, lies in the user-based sculpting of their voxel structure, for editing or constructing terrain. Once processed, their terrain is rendered and textured in real time.

Another way of generating terrain and cave structures is through the use of cellular automata. A cellular automaton in its simplest form is a grid of cells, where each cell reacts to what is happening in its neighbourhood. Each cell has set of states, as well as a set of transition rules to switch between those states[46]. When the cellular automaton is run, it checks the state of its neighbourhood to see if any transition rules have been satisfied, in which case it will switch itself to the

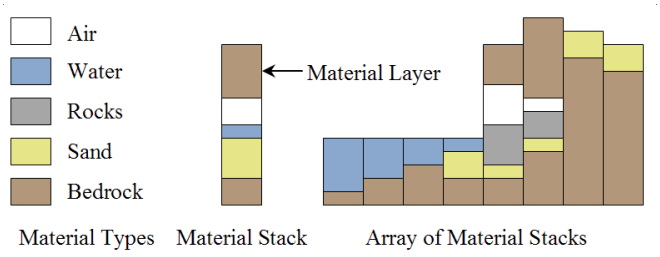


FIG. 30: Discrete voxel model which processes the ”physics” and ”settling” in the stack of various material types[23]

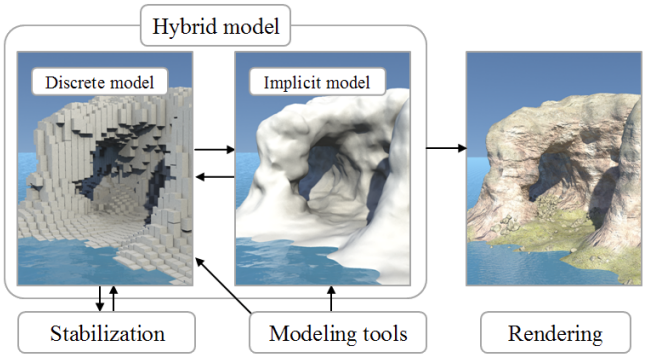


FIG. 31: Discrete voxel model in the rest of the hybrid model pipeline. ”Modelling tools” refers to user voxel sculpting tools[23]

appropriate state (which is normally either ”on” or ”off”). This simple structure can be used for a huge variety of purposes, one of which is shown in figure 32.

Cellular automata used for real-time cave generation has been explored as a method by Yannakakis et al[3], as well as Kun[4]. Both methods provide infinite procedural caves but only for 2D maps. The possibility of the extension of this method to 3D was left for future work. Extending this approach to 3D, however, exponentially increases computational requirements, as well as leaving little room for an efficient way of avoiding the inevitable floating geometry which will result from the limited view of a cellular automaton.

Cellular Automata is however not without use in the realm of 3D procedural voxel terrain. Trettner[5] presents a solution for determining voxel material types on the surfaces of voxel terrains. CA[5] is used on the surface voxels to determine whether a voxel is turned into air, grass, rock, or other materials, according to its neighbours. For example, air next to flat earth will turn into grass.

4) Natural geologic phenomena

An academic publication by Boggus and Crawfis[24] aims to create caves which realistically mimic the overall structure of real cave networks. Their results are low-scope and not particularly noteworthy, as even though they emulate acidic water erosion, they only use noise and heightmaps on 2D

[5]Cellular Automata

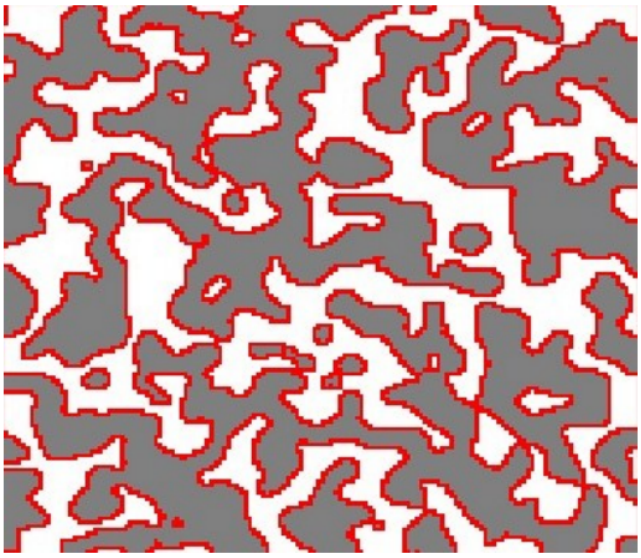


FIG. 32: A 2D cave generated with a cellular automaton applied to a grid of random points [3]

surfaces. They produce 3D cave terrain by mirroring their heightmapped surface.
However, very noteworthy is the bulk of their research which heavily analyses the natural phenomena that result in the formation of natural caves. They identify that approximately 99% of caves are formed due to expansion of large pre-existing fractures or partings, and that the dissolution process which creates the caverns can form either Phreatic passages (figure 34 on the next page), Vadose passages (figure 33), or a combination of the two (figure 35 on the next page).
Figure 36 on the following page shows the patterns that

FIG. 33: Vadose passage in Mammoth Cave National Park[24]

FIG. 34: Phreatic passage in Mammoth Cave National Park (largest cave system on the planet)[89][25]

FIG. 35: A natural combination of Phreatic and Vadose passages, in Mammoth Cave National Park, as described by Boggus and Crawfis[24], from [109]

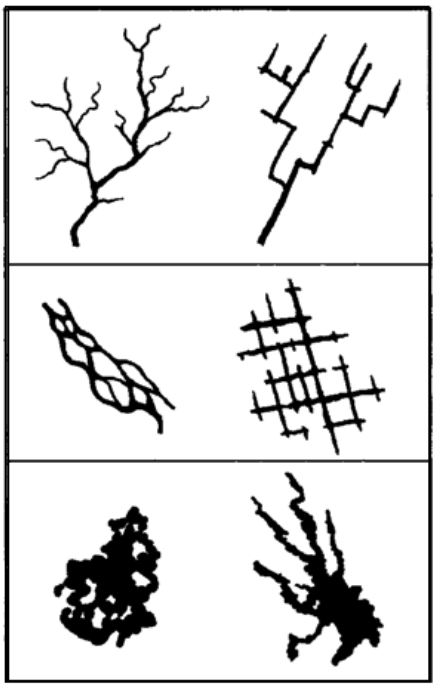


FIG. 36: Common natural cave patterns, identified and referenced by Boggus and Crawfis[24][25]

FIG. 37: Scallops on the walls of a phreatic tunnel, in the Great Orme Elephant's Cave in Wales[90]

these passages can take as the fractures expand. They range from tree-like structures, to organic tunnels, to maze-like structures. These passages provide a good base level of detail for procedural cave generation. Their shapes and the way the branches connect, depend on the rock structure and flow of water, but some of those aspects can be approximated.

The water also carves finer details, called scallops, which are spoon shaped depressions of various sizes, found on all sides of the walls. Their size depends on the velocity of the water, which flowed in the shape of eddies (vortexes) along the tunnels.Figure 37 and figure 38 on the following page show typical scallops on the walls of a Phreatic tube.

C. L-Systems

In order to create an overall structure for our cave, we chose to use L-Systems[6]. An L-system is a type of formal grammar, which at its core is a set of terminal and non-terminal symbols as well as a set of production rules. These rules governs how axioms, which are strings consisting of non-terminal symbols, can be rewritten. What sets L-systems apart from other types of formal grammars, is that the rewriting occurs in parallel, with each symbol being rewritten without knowledge of any

[6]Lindenmayer systems

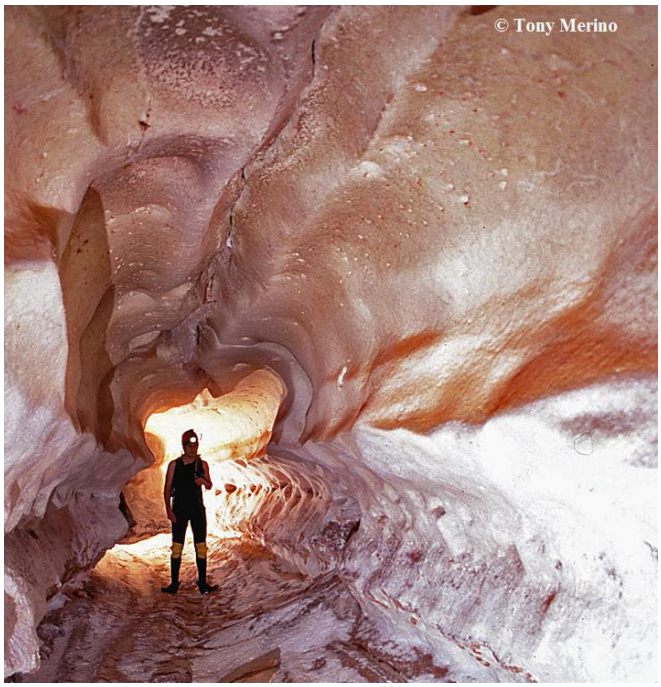


FIG. 38: Scallops[102].

FIG. 39: Romanesco broccoli showing its clear fractal structure

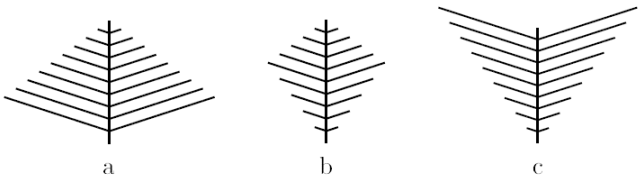


FIG. 40: Basitonic (a), mesotonic (b), and acrotonic (c) branching[31]

other rewrites[28].
Formal grammars are mainly used for validating and parsing formal languages, such as programming languages, but they have other uses as well. Within the field of PCG, however, L-systems are used to represent and generate many natural phenomena, such as streams or cracks in the side of buildings, by interpreting the rewritten string of symbols as instructions for a turtle[7].
All of these uses take advantage of two of the defining features of L-systems: that they grow organically from previous structures, and that they are inherently self-similar.

As seen above, the final expanded string of an L-system is often used as drawing instructions for a turtle to create a graphical representation of the L-system. This has one major limitation when using regular L-systems, namely that the entire structure must be drawn as a single line. The turtle cannot lift off and start in another place to create a branch. To solve this problem, bracketed L-systems are almost always used. A bracketed L-system uses brackets to signify the start and end of new branches. When an opening bracket ([) is reached by the turtle, it pushes its current position onto a stack, and keeps on drawing. When it then encounters a closing bracket it pops a position off the stack and returns to it before continuing to draw.

Despite the numerous other application of L-systems, their main use within PCG is to generate vegetation. Lindenmayer [29] initially designed L-systems in order to model the growth of plants and algae, and they have proven to work exceptionally well in that role. The way that the initial axiom of an L-system gets rewritten, by the same production rules several times, makes them inherently fractal in nature. Many plants share a similar fractal structure, as seen on figure 39, making an L-system a great tool for creating believable procedural vegetation.

A great example of this, is shown by Prusinkiewicz et al.[31], which goes over the basic theory of L-systems and show how they can be used to simulate plant growth. It especially focuses on parametric L-systems[8], which allows much more control over the shape that a given L-system is going to produce. For instance, a non-parametric L-system cannot generate the mesotonic and acrotonic branching patterns shown in figure 40, whereas a parametric L-system can[31]. The authors also present various examples of context-sensitive L-systems, such as an L-system to simulate the attack of a bug on a plant, as well as an L-system that can simulate the response of a plant to its environment, when it is, for instance, pruned.
Another example of using L-systems to generate vegetation is shown by Knutzen[30]. He presents a way to generate climbing plants by simulating their growth using tropisms[9] and representing this growth using an L-system. This L-system consists of a tip that effectively places stem segments as the System is expanded. The orientation of each new

[7]An agent that draws a continuous line, using the L-system symbols as instructions

[8]An L-system where each symbol is followed by one or more parameters, allowing for more fine control

[9]Various parameters affecting the growth of the plant, such as wanting to grow towards light, or having to work against gravity.

stem segment that gets added to the plant is governed by the various tropisms, and combined with the parallel nature of the L-system, it simulates the parallel growth of a climbing plant, as it branches and grows.

Besides vegetation there are many other uses for L-systems within the field of PCG. Parish and Müller[32] presents a way to use extended L-systems to model an entire city. Their generation method consists of three major parts. The first part is the L-system which generates the overall structure of the network, the second part is a set of global goals, which sets the parameters of the roads generated by the first step, and the third part which is a set of local constraints, which fits the parameters determined in step two to the local environment of the road. After the road network has been modeled, buildings are generated with the use of L-systems as well. The axiom of this L-system describes the bounding box of the building, which allows the generation of lower detail versions of a building, by simply using an earlier generation.
The problem of structuring a road network is actually similar to the problem structuring a cave, as even though the road network is in 2D and the cave network is in 3D, the problem of creating a well-formed network of connections is the same. However, whereas a road network needs to be structured in a logical fashion, a cave, as seen on figure 36 on page 14, can have a much more dynamic and random structure. This means that an approach similar to the one presented by Parish and Müller would be less suited to caves than it is to generated logically structured road networks.

Another interesting use of L-systems is shown by Danks et al.[33], and describes the use of stochastic L-systems to simulate the folding of proteins. To achieve this, a parametric and stochastic L-system is used. At each rewriting step, the probability of each of seven different chains of amino acids are calculated from the state of the environment, and the L-system is expanded based on those probabilities. The actual structure of the generated protein is based on the parameters of the amino acids, which consists of two torsion angles.
The structures generated with this approach is shown on figure 41, and actually bears a resemblance to 3D cave structures, as well as the cave structures shown on figure 36 on page 14. In our approach we tried to use our own stochastic L-system to replicate that twisting structure for our caves.

Besides being used to generate new structures from an existing ruleset, L-systems can also be used to represent existing structures. To accomplish this goal, St'ava et al.[73] presents a way to generate parametric context-free L-systems from a 2D vector image. Atomic components of this vector image, such as curves or lines, are converted to a terminal symbol in an L-system alphabet, with the transformations needed to go from one component to the next stored as 4D transformations. These transformations are clustered and analyzed to allow for extraction of L-system rules that generate the final L-system along with the alphabet.
An extracted L-system like this, is an interesting way to store

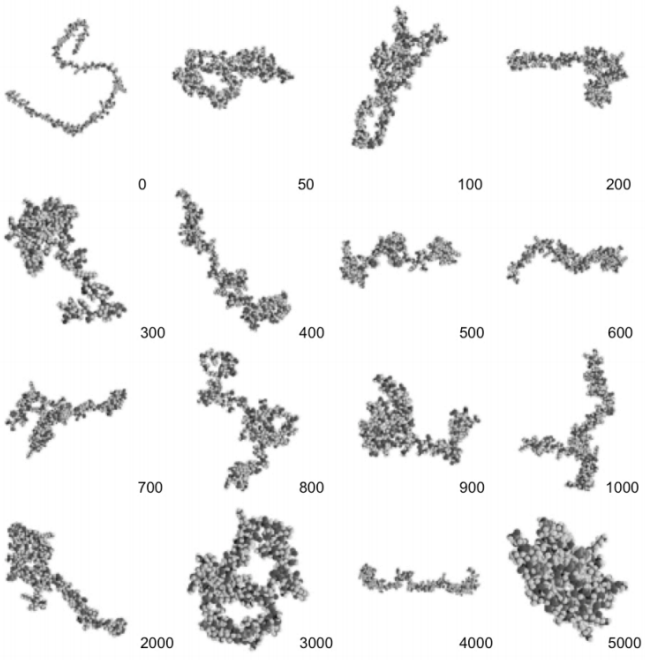


FIG. 41: Proteins folded with an L-system[33]

an image, as it allows the image to be edited in new ways. The parameters of the L-system can be changed to perform various operations on the image such as randomization. By doing this, it is possible to generate many similar, but still different, images from a single source image. This could, for instance, be used to generate many different permutations of an original human created image for use as procedurally generated content.

While modelling structures as L-systems opens up a lot of possibilities for interesting procedural content, the current method presented by St'ava et al. is not fit for use as an actual tool yet. Firstly, while being able to model 2D structures as L-systems could be useful for some applications, expanding it to 3D would be needed for it to be used for a wide variety of purposes. Secondly, the approach shown can take up to 20 minutes to encode an image as an L-system which is not ideal in an actual production environment.
Never the less, even with these downsides, creation of L-systems from existing structures is definitely an exciting prospect for PCG in general.

D. Voxel data management

A voxel is a unit, an atom, of data as part of a grid in 3D space. It is analogous to a texel, which is a texture point on a 2D grid, and a pixel which is a point on the screen. The simplest voxel holds world position coordinates and colour information. Depending on memory constraints and the needs of the developer, a voxel can hold various other properties such as material type (ground, air, water, vegetation), more complex surface information, such as the voxel's normal

vector, and even velocity.

1) Spatial Object Data Structures

In order to define and manage shapes using voxels, they need to be stored in an efficient data structure. Storing volumetric data for an entire world in a 3D grid is infeasible when it comes to memory usage and voxel searching speed. Therefore, tree data structures are preferred, due to their ability to recursively subdivide the volume of data, each node representing the bounding volume of all of its children. This way, voxel data can be sampled at different levels of detail, with the most amount of information fetched from the spatial partition which we are currently most interested in (ie the part closest to the camera), and also generally the parts of the model which are most detailed compared to others (others such as flat surfaces). Samet [47] analyzes the main voxel data structures available. The state of the art in the idustry are the Point Region Octrees, and the KD-trees.

A classic octree has its root node encompass the entire volume of the voxel data. Each subsequent node holds eight nodes (octants) which are equal size cubes inside the original volume, partitioning the data within their volumes. The subdivision is recursive and encloses finer and finer detail. The position itself of an element in the octree is enough information to know its spatial location in the world (since the subvolume would be a division of the parent volume). Figure 42 illustrates a 2D representation of an octree (a quad tree), with its partitioning and the associated tree structure, resulting the placement of the jade dot in the spot shown(the rest of the boxes would be considered empty).

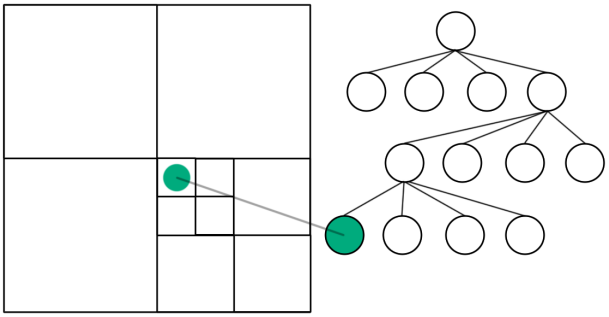


FIG. 42: The subdivision of a quadtree and its associated tree structure.

Point region octrees are more useful for terrain than classic octrees. Unlike classic octrees, which always split the volume through its centre, into equal subvolumes, a point region octree allows for splitting at different (not symmetrical) positions. Each node explicitly stores a 3D point representing this node's splitting point, which makes it use a little bit more overhead memory, but this is more than compensated for overall. This point also represents one of the corners for each of the four children of the current node. So in the case of voxel terrain, which would only encompass for example, a fraction of the lower side of the global bounding volume, it would be less wasteful to lower the splitting point. It can be seen in figure 43, that less subdivisions are required, and less squares are wasted with empty space.

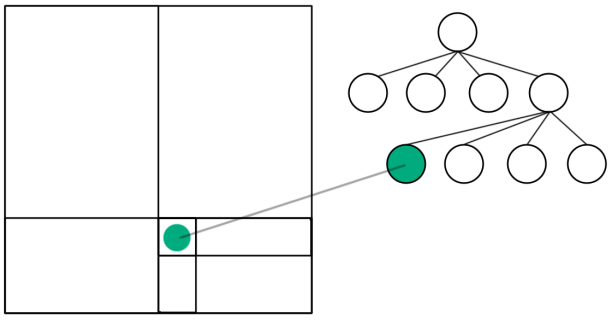


FIG. 43: The subdivision of a point region quadtree and its associated tree structure. Note the sparser tree structure.

Figure 44 shows a 3D representation of an octree.

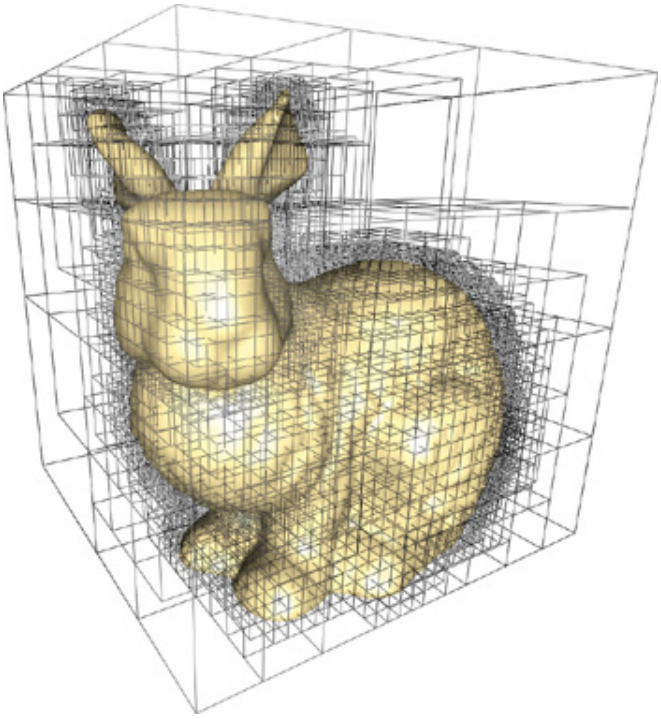


FIG. 44: Rabbit voxel model with overlay illustration of a 3D octree[48]

KD-Trees [51] are a type of binary search tree, generalised for K-Dimensions. Every non-leaf node is a k-dimensional point which can be considered a hyperplane which splits the k-dimensional space into two parts. The splitting axis position can be arbitrary (selected according to needs). Otherwise, to save on memory overhead related to storing the axis position, the splitting axis can be a function of the node's height in the tree.

In simpler, and more practical terms (and related to 3D voxel volume management), a KD-tree will repeatedly pick

an axis and slice the volume into two parts: figure 45. If we were to store a single point in a 2D KD structure, we would split along the X and Y axes as in figure 46.

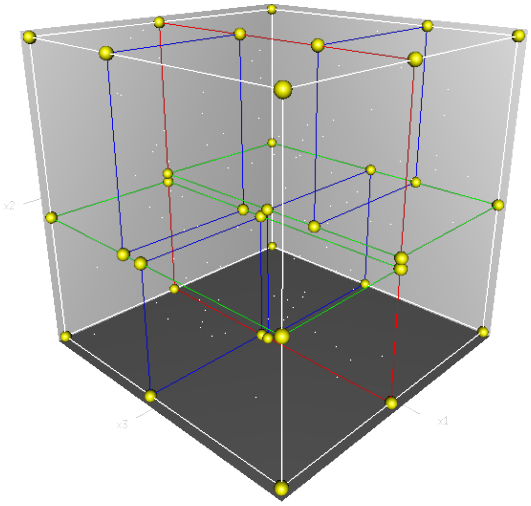


FIG. 45: The volume is first split along the x axis (Red), then each sub-cell is split along the Y axis (Green), then the resulting sub-cells along the Z axis (Blue)

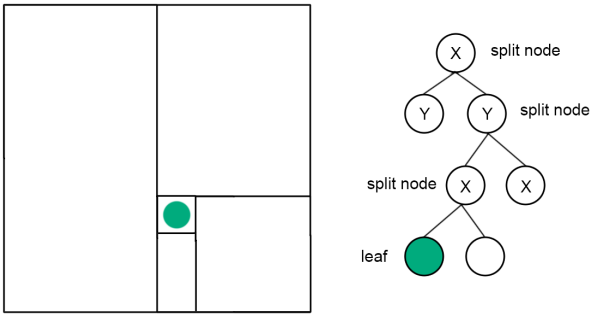


FIG. 46: First, the x axis is split in half, then y, then x again, the target leaf node is reached.

The important selling point of KD-Trees, is that even though they are k-dimensional, they are still searched like binary trees in $O(logn)$. Since each axis is split in two, one side will have spatial coordinates of that axis smaller than the value of the splitting point, and the other half will have the higher values. Therefore, searching a KD tree can be thought of as continuously splitting space in half, and then recursively continuing only in the half that contains values of the range in question.

There is no clear winner among these tree data structures in terms of which to pick for voxel data management. Each has its benefits and drawbacks, and depending on implementation and needs, can be optimised very well on a deep technical level. For example, for raymarching and traversing (Revelles et al [49]) octrees can be in certain cases made even faster to search than the $O(logn)$ speed of the KD-tree; there are also memory and cache optimisations (Laine and Karras [50]) etc. It is however beyond the scope of this paper to delve any deeper into this matter. Our choice for procedural terrain (and caves) would be to employ the use of a Point Region Octree, as it would be sparser and better adapted for the tubular nature of caves.

2) General-purpose computing on Graphics Processing Units (GPGPU)

GPUs were originally constructed to process a lot of math operations on individual vertices in parallel. In recent years there has been a push to take advantage of the GPU's high computation and data throughput capabilities for more than the standard geometry rendering pipeline. GPGPU [94] computing languages were developed to provide a functionally complete set of operations to be performed on arbitrary bits of data (instead of specifically on triangles for example). This allowed custom programs to be written and run entirely on the GPU, and it is extremely useful for n-body simulations, cryptocurrency mining, voxel data processing and any other operation that benefits from massive parallelization.

In order to implement the tree structures (such as the octree) on the GPU, 3D textures must be used as hosting data structures [48]. The octree, as a dynamic data structure, cannot simply be saved in the classic 1D buffer that the GPU typically expects to run its shader code on in parallel. The technical side of GPU architecture dictates that the only place where it is possible for us to store a changing data structure is on a texture. The GPU is wired to fetch data from a texture and then (for the sake of simplicity we can say that it) turns it into the aforementioned 1D array to which the shader functions are applied. Similarly, the GPU can write and save data to a texture. This means that the only place to store and manage an octree is in a 3D Texture.

The major players in graphics processing, namely Nvidia, Microsoft and AMD all have GPGPU solutions: CUDA [112], Direct Compute [111], OpenCL [114] and Mantle [115].

CUDA is a C and C++ toolkit which includes a compiler and set of tools for Nvidia GPUs. It is a proprietary toolkit specialized exclusively for Nvidia graphics cards. Therefore it is very easy to use and debug, but only works on their cards.
OpenCL is the dominant open general-purpose GPU computing language, for C++, but being open, it is very hard to set up and the syntax is more difficult. Whereas in CUDA the kernel functions can be called just as any C++ functions and there are plenty of helper functions in the API, OpenCL appears more archaic; the correct driver must be selected and compiled for the specific graphics card, and there is not much debugging support. The support in general is lacking except for AMD graphics cards.
At the end of 2013, AMD has announced a new universal, very flexible and very low level API called Mantle, however documentation and toolkits have not yet been publicly released (as of March 2014).

DirectX has trended over the years towards increasing the

flexibility of their shader programming model, and Microsoft's Direct Compute shaders are now constructed to offer the ease of classic shader programming on arbitrary data. Compute shaders work in HLSL (High Level Shader Language, one of the most common shader languages) therefore they are very easy to write and can be compiled just like any DirectX HLSL shader, for Windows and all graphics cards which support DirectX11 or higher. The obvious drawback is that DirectX11 only works on the Windows platform, and on newer GPU's, but even so, it covers a wider market than Nvidia's CUDA, for instance. Another drawback is that it is not as flexible as OpenCL or Mantle as far as low level memory management and data structures go, since it is a higher level API designed for multiple brands of GPUs.

The most important selling point of Direct Compute however, is that it was recently integrated in the Unity3D game engine [96]. Unity is a very flexible modular game development engine specifically constructed for small indie developers. Unity's C# Direct Compute wrapper allows for easy integration and access of DX11 compute buffers right from C#, among the rest of the game code. Unity is also very PCG and customisation friendly, as opposed to more rigid AAA engines such as the Unreal Development Kit [95]. Finally, developing directly in C++ without a game engine would be needlessly out of scope.

E. Voxel Rendering

1) Spatial Object Raytracing

Ray tracing (Levoy 1990 [54]) is a rendering technique which relies on a ray being cast from each of the screen's pixels, to the game world, and identifying which object (or voxel) should be rendered for that pixel. More than that, with this technique, each light ray travels through the world recursively, and back-traces all (up to a certain recursion limit) the necessary objects that would make up the resulting pixel color (ie objects reflected in a surface, refraction, medium density, scattering etc).
Raytracing would provide an accurate physical simulation of light physics, and evidently beautiful results, instead of all the tricks and approximations that rasterization uses, along with all the difficulties and limitations associated with it (Carmack 2011 [56], Carmack 2013 [57]).

The main issue for raytracing in real-time for video games is performance. As Carmack points out [55], there are many other elements that must be covered, apart from just replacing rasterization with standard raytracing. Texture sampling, shader program invocation, color-object blending etc. would also have to be recreated for the raytracing renderer and would add an overhead on a per-object level as well as per-ray level. Therefore, even if the hardware to render billions of recursive rays per second would be commonly available, it would still not be enough to fully implement an equivalent solution to the entire rasterization forward rendering pipeline. Carmack does believe that raytracing will win eventually, however. For now though, raytracing is used in non-real-time scenarios such as rendering for the movie industry (Pixar [98]).

A simpler and older version of raytracing exists, namely raycasting (Roth 1982 [58]), which simply casts rays only to retrieve the first object (voxel) it encounters, and to trace the light coming from the light source and hitting that point. This would result in a much poorer representation than the standard rasterization model and all of its associated rendering techniques.

2) Spatial Object Isosurface Extraction

Since raytracing is not yet a feasible prospect, a lot of work has been put into defining and refining isosurface extraction algorithms, which means extracting the surface between solid space and "air" and transforming it into a polygon mesh to be rendered in a traditional fashion.

The following sources survey the history of mesh building approaches and isosurface extraction techniques, and provide benchmarks: Lysenko [52], Tristam [53].

One of the first, and easiest to use techniques is the Marching Cubes algorithm (Bourke [1], Lysenko [52], GPU Gems 3 [17]), first published in 1987 by William E. Lorensen and Harvey E. Cline [2]. Even though parts of the implementation are not trivial, the concept is simple: it samples the 3D density data grid (or the density function along a 3D grid), making cubes out of groups of 8 adjacent grid points. It then checks if any of the corners containing density data of each cube are on the opposite side of the density threshold than other corners in the same cube (most commonly the density data is between -1 and 1, and the threshold is 0). If so, it means that the surface edge intersects this cube, therefore the intersection points of the surface to the cube will be identified through weight based interpolation, and vertices will be created.

The difficult part of Marching Cubes (and the part which makes it fast), is the way the algorithm figures out which surface(s) to create between these surface intersection vertices on the cube. There are 256 cases, the first being an empty cube. The rest of the cases determine how many triangles to output (between one and five) and how to stitch them (figure 47 on the next page), based on the sign of the function at the 8 vertices of the cube (sign, or threshold comparison, if threshold is not zero). To solve this, the algorithm relies on two lookup tables (GPU Gems 3, 1.2.2 [17]). The first one stores the 256 possibilities of how many polygons to create for that case, and the second one (which is much larger) stores for each case, information on which vertices to connect in order to create the required triangles (up to five). Some of these cases however are ambiguous (Nielson and Hamann [61]), having more than one solution to how to generate triangles (one example is figure 48 on the following page). This in turn, results in the Marching Cubes meshes

not being manifold meshes [60]. The Manifold conditions are that no more than two faces can share a common edge, and the faces sharing one vertex can only form the shape of a closed or an open fan.

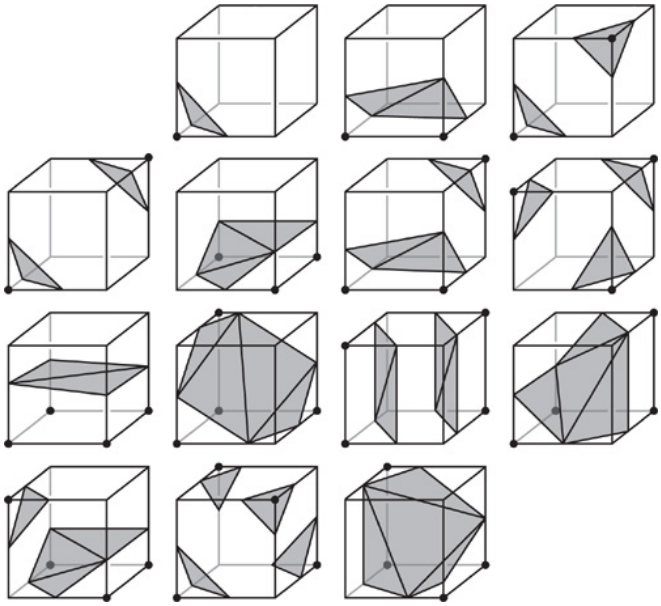


FIG. 47: Illustration of the basic cases resulting from the application of marching cubes[17]

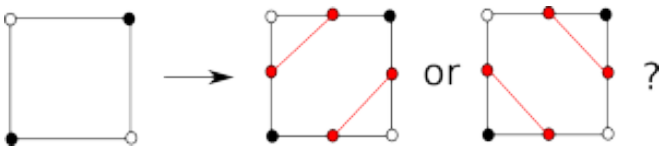


FIG. 48: Illustration of an ambiguity case, visualising a cube in 2D [52]

Marching Cubes is reasonably fast, produces reasonably efficient meshes and is free to use. However, it is one of the least accurate solutions, especially at low voxel densities. If for example a corner from the density data happens to end up inside the volume of the cube instead of on the surface (which is most likely) as in figure 49, it will be cut off: figure 50, since the algorithm only draws vertices on the surface of the cube.

Marching Tetrahedra (Doi and Koide 1991 [59]) was developed as an alternative to Marching Cubes in an attempt to overcome its perceivable shortcomings and its manifold problems. The algorithm was not patented and is conceptually similar to Marching Cubes. Six tetrahedra are used instead of each Marching Cubes cube for the grid division. Because of the simpler tetrahedron shape, Marching Tetrahedra only has 16 cases instead of 256 (2^4 (4 faces) instead of 2^8 (8 faces on a cube)). It also does not have ambiguous topology and always produces manifold surfaces.

The downside of Marching Tetrahedra is that because of the smaller volume of the tetrahedron, the resulting meshes are typically four times higher resolution than the ones made by marching cubes. This is a major drawback for real-time

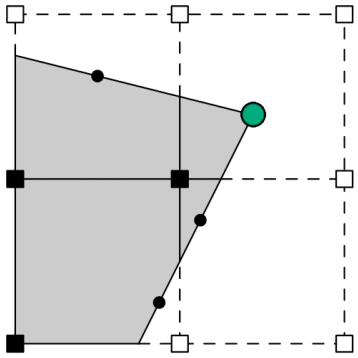


FIG. 49: 2D representation of a set of four cubes from Marching cubes, encompassing an object with a sharp corner (the jade dot in the top-right cube) [65]

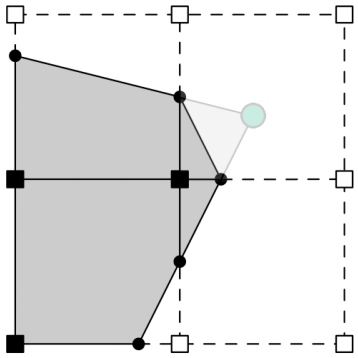


FIG. 50: Illustration of the loss of detail after the Marching Cubes isosurface extraction[65]

generation and even more so on real-time rendering of the resulting mesh. Therefore one would normally use tetrahedra over cubes in games, only if having a manifold mesh is of utmost importance: for example if triangles are required to share their vertices, and if tessellating a non-manifold mesh a certain way produces undesirable visible artifacts.

As research in isosurface extraction continued, a new general class of methods has emerged which has proven most effective: the so called ”dual” grid schemes. The duality means that a secondary grid is generated from the first grid in which the density values reside.

With primal methods such as Marching Cubes, finding the vertices on the cube between which to draw the mesh was fairly easy, and the hard part was determining and handling the many cases of how to stitch the triangles together. Dual methods take a somewhat opposite approach, as creating the rules of accurately assigning vertices to the cubes is the difficult part, after which there are no more ambiguities.

The first of the dual methods is Sarah Frisken Gibson’s Constrained Elastic Surface Nets [62]. Similar to Marching

Cubes, Surface Nets creates a cubic grid, grouping every 8 voxels into one cube. Surface Nets checks if the surface intersects somehow with the current cube, by comparing the values in the corners, and if so, places a Surface Net Node in the centre of that cube. Each Surface Net Node is linked to as many as six neighbouring net nodes (left, right, top, down, front, back) (depending on how many exist). Next, the algorithm smooths the Surface Net Nodes in multiple passes. Each smoothing pass moves each Surface Net Node into the average position of its neighbouring net nodes (of course, the movement is applied to the nodes only after all new positions have been calculated). Gibson has defined that a net node must never move outside of its parent cube as a result of smoothing, this being the limit to the elasticity in Elastic Surface Nets.

The idea is a bit slow in practice (slower than Marching Cubes), due to the multiple smoothing passes required to produce a visually good result (according to Gibson, usually between 10 and 100 passes). An alternative is to use an approximation or some sort of heuristic for where to initially place the Surface Net Node inside each cube to try to improve performance. A naive approach would be to place the net node at the centre of mass of the cube, based on the weights of the cube corners. For a similar run-time, the results of Surface Nets are usually less accurate than Marching Cubes, albeit using less triangles: figure 51.

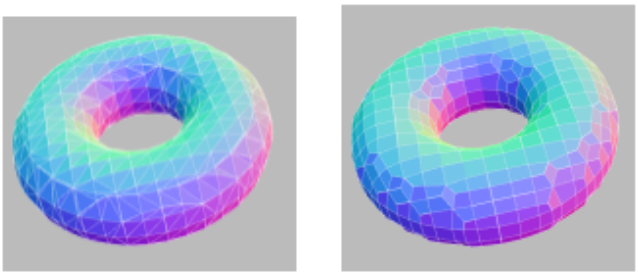


FIG. 51: Marching Cubes on the left, versus a similarly fast naive Surface Nets on the right[52]

The methods presented so far are not able to reproduce sharp accurate detail from a sample function. One algorithm which very accurately approximates these details is the octree-based solution of Dual Contouring of Hermite Data (Ju et al 2004 [65], [69], [68]). This technique depends on the use of ”Hermite Data”, which essentially means that a function (or the voxel data) holds, in addition to position and noise (or function) values, the partial derivatives for each point. Moreover, in the case of editable voxel terrain, any 3D tool altering (sculpting) the data would also have to provide partial derivative information to the surface it leaves behind, because otherwise calculating it would be too slow for real-time. The partial derivatives of x, y and z are what makes the gradient (or normal vector) of the point (the derivative of a function finds the tangent to any surface point of that function).

Similar to Surface Nets (and dual methods), Dual Contouring uses the same principle of a grid of cubes (except on an octree structure), and spawns a Feature Node inside each cube which intersects the isosurface somewhere. Like in previous methods, the weights in the corners of the cube are used to determine exactly where the isosurface intersects with the surface of the cube. Furthermore, a normal for this intersection point is interpolated from the normals of the cube corners (figure 52). Knowing these normals, it can be relatively accurately predicted where the intersections will converge inside the cube, namely at the smallest quadratic distance to all tangent planes of the previously defined cube intersection points. Their proposed solution to find the point closest to all the intersections is to minimise their Quadratic Error Function (Ju et al, equation 1 [65]) to approximate where the Feature Node should be placed relative to the tangent planes to best match the surface of the object. (figure 53). The cube intersection points are discarded, and the Feature Nodes are connected to form an accurately approximated mesh: figure 54 on the following page.

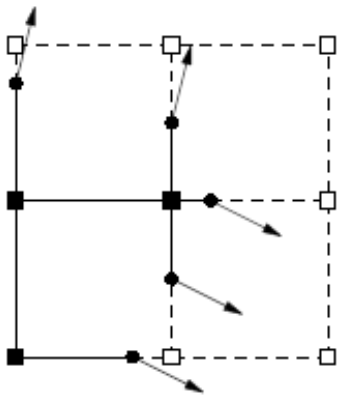


FIG. 52: 2D representation of a group of four cubes of 4 voxels each, as part of the Dual Contouring scheme. The round black dots represent the points at which the isosurface intersects the cube surface. The arrows represent the calculated normals of those isosurface points[65]

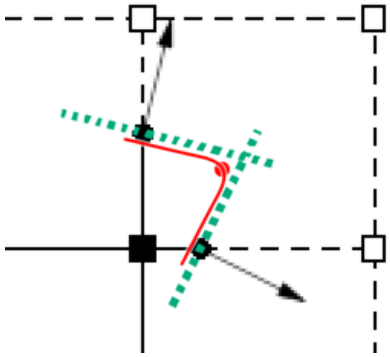


FIG. 53: 2D illustration of an example of a point picked by the Quadratic Error Function (red point). Dotted lines represent the tangents.

This method of using the QEF provides a very accurate Feature Node position, without the need for an explicit test

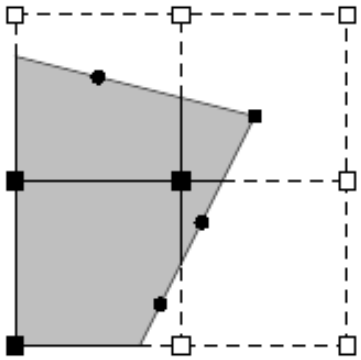


FIG. 54: The points inside the cubes are the Feature Nodes[65]

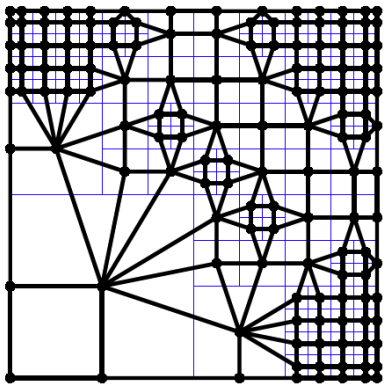


FIG. 55: Connectivity of the dual grid (thick lines) on top of the octree primal grid (thin blue lines). 2D representation[64]

for features, like the elastic smoothing done in multiple passes for the Elastic Surface Nets dual method. Of course, it comes at the price of requiring Hermite data.

Another advantage of Dual Contouring (as opposed to Marching Cubes) is that it can handle Levels Of Detail well [67], without cracks at the LOD transition areas. Assuming the LOD resolutions differ by a power of two, and that the voxels are aligned in an octree structure, it is possible to simply transition the cube-corner sampling from the finer grid to the coarser neighbouring grid. The transition will go from the normal of a high resolution voxel to the normal of a coarse voxel.

There have also been attempts to improve Marching Cubes, for example by smoothing its meshes with the principle of Elastic Surface Nets (Holmstrom [63]), or the Extended Marching Cubes [66] algorithm which is similar to Dual Contouring but less efficient according to Ju et al [65]. The best approach however is that of Dual Marching Cubes (Schaefer and Warren 2004 [64]), which aims to improve the dual methods.

Dual Marching Cubes has the potential to be much sparser than the other methods. Similar to Dual Contouring, its Feature Node dual grid, is able to transition from any octree cube to any other, regardless of the difference in scale or coarseness: figure 55. However, since Dual Marching Cubes uses an additional surface approximation step after the calculation of the Feature Nodes, it can afford to use a sparser octree.

This results in the ability to accurately represent thin geometry (or sharp details) without the use of an unnecessarily fine grid which would waste triangles: figure 56.

Similar to Dual Contouring, this algorithm computes the minimum quadratic error to place each Feature Node somewhere near the tip of the corresponding intersecting tangent planes (figure 53 on the previous page). The result of this stage is still a surface of one feature point (thus one vertex) per cell, and unlike classic Marching Cubes, the cells

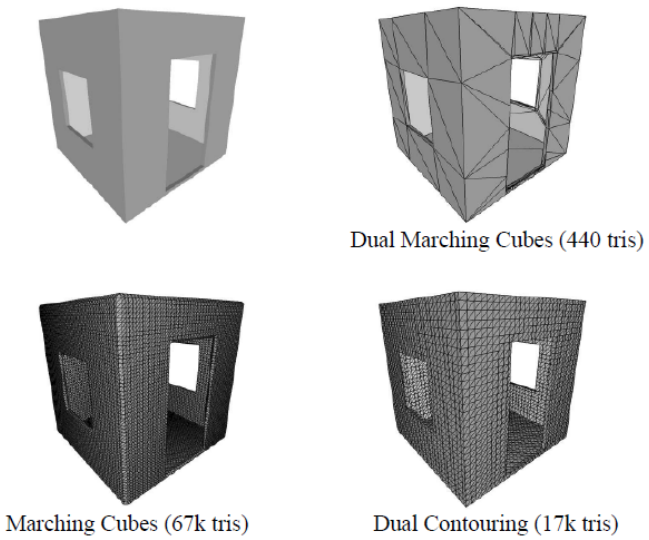


FIG. 56: Illustration of a 3D room made out of thin primitives. The accuracy and polycount of reproducing that geometry is compared between methods[64]

are of varying sizes, with high division only where detail is necessary, due to the octree structure. An example of a resulting dual grid of Feature Nodes is in figure 57 on the following page.

The clever part of Dual Marching Cubes is in an additional step, where the standard Marching Cubes algorithm is applied to the dual grid of Feature Nodes. Marching cubes was designed to run only on axis-aligned, regular cubic grids, whereas the dual grid has arbitrarily positioned Feature Nodes. Schaefer and Warren realised that the dual grid cells, which can be either a cube, a cuboid, or a ”degenerate cube” where some vertices are shared (i.e. is a tetrahedal shape), are topologically equivalent to the regular cubic grid with regards to how the algorithm is used. This means that it can be ”pretended” that we have all eight corners of an axis aligned cube for every dual grid cell, and use the values stored in each cell’s Feature Nodes to compute edge intersection points to the surface. This effectively obtains an even finer approximation of the original shape, since the Feature Nodes

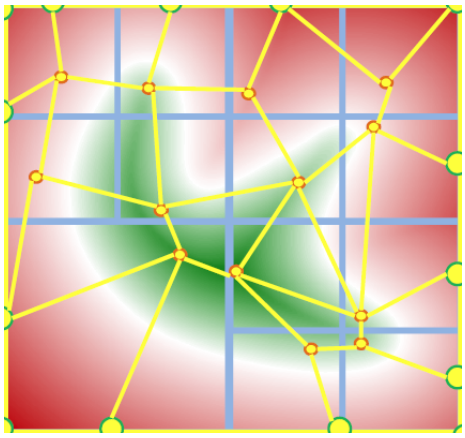


FIG. 57: Grid of Feature Nodes on top of the base octree grid [72]

were already well approximated to the surface (potentially as well approximated as Dual Contouring): figure 58.

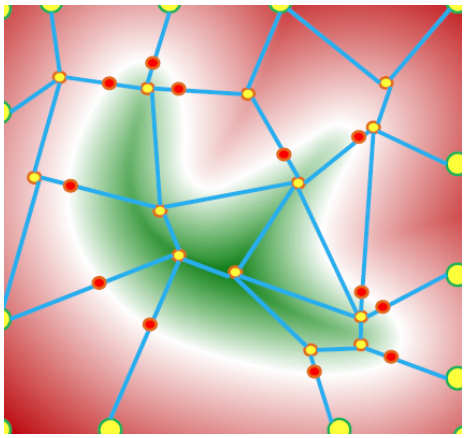


FIG. 58: Classic Marching Cubes sets grid-to-surface intersection points (red dots) on the dual grid (blue)[72]

As far as performance is concerned, according to Schaefer and Warren, it takes no longer than contouring primal grids. This is mostly due to the fact that their dual method is not only based on an octree but the octree can afford to be sparser than it has to be in other dual methods. They do not offer a GPU implementation of their algorithm, therefore they provide no data on real-world interactive procedural environments.

There are additional isosurface extraction algorithms, as well as additional variants of the aforementioned ones, but these are the the most relevant and ifluential.
To conclude the review of the isosurface extraction methods, we must attempt to choose the best option. It appears that we are left with one big choice: either use primal methods, or dual methods. Using a dual method without Hermite data, would yield minecraft-like cubes (a feature node in the centre of each cube). Therefore it boils down to: do we need (and can we provide) Hermite data? Choosing primal Marching Cubes would allow more freedom of experimentation such as attempting to generate cave environments by running Cellular Automata on the voxel data, or otherwise distorting the density values without having to worry about maintaining point tangent (derivative) information. On the other hand, managing to implement Dual Marching Cubes on the GPU (or even Dual Contouring) would yield a very high quality result. Games such as Everquest Next [88], and even Minecraft [81], use dual methods.

3) Procedural Shading and Solid Texturing

Procedural texturing, and solid texture synthesis (Forsyth [43], Kopf et al [42]) are methods of particular importance to voxel carving. Since solid textures are produced as (typically based on noise, thus infinite) world positioned volumetric data, they can provide texture information for the entire volume bounding the object to be textured. In the case of voxels, this is very useful, as it essentially means that any carving of the object will reveal a coherent continuation of the material in question (example: figure 59).

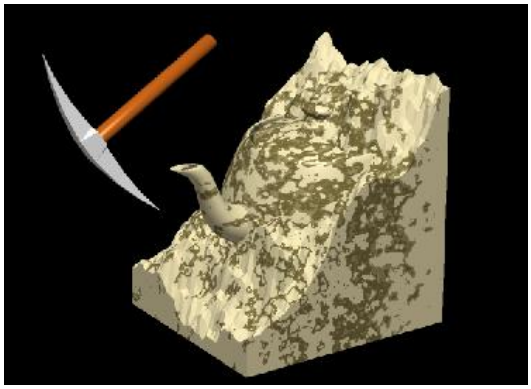


FIG. 59: Illustration exemplifying the nature of 3D procedural solid texturing[39]

GPU gems 3 [17] suggests a set of methods of creating procedural or hybrid textures that enhance the appeal and believability of procedurally generated landscapes.

The main issue discussed is how to map textures to procedural (arbitrary) meshes in the first place, considering that a procedural mesh can never have UV (texture) coordinates. Artists typically unwrap a complex 3D model onto a 2D plane and map texture coordinates to it using various 3D modelling software. Applying a simple texture projection would result in undesired distortions, as seen on figure 60 on the next page. The standard workaround in lieu of this is to use triplanar projection, seen on figure 61 on the following page. Triplanar projection uses a separate seamlessly tileable texture, projected onto each plane (x, y and z). The transition between one texture and another is handled by the weights (angle) of the vetex normals, ie ”is this region mostly flat? then sample more from the Y texture than from X and Z”. The three sampled colours are added together and this achieves a

transparent blending between the three textures at transitional angles.

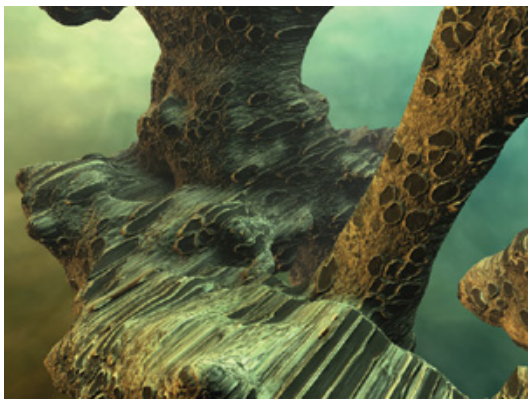


FIG. 60: Simple texture projection over arbitrary terrain; shows excessive stretching[17]

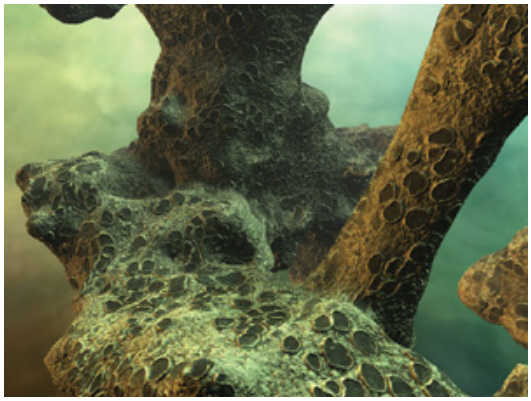


FIG. 61: Triplanar texture projection over arbitrary terrain[17]

An approach to create a hybrid procedural texture is to slowly sample a striation texture over the Y coordinate, to obtain transitions between one texture/color and another: figure 62. Altitude (and more generally, world-position) based lookups can provide a natural sense of progression in the environment, as for example the texture sampling would reach, for instance, a muddier part of the rock texture as you descend deeper down into the world.

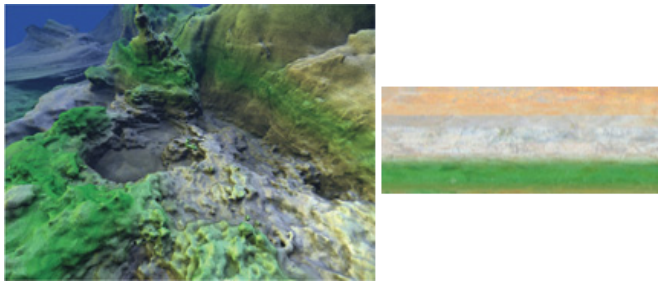


FIG. 62: The terrain on the left samples from the "Variable Climate Lookup Texture" on the right[17]

An entirely procedural texture can be created by, for example, computing a sine wave calculation and multiplying it (perturbing it) by a 3D Simplex noise lookup based on vertex world position. This can create a marble-like texture strip along the terrain, as shown in figure 63. The author suggests combining the two approaches for more varied and highest detailed results: texture projection for details, and procedural alterations for variation.

The book also contains technical guidance with regards to shader programming: approximating tangents by swizzling the vertex normals (rearranging the normal's coordinates and flipping one of their signs), and perturbing the normals according to triplanar projected bumpmap textures to obtain bumpmapping.

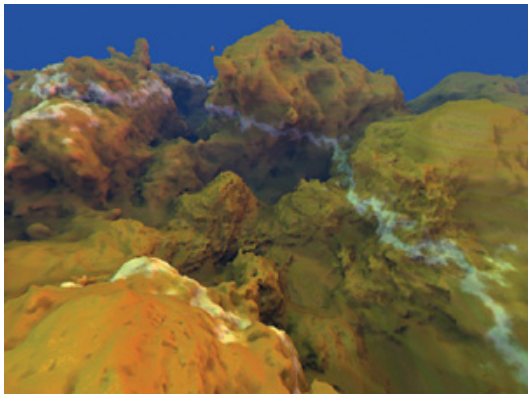


FIG. 63: The noisy white lines are the procedural marble texture[17]

III. Methodology

This section shows the framework and pipeline we built to obtain fully rendered 3D caves. It details the components of this pipeline along with an overview of the iterations that led to their final form, our decisions in relation to, and the comparisons with, the literature discussed in the above section.

A. The Framework and Pipeline

1) The Engine

The solution is built within the Unity3D game development engine, mainly making use of its graphics APIs and modular C# scripting structure.
It does not come with a lot of features out of the box tailored to a specific genre of games, and this along with the fact that it is very customisable, allows it to be used for rapid development of every purpose. These properties give it a distinct advantage over an engine such as the Unreal engine[95], which has evolved to specifically support first person shooters, in a AAA pipeline with little freedom to negotiate procedural content. Implementing functionality that does not fit into this narrow scope is time consuming and problematic, as would also be attempting to write everything from scratch without a game engine at all.
Unity3D, does supply a solid and customisable base for the rendering pipeline, and the game world representation needed for us to implement our solution without worrying about these aspects. It also allows us to interact with the GPU with relative ease as described in section II-D.2 on page 18.

2) The Cave Generation Pipeline

The overall structure of the cave generation pipeline is shown on figure 64 on the following page. It consists of 3 major components as follows:

1) A structural component, which uses an L-system to generate a set of structural points.
2) A tunnel generation component, which generates the actual cave tunnels around the structural points.
3) A rendering component which extracts a mesh from the voxel data, and applies materials and shaders.

To elaborate, the first step is used to create the overall structure of the cave. This structure is generated with an L-system (described in section III-B). The structural points themselves are generated when the L-system string is interpreted as instructions for a turtle, which generates structural points as it traverses the generated L-system string. Depending on the parameters given to the L-system, some branches may be pruned to fit a certain worldspace volume, and/or connected together to reduce or remove the number of dead ends found in the cave. This is achieved by generating a line of structural points, curved by noise, between them.

The structural points are then distributed into a tree structure of tiles. Each tile represents a cubic volume of space, and is a fragment of the entire volume described by the overall L-system structure. Each tile holds a list of structural points (L-system points).

After all the structural points have been distributed, this tree of tiles is iterated through, and each tile is processed one after another: first to generate a voxel representation of a cave segment, and then to generate a mesh to display and save to the game world. This processing is done through the ”Cave Generation States”, and represents steps two and three of the three major components.

As seen on figure 64 on the following page, the voxel representation of the cave is generated from chunks of structural points (tiles). Our tile processor is a multi-pass, multi-layer system. It consists of a set of states, each of which is responsible for updating the GPU buffers, assembling voxel data from the current tile information (structural points), and generating the shape of the tunnels through the use of a metaball which moves along the structural points. Additionally, there is a layer for the distribution and generation of various features for each tile, such as stalactites. The methods of these states are explained in detail in section III-C on page 33. Finally, the last layer fetches the voxel data from the GPU buffers and sends it to the isosurface extraction ”Rendering node”.

The Rendering node consists of a series of steps. First, a mesh is extracted from the voxel data by using the Marching Cubes algorithm, then the smooth normals for the mesh are calculated, and the mesh is added to the game world. Lastly a material and shader are applied, to add detail and enhance the simulated cave features. This process is described in detail in section III-E on page 45.

This structure is very modular, and new states can be added to the pipeline if needed. If for instance, one wished to do erosion simulation on the voxel data before extracting a mesh, a state accomplishing this could be plugged into the cave generation module, with little difficulty.

This process is repeated for each tile until all tiles have been processed and a fully formed cave is left behind.

B. Generating the Overall Structure

As seen on figure 36 on page 14, the overall structure of a natural cave normally falls into one of six types of branching structures. Furthermore, as seen in section II-B.4 on page 13 caves are generated by expansion of pre-existing fractures and passages by water.
These types of structures are often modelled with success by L-systems, as seen in section II-C on page 14, and due to this match they were chosen as the way to create the structure of our caves.

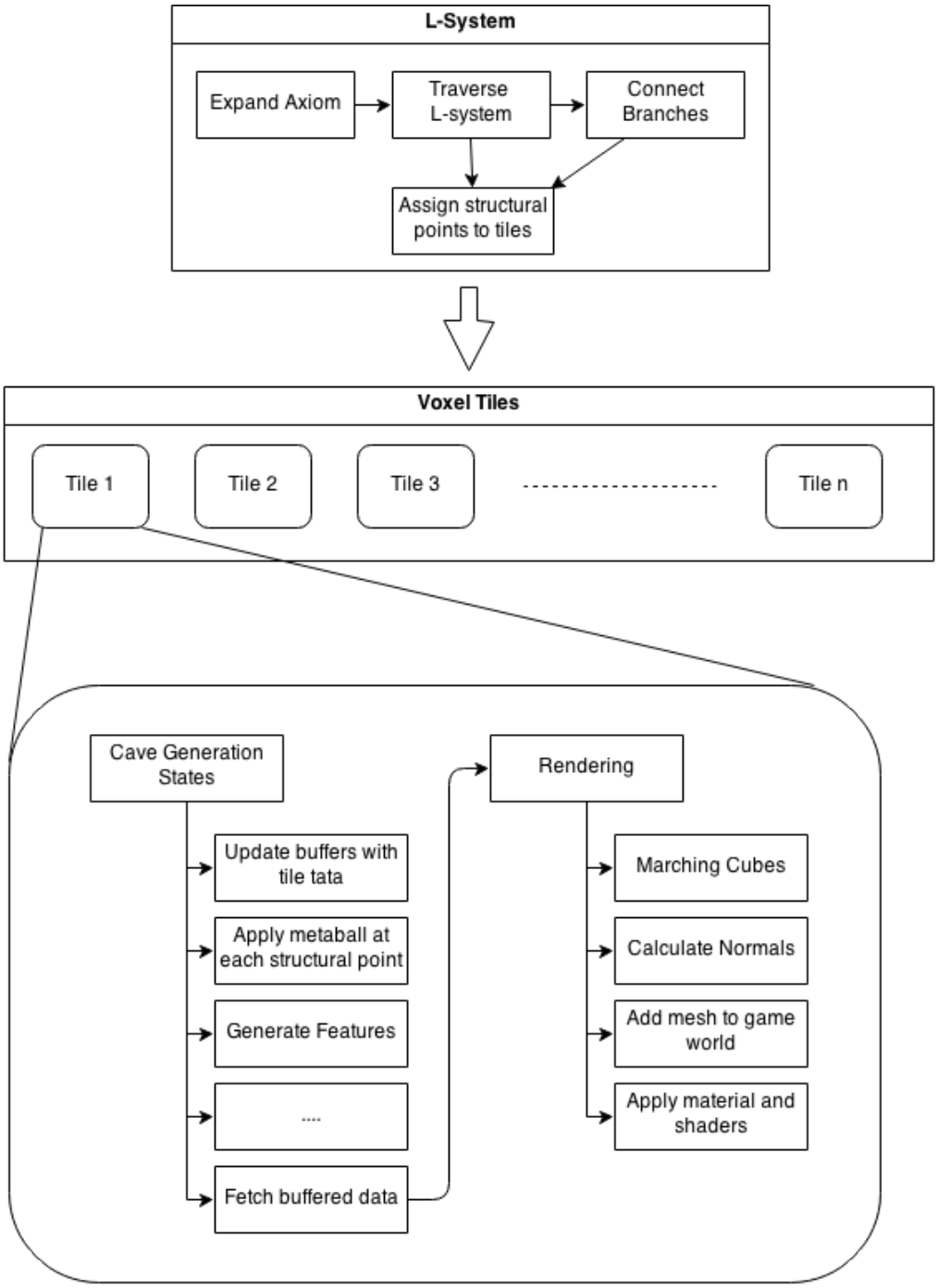


FIG. 64: Diagram showing the overall structure of the pipeline

1) Structure of the L-system

Early on in the design process it was decided that, although a natural look was desired, it would not be achieved through a physical simulation of cave formation. As such, it was only a requirement for the L-system that it could emulate a wide range of structures, with shapes based on empirically observed natural cave formation behaviour and characteristics, ie figure 36 on page 14.
The initial cave structuring attempt was therefore a very simple bracketed L-system, with the alphabet shown in table I on the next page. When converting the L-system string into the structure of a cave, each of the six direction changing symbols are simply interpreted as vectors going in the corresponding direction. These vectors are then added together until a G is encountered, which signals that the current direction vector should be drawn. For instance applying the production rule in grammar (1) on the following page to the axiom G, will

F	Forward
B	Backwards
+	Right
-	Left
U	Up
D	Down
G	Execute move
[ ]	Start/End new branch

TABLE I: The Alphabet used in the initial L-system

result in a vector that points 45°to the right.

$$G \rightarrow F + G \tag{1}$$

The vector is then rasterized to fit the voxel grid and handed off to the next step in the pipeline. This type of L-system, while simple, showed the potential to create cave-like structures with the correct production rules. It did, however, prove problematic to create a wide range of structures using this L-system. While treelike structures were easy to achieve, more complex organic shapes were harder to create. Furthermore, it proved difficult to work with, due to its limited range of capabilities, and it was ultimately abandoned.

In order to address the problems with the first iteration of the L-system, a second bracketed L-system was developed, with the alphabet shown in table II. This L-system uses the

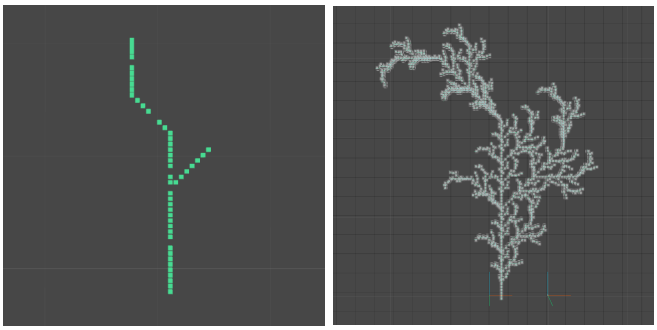
F	Move forward
R	Yaw clockwise
L	Yaw counterclockwise
U	Pitch up
D	Pitch down
O	Increase the angle
A	Decrease the angle
B	Increase the step length
S	Decrease the step length
Z	The tip of a branch (to be expanded)
0	Stops connection to other branches
[ ]	Start/End new branch

TABLE II: The Alphabet used in the final L-system

more traditional turtle approach to generating points from the L-system string. Similarly to the system mentioned above, each angle-changing symbol adds a predefined angle to a rotation vector, which is then applied to a forward pointing vector each time the F symbol is encountered. This vector is then rasterized and handed off, in the same way as above. Similarly to the production rule above, the production rule seen in grammar (2) creates a line going 45°to the right.

$$Z \rightarrow RFZ \tag{2}$$

In addition to this drawing functionality, several symbols that allow finer control over the drawing were included. Table II shows four symbols used to increase and decrease the turning angle and forward step length. These symbols allow for finer control over the shape of the branches. For instance, by decreasing the turn angle and the forward step size, smoother shapes can be created.
Lastly the Z and 0 symbols have no functionality for drawing the structural points, but provide utility. Z denotes the tip of a branch, allowing for future expansion, and 0 signifies that the current branch should not be connected to other branches in the postprocessing step described in section III-B.3 on page 29.



(A) (B)

FIG. 65: The production rule in grammar (3) expanded once (A) and five times (B)

This alphabet can be used to draw a large variety of shapes by creating various production rules, such as the ones seen on figure 65, which are made with the production rule seen in grammar (3).

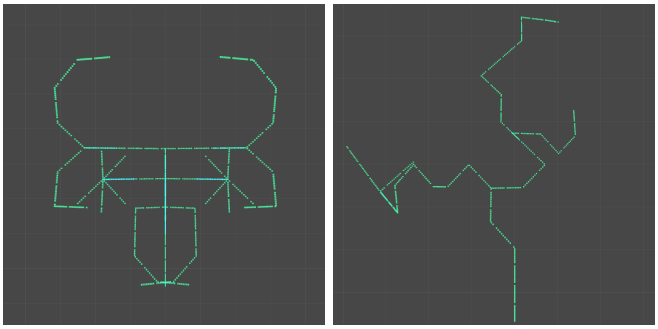
$$Z \rightarrow FZ[RFZ]FZ[LFZ[RF]] \tag{3}$$

Due to the general properties of bracketed L-systems, these structures will, however, have the appearance of organic vegetation (as well as repeating small scale fractals), and while caves share some structural similarities with vegetation, they tend to be less strictly structured.

The main reason for this vegetation-like structure is the use of small segments and simple rules, which are used to rewrite the axiom many times, This creates a very organic and fractal structure, due to the high level of self-similarity. To combat this, longer rules with more complex shapes had to be introduced. These rules requires less expansions to generate a large structure, which result in less self-similarity and a more chaotic structure. To even further reduce the amount of self-similarity of the L-system, as well as increasing its expressive range, an element of randomness was also introduced.
Both of these goals were achieved with a macro system that adds a layer of abstraction between the main production rule and the simple alphabet shown in table II.
These macros, which are shown on table III on the following page, each encapsulate a string of other symbols, which get substituted for the macro when rewriting the axiom. An example of this is seen in grammar (4) on the next page. The

C	A curve
H	A vertical ascent that returns to horizontal
Q	A branching structure that generates a room
T	Similar to the H symbol, but splits into two curves
I	Represents a straight line

TABLE III: The macros used in the current iteration of the solution



(A) (B)

FIG. 66: Two different room structures seen from above. Both of these can be substituted for the macro Q

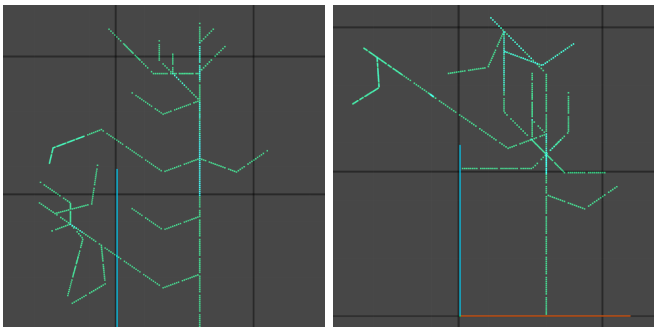
first production rule consists of two macros, which will get substituted for a rule similar to the second production rule before it gets written into the axiom.

$$\begin{aligned} Z &\rightarrow C[H] \\ Z &\rightarrow FLFLF[FUFFDF] \end{aligned} \tag{4}$$

This level of abstraction allows for more compact production rules, but that is not the main purpose for it. Instead, the abstraction was implemented for two reasons. Firstly, it allows the tool to generate relatively well formed production rules, simply by randomly combining a number of macros and brackets, and secondly it allows the L-system to become stochastic, allowing for the randomness mentioned above.
The randomness is introduced by allowing each macro to expand into multiple different strings of symbols as seen on figure 66. Each time a macro gets substituted, all of the strings associated with this symbol have a random chance of getting chosen as the substitute. This chance can be modified by assigning a weight to each string, allowing for some control over the substitution process. A full list of the substitute strings for each macro can be found in section VIII on page 63.

The unusual structures shown by some of these macros are to facilitate the reproduction of certain natural phenomena when the noise gets generated around the structural points. This interaction will be described in detail in section IV-A.1 on page 51.

This stochastic substitution means that the same production rule can generate completely different structures. On figure 67 this is shown in action. The two structures seen are both generated with exactly the same production rule, but differ



(A) (B)

FIG. 67: Two L-systems that are both generated with the production rule $Z \rightarrow I[I][I]$ and expanded twice. Seen from above

due to the stochastic substitutions. The result of this, is that the shape of the structure that is generated is only partially controlled by the production rules used. The production rule will still determine the general tendencies of the L-system, but the actual shape is controlled by the weights of the substitutes at the time of the individual macro substitutions.

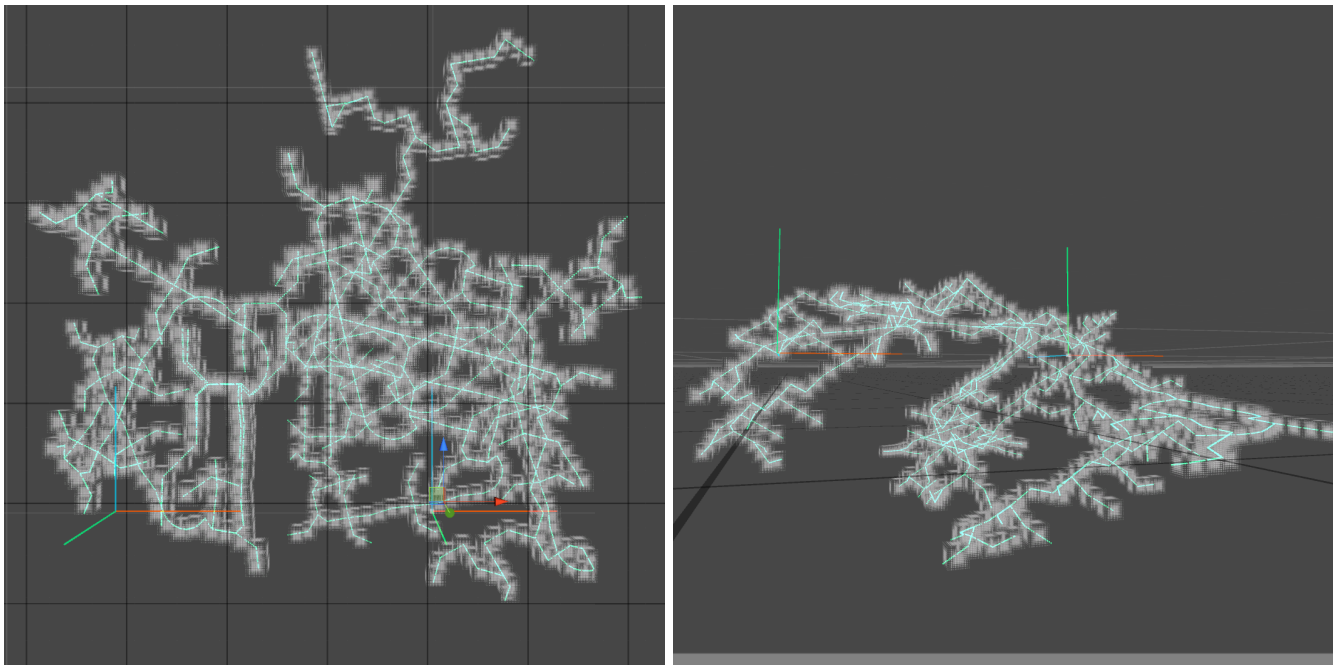
While the production rules of the L-system can be specified manually, that is not an ideal solution, as even though it would be capable of generating a large variety of caves, they will all have the same overall tendencies. To combat this we added functionality for generation of random production rules that can be enabled by the user. These rules consist of a number of macros along with sets of brackets. Examples of these rules are seen in grammar (5).

$$\begin{aligned} Z &\rightarrow Q[III]I[Q] \\ Z &\rightarrow H[Q][H]II[Q] \\ Z &\rightarrow I[I][TT][I[Q]] \end{aligned} \tag{5}$$

The number of macros in each production rule is determined by a range set by the user, and so is the chance for a bracket to be inserted after each symbol. There is, however, a guarantee to have at least a single set of brackets to avoid completely linear caves.
An example of an L-system generated with the random rules described above, is seen on figure 68 on the next page. Repeated empirical observations of this pattern of structure determine similarities to the cave structures shown on figure 36 on page 14, which satisfies our goal of simulating the distribution of natural caves formed via fractures.

2) Comparisons to Other L-system Based Approaches

Compared to the approaches shown in section II-C on page 14, the approach shown above is relatively simple.
Firstly it is not parametric, as opposed to most other L-systems used for PCG. The main reason that the functionality for accepting parameters was never implemented, is that it was not needed to create the cavelike structures shown on figure 68 on the following page section VIII on page 63. The



(A) (B)

FIG. 68: An L-system expanded from the randomly generated rule: $Z \rightarrow I[Q[C[T[TQ]]]]$. Seen from above (A) and the side (B)

fine control over the structure that parameters can create, and which is shown by Parish and Müller[32], and Knutzen[30], is not required in order to create a structure that simulates a winding cave.

Furthermore, some of the parameters that would be included in a parametric L-system are instead included as symbols in the alphabet, such as the symbols to increase or decrease the turning angle.

In order to further expand the functionality of the solution in the future, however, it would most likely be beneficial to make the L-system parametric. This would allow, not only for greater control over the shape of the L-system, but also allow the L-system to interact directly with the metaball brush. An example of this could be a set of parameters that controlled the radius and shape of the tunnel. These parameters could then be used to, for example, have tunnels become more narrow as you progress into the cave, giving a more claustrophobic experience to the player.

3) Connecting Branches

While the L-system described above, satisfies our goal of simulating cave structure models, they leave open ends in the branching, which in gameplay terms translates into tunnels with undesirable dead ends. Even though this is not uncommon in nature, it is common knowledge that needless backtracking should usually be avoided in a videogame. To remedy this problem another step is performed after parsing the final axiom. This step simply involves connecting the ends of branches together to reduce the number of dead ends, or to eliminate the dead ends completely, depending on the selected parameter value.

To achieve this, a list of points where a branch ends without being expanded (hereafter referred to as endpoints) is created when the final axiom is parsed by the L-system. Each branch is considered to be an endpoint until it gets expanded further, in which case it gets replaced with the endpoint of the expanding branch.
After the axiom is fully parsed and a complete list of endpoints is compiled, the list is shuffled, and traversed in order to connect the endpoints to each other. When the list is traversed, each endpoint has a chance, determined by the user, to become the start of a connection. If a start already exists, it becomes the end of the unfinished connection instead, and a line of structural points is drawn between them.
To avoid having a network of straight and unnatural looking tunnels between connected dead-ends, each point on the line is distorted by Curl noise, as seen on figure 69 on the following page. This distorted line guides the metaball brush to generate complex shapes as seen on figure 70 on the next page and figure 71 on the following page.

4) Controllability

A lot of functionality has been added to help control the shape and properties of the L-system. The functionality which has the greatest effect on the shape of the cave is the ability to change the macros used to generate the L-system axiom. While it is not possible to modify the base alphabet or add

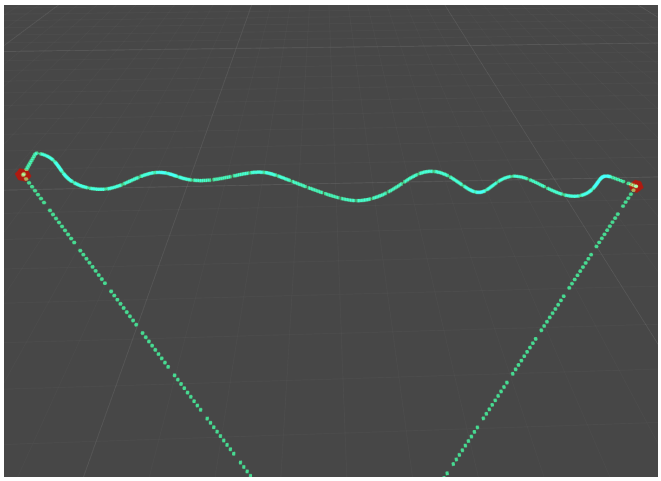
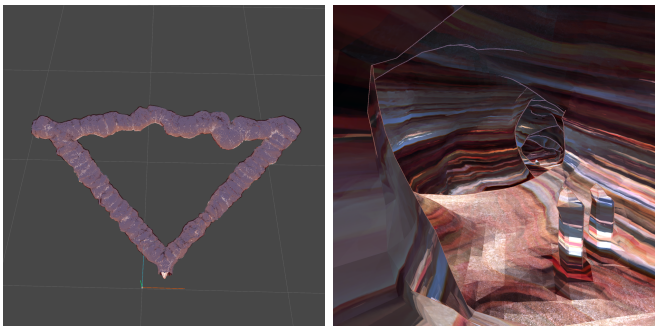


FIG. 69: An example of two endpoints, marked with red spheres, being connected with a distorted line



(A) (B)

FIG. 70: An example of the connection between two endpoints after the finished cave generation process

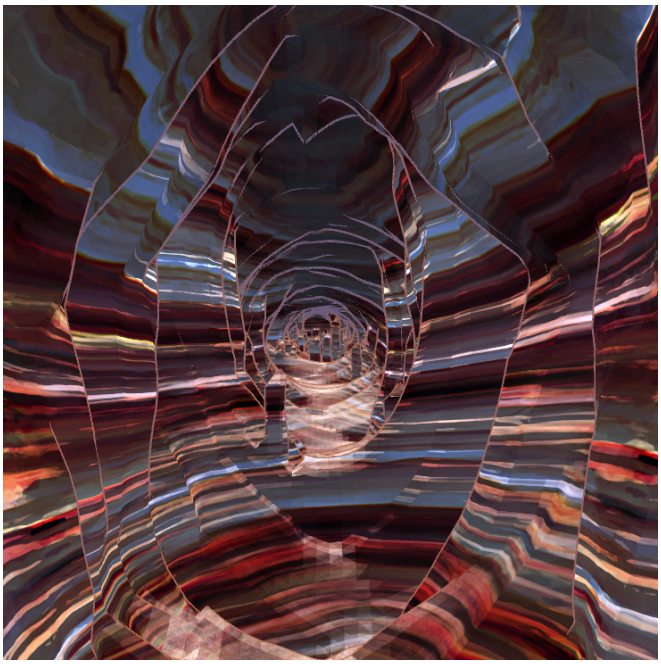


FIG. 71: An example of a cave tunnel generated from a straight structural line. We also note that it presents no symmetry along any axis.

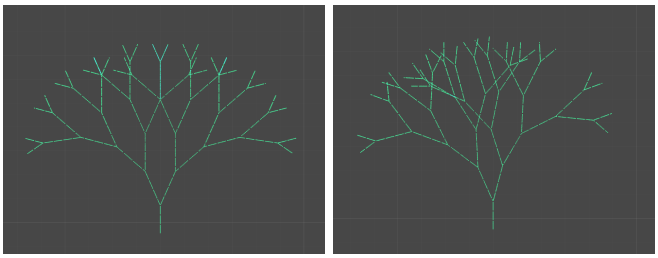
macro symbols without changing any code, it is possible to specify the strings that are substituted for each macro in a special rule file by using the syntax:

```
Symbol;SubstituteString;Type;Weight
```

The Symbol refers to the macro symbol, and SubstituteString refers to the string that is going to replace it when the axiom is rewritten.
The Type field refers to the type of structure that this string is modelling. Currently this field only supports `none`, `tunnel` and `room`. The type is used with the balanced mode described below.
The weight field is a floating point number, that is used to help determine which string gets chosen to replace a macro. If more than one string is available to replace a macro, each of them is assigned a random number between 0 and 1. The weight specified in the weight field is then added to this number, and the string with the largest number is then chosen to replace the macro.
It is also possible to specify production rules in this file, which will be used if randomly generated rules are turned off. The format for these rules are the same as for the macros, with `Z` replacing the macro symbol in the `Symbol` field. The `SubstituteString` field of these production rules can include both macros and symbols from the base alphabet in order to create the desired structure for the cave.
In addition to modifying the rule file, it is also possible to specify the initial axiom to be expanded, allowing control over the general structure of the L-system.

As described in section III-B.1 on page 26 it is possible to randomly generate production rules in order to create a large variety of structures. This functionality has a number of parameters associated with it in order for the user to better control the process.
Firstly it is possible to specify the minimum and maximum number of macros that should be in each randomly generated rule, as well as a percentage chance to insert a bracket between each of them. This allows for control over the size of the L-system as well as its degree of branching.
In order to more finely tune the randomly generated rule, it is also possible to assign a weight to each macro, which works in a similar fashion to the weights used to substitute strings for macros. These weights modify the chance for each symbol to appear in a generated rule. With a weight of -1 it is, for example, possible to stop a macro from appearing in rules completely.

Apart from macros and production rules, many other parameters that control the L-system are available to the user. On an overall level, it is possible to change how many rewrites are performed, which mainly control the size and detail of the L-system.
It is also possible to change the length of each forward step as



(A) (B)

FIG. 72: Two L-systems generated with the exact same ruleset. (A) has no angle deviation, whereas (B) has 15 degrees of angle deviation

well as the angle that a rotation symbol modifies the overall rotation by. These two parameters specify the initial values for these quantities, but they can still be modified while parsing the axiom, if any of the relevant symbols, seen on table II on page 27, are encountered. The value that these symbols modify the overall parameters by, can also be specified via their own parameters.
The vertical angle of the L-system can also be limited. If this parameter is set to a positive number, it will limit the pitch of any L-system lines to an angle specified by this number. This can be set in order to limit the verticality of the system, if a relatively flat cave structure is desired.
Additionally it is possible to add randomness to the angles mentioned above via another parameter. When choosing the angle to add to the rotation vector, a random value between the original angle +/- the parameter. This can give a more organic look to the L-system as seen on figure 72.

To control the overall tendencies and extent of the L-system, three different modes are available:

1) A target mode, which allows the user to guide the L-system in the general direction of a point.
2) A balanced mode, which allows for some balance between rooms and tunnels.
3) A constrained mode, where the L-system can be constrained within certain boundaries, denoted by either a sphere or a box.

These modes can be mixed and matched to determine the overall shape of the L-system.

The target mode lets the user define a target point, via the placement of a game object in the unity scene. If target mode is enabled the L-system will then try to gravitate towards this point by modifying the weights of the various macros when substitution occurs. The weights are modified based on two parameters. One is based on the change in distance to the target that applying this specific substitution would create and the other is based on the angle between a vector pointing straight at the target and the vector that describes the facing of the end of the current branch after applying the specific substitution. The prospective weight changes from these two

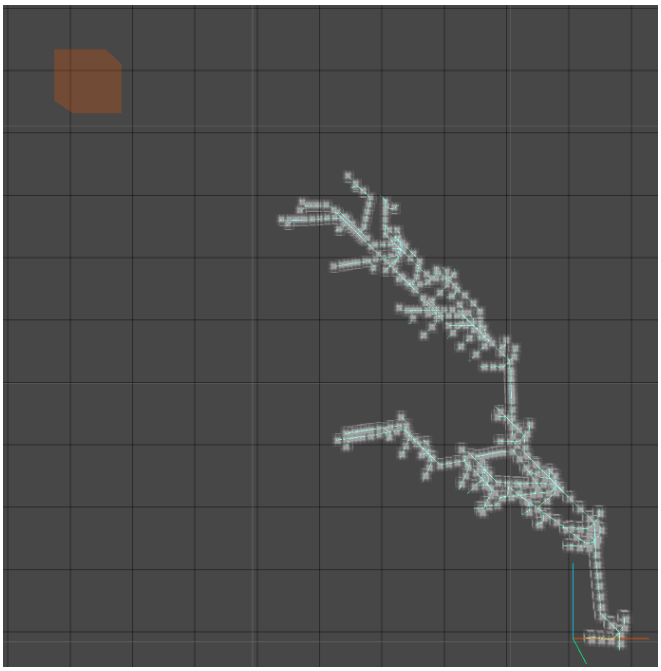


FIG. 73: The simple rule $Z \rightarrow II[I][I]$ expanded three times with target mode turned on

parameters are then averaged out and added to the weight of the substitution string in question.
The reason for using two separate parameters to modify the weight, is that each of them separately do not always tell the full story about the orientation of the L-system after applying a macro substitution. It is, for instance, possible for a string that ends up turning the L-system away from the target to end up being closer to the target than the alternatives that could possibly have a more beneficial facing. Similarly it is possible for a string that has a beneficial facing to end up further away from the target than other strings with slightly less beneficial facings. With that in mind, it is possible to get a candidate that is likely to both face towards the target and be closer than the current point.
The target mode, while great at directing simple rules, such as the one seen on figure 73, struggles to direct complex rules, such as the ones generated randomly, as seen on figure 74 on the next page. The problems with target mode are twofold. Firstly only the macros C and I have enough variation in direction to alter the general tendency of the L-system towards the target. Secondly, branches are not considered when calculating the overall direction of a substitution string for use with target mode. This is the case, as a single weight must be calculated for the entire string, even if it contains branches, which might potentially go in a different direction than the main stem. This means that the branches of a string might actually point away from the target and end up getting expanded in the wrong direction. This, along with the limited direction changing ability of most macros which was mentioned above, means that target mode does not deal particularly well with macros apart from C and I. Without fundamentally changing the functionality of the target mode

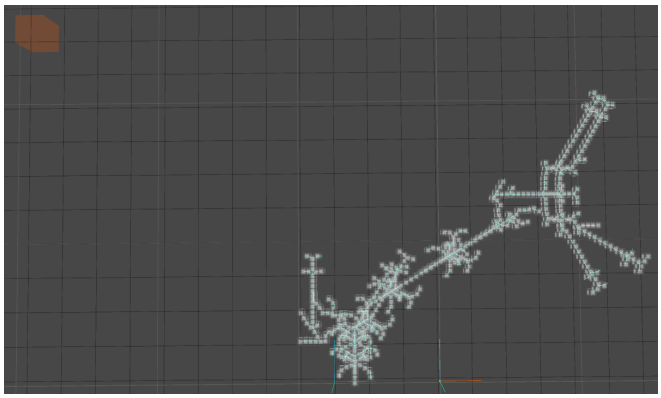


FIG. 74: The more complex rule $Z \rightarrow QIH[CT[T]]$ expanded two times times with target mode turned on

to directly alter the L-system, the only way to fix this is to add more substitution strings to the macros that needs them.

If balanced mode is turned on, the L-system will try to strike a balance between rooms and tunnels. A substitution string's type is specified in the rule file, and can be either a room or a tunnel. It tries to balance the two out by, again, modifying the weights of substitution strings. If a tunnel is substituted for a macro, the weights of any room strings in following substitutions will be incremented and the weights of any tunnels decreased by an amount specified by the user, making it more unlikely to pick another room. If a room is substituted, the same process occurs, but in reverse.
This works well in theory, but with the set of macros that currently exist, it is not very useful. This is because it only works on a macro by macro basis, as it modifies the weights of each substitution string when deciding which one to substitute for the macro. That means that if every single structure that can be substituted for a given macro is of the same type, balanced mode has no effect. This is currently the case for all macros in the rule file that is supplied with this solution, but it could be made useful with a custom rule file.

In addition to the other two modes, the L-system can also be constrained to a volume. The user can simply define either a box or a sphere, and any branches that are outside of this volume will not get drawn. Any branch that gets cut off by the edge of the volume is also considered an endpoint, so it can be connected to the other endpoints as described in section III-B.3 on page 29.

It is also possible to control the connection of endpoints via a number of parameters.
Firstly it is possible to specify the chance that each endpoint has to be connected to another endpoint.
Secondly the user can specify a maximum distance for connections. This is a useful parameter that can be used to avoid the creation of very long singular tunnels between distant corners of the cave, if those are not desired by the user. To avoid the end point connection algorithm getting stuck on an endpoint that is too far away from any other endpoints to be connected to anything, it will discard any starting point of a line that is too far away from the ending point it tries to connect to. The ending point will then be used as the starting point for a new line instead.
Lastly it is possible to modify the frequency and amplitude of the Curl noise that is used to distort the line, to be able to control the feel of the endpoint connections.

5) Alternatives to L-systems

While L-systems were ultimately chosen as the main method of creating the structural framework of the caves, other methods were initially considered to fill this role.
Togelius et al.[46] provides an overview over several such methods for generating the overall representation of a cave or dungeon.

One of the methods described is the creation of dungeons through space partitioning. This approach is presented in 2D using quadtrees, but could be extended to 3D by using octrees for partitioning.
The basic principle of this method is to partition the level space via the Binary Space Partitioning algorithm (which ensures that no two rooms overlap), and then assign each partition as either empty or full after partitioning has completed. The partitions, which are now acting as rooms, are then connected to form the final level.
The reason that this method was not used over the L-system, is that due to the way that rooms and tunnels are generated, the overall structure will appear to be a number of boxes with tunnels connecting them, as seen on figure 75 on the next page. While this is perfectly acceptable for a 2D dungeon crawler, it would not be ideal for a natural looking cave. The partitions could be smoothed out to create more natural looking rooms, and the path of the tunnels could be distorted to become windy. The overall structure of the cave would still however resemble a string of large rooms connected by tunnels; as is also the case with Yurovchak's approach [44] covered in section II-B.3 on page 10, figure 29 on page 12.

Another method that is described by Togelius et al.[46], is the use of carving agents to dig the overall structure of a dungeon or cave. This is again shown in 2D, but there is nothing stopping this approach from being extended to 3D as well.
The basic idea behind this carving agent is to have a software agent with AI, carve a dungeon out of a solid block. The shape of the final dungeon or cave is completely decided by the type of AI used for the agent. The first example presented, is a stochastic agent that has a set of changing probabilities tied to actions such as turning, and creation of rooms, and carving out a dungeon at random.
This approach is, in fact, fairly similar to our L-system based approach. The turtle that draws the L-system is in some ways similar to a carving agent. It takes the L-system string as instructions in lieu of an actual AI, and then places a list of structural points to carve out the structure of the cave.

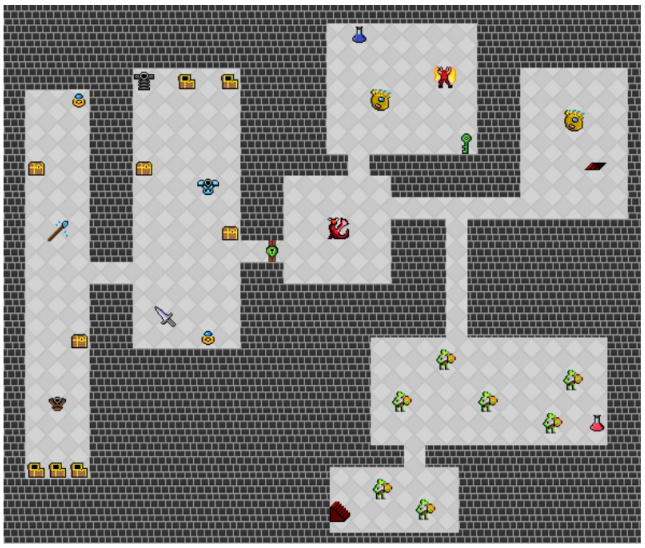


FIG. 75: A 2D dungeon generated with space partitioning. Source [46]

The advantage of this approach over a regular agent based approach is the better control over the entire structure that the L-system offers. While the AI of an agent can be made more sophisticated than the simple stochastic example above, it would require more work to get the level of overall control that the L-system offers and more importantly, would become too slow for real-time generation.

A method that is worth mentioning as well is Perlin Worms, which are used by Minecraft[81] to generate its caves. This method uses Perlin noise to generate winding structures similar to worms.
The basis of this approach is a straight line going through a field of Perlin noise, which stores the values of noise it encounters. The second component is a ”worm”, which consists of a large number of segments. To each segment is assigned a noise value from the line, based on its position in the worm. The segments are laid out one after the other, with each of them angled based on its noise value, creating the winding pattern seen on figure 76.
These structures are extended to 3D simply by using 3D noise and coordinates rather than 2D, and form the basis of the cave systems in Minecraft. As they are used in one of the most popular games that involves exploring caves, they are clearly well suited for this purpose of modeling caves in games, but they are still not ideal for our purpose.
The main reasoning behind avoiding Perlin Worms for this solution is that they are not controllable. The only way of controlling the path of a Perlin Worm is by changing its seed, or starting position, in the 3D noise field, and the amplitude of distortion. This leaves no control over the direction, specific shape. Control is important to our solution, as it was previously described (section I-B on page 2). Perlin Worms are definitely suitable for a game like Minecraft, where the world is represented in very low resolution, and the ability to dig your way out of a dead end exists.

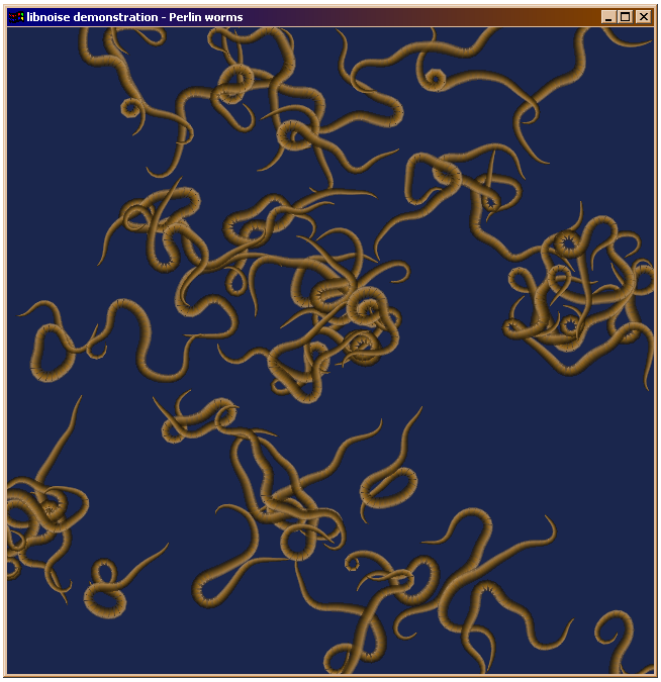


FIG. 76: 2D worms drawn with the perlin worms algorithm[97]

C. Voxel Processor and Detail Modelling

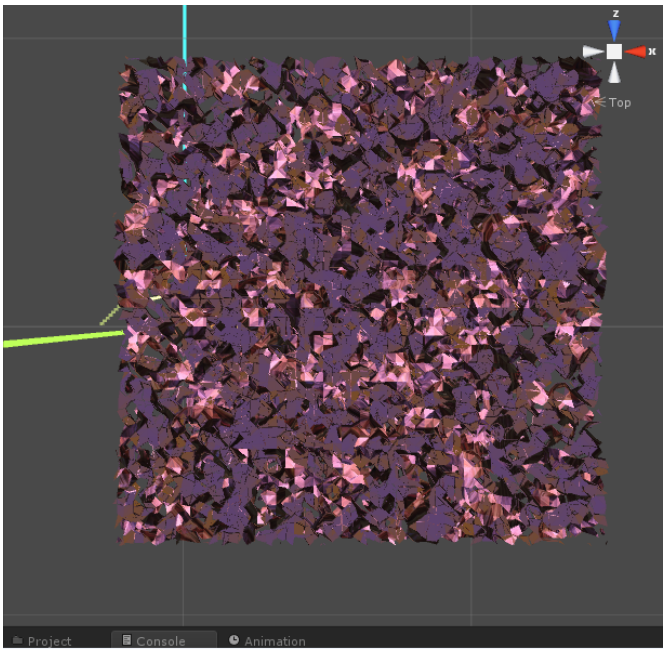
This section elaborates the methods explored for the modelling of the cave interiors. The methods are explained, and related to the approaches in the reviewed literature (section II-B on page 7) and the natural geologic phenomena (section II-B.4 on page 13).

The skeleton of the cave model described by the structural points (the L-system), is encompassed in a cylindrical volume of Simplex noise. Then, depending on the approach (listed below), the noise gets post-processed, moulded into a cave, through subsequent layers of Compute Shader programs followed by the Isosurface Extraction. Figure 77 on the following page depicts this process, visualising the voxel structure for the case of Cellular Automata being used to model the cave interior as well as to build stalactites and stalagmites.

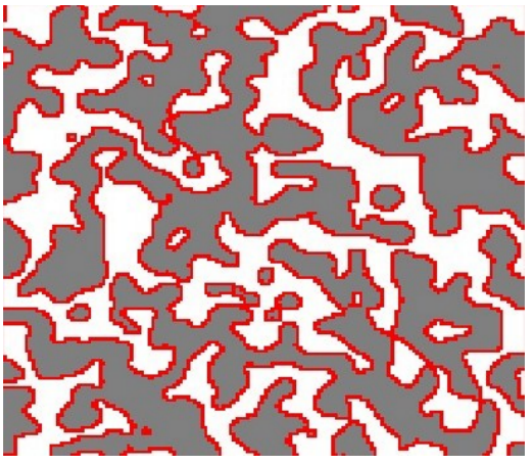
1) Cellular Automata Based Approach

As mentioned in the Related Work section, Musgrave et al [13] used CA on a sparse, low resolution voxel volume of different material properties, to simulate erosion, after which other smoothing and mixed initiative modelling layers were applied. Since our aim was to simulate caves in a quick manner (for real-time generation and not interactive modelling), erosion was not considered a beneficial step. A faster method had to be used to create the same landmarks (walkways, arches, overhangs).

Our initial attempt however, was to test the potential of Cellular Automata in a similar way, as an experiment of



(A)



(B)

FIG. 78: Our initial CA approach applied to a cube of voxels(A) and the 2D CA shown by Yannakakis et al.[3](B)

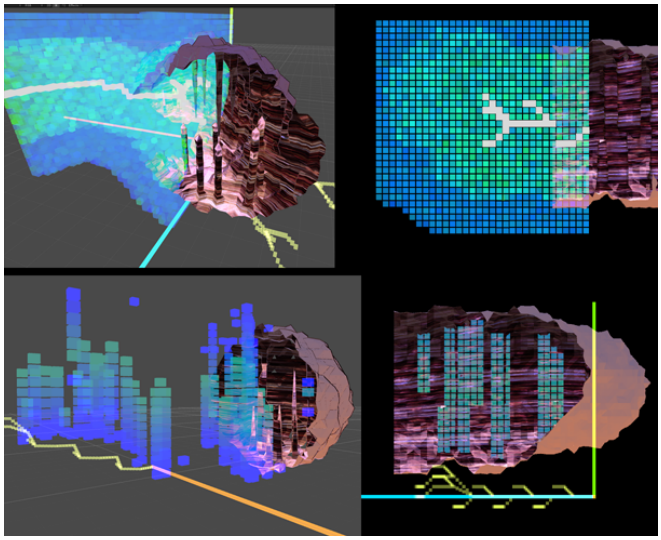


FIG. 77: Top half: noise is populated around the structural line (white), and further processed by CA. Bottom half: afterwards, noise is used to pick positions for stalactites and stalagmites and their height, and CA is used to build them. Green represents emptier space (closer to 1) and blue represents fuller space (closer to -1).

extending the methods described by Yannakakis et al.[3] and Kun[4], for 2D cave generation, to 3D.

Our first attempt to extend the 2D method to 3D simply involved applying a CA similar to the 2D methods mentioned above to a 3D noise field. This approach, as seen on figure 78, was rather naive, as expanding this 2D structure to 3D essentially just creates random geometry. Additionally the goal of the 2D approach is to create both the tunnels and structure of a 2D cave, whereas the structure or our cave was already determined by the L-system. This approach, when applied around the L-system to form tunnels, was essentially digging caves inside our caves.

To try and remedy this problem, the CA rules were modified to create larger chambers and passages. This did improve the result, as seen on figure 79 on the following page, but it did, however, not produce acceptable caves, mainly due to floating geometry and (partially) obstructed tunnels. One aspect of the 2D approach is that any isolated clump of solid material simply becomes a column in the final cave. In 3D, however, those clumps may not be attached to the outer wall of the cave at all, creating floating rocks inside the cave. It was also possible for the CA to block off tunnels, as seen on figure 80 on the next page and partially on figure 79 on the following page. It would have been possible to work around these blocked tunnels by using constructive methods such as the ones seen in [46], but it was decided that it would not improve this approach enough to make it useable. Having to use a constructive approach would add another performance intensive step to the process as well as partially negate the advantages that were gained by using an L-system for the structure in the first place, as described in section III-B.5 on page 32.

Realizing that a traditional CA approach would not work in a 3D environment to create functional tunnels and rooms, a different approach was adopted. Instead of letting the CA create tunnels from scratch, a tunnel of empty space was created around the structural points to ensure that the center of the tunnel would always be empty and passable. A CA step was then performed to widen the tunnel and create details on

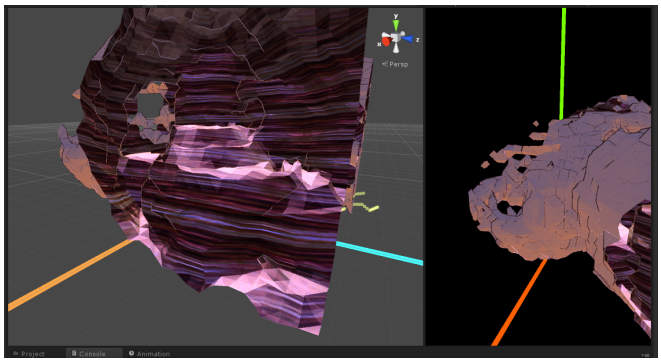


FIG. 79: A later iteration of the CA approach. Notice the partially obstructed tunnel, the floating geometry and holes in the cave wall

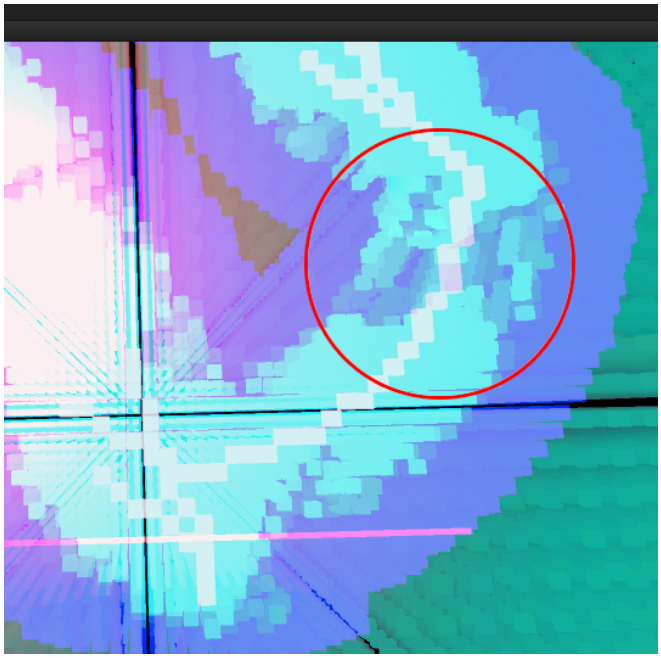


FIG. 80: Red circle shows region of CA which has blocked the tunnel passage.

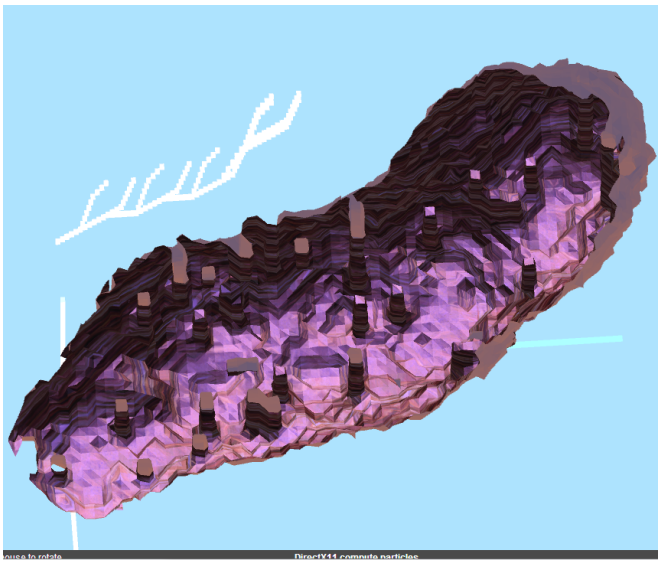


FIG. 81: Cave section generated with an empty tunnel in the middle and expanded by CA. Notice the blocky nature and seeming lack of detail.

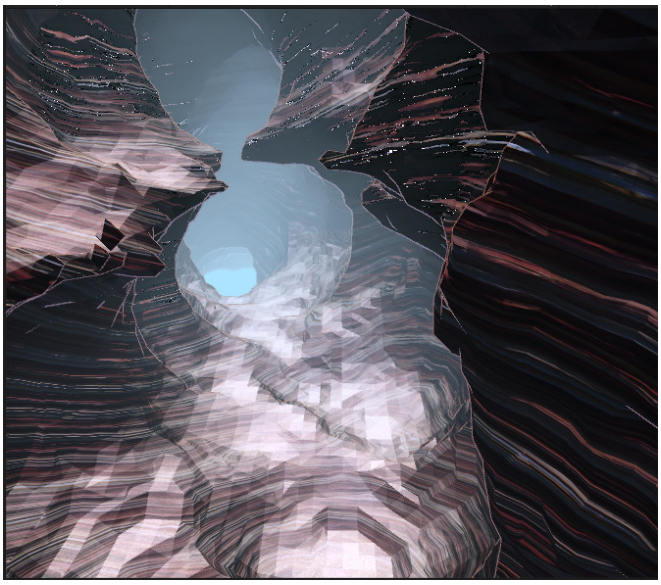


FIG. 82: The smoothing CA in action. Notice that the outer wall is generally much smoother, but it still looks like a low resolution mesh in many places

the outer wall.

As seen on figure 81, this did, however, not create a smooth and detailed cave. This is due to the mechanics of the isosurface extraction, described in section III-E on page 45, which require smooth transitions between the noise values of neighbouring voxels in order to create a mesh with a smooth appearance. This was not possible to achieve with a simple CA approach that also produces interesting details and features on the inside wall. Instead, it appears as if the mesh has a much lower resolution than the number of polygons it actually has.

To try and get both the interesting features, and a high resolution (smooth) appearance, a second CA was used to smooth out the features created by the CA described above. This did create a noticeably better result, as seen on figure 82, but the perceived resolution is still fairly low, which is especially visible on the floor of the cave.

Another problem with this solution is that it is quite performance intensive. In order for the CA to build features and also perform an acceptable amount of smoothing, it has to run for 8-12 frames per tile. Furthermore The CA operations cannot be applied immediately, to avoid undefined behaviour when the noise values of voxels are updated in parallel. Instead, the result of a CA step has to be applied in a separate frame, doubling the amount of frames required. As the number of tiles required for representing an entire cave is 800-1000 for regular sized caves, an extra 16-24 frames per tile could reduce the total generation time by up to a second per tile.

Lastly, while this approach produced an almost acceptable perceived resolution of the mesh, it could not actually produce interesting structures. With the smoothing in place, the tunnels

would be almost completely cylindrical, with little interesting features or detail. It was clear that a fractal approach would be more beneficial to creating the sort of detail on the cave wall that was desired. Additionally, with the CA approach it was very hard to control direction, with the end result only depending on the initial input of noise, and due to our desire of having more direct control over the detail of the cave walls as well as the need for a more fractal approach, the metaball approach described in section III-C.2 was developed, which removes the need for CA to model the details of the inside wall completely.

While CA for the overall shape of the tunnels was abandoned, they are still used for the features that are positioned in the cave after the initial expansion of tunnels, which are described in detail in section III-C.3 on page 38.

2) Metaball Approach

In order to provide the variation and the features was required, an approach was decided on that leveraged the L-System structure as a guiding hand for a virtual 3D sculpting brush. Greeff's [14] Wires solution (explained in section II-B.2 on page 8) served as an inspiration. Whereas Greeff's wires are shaped by a human agent, our L-System structural skeleton is automatic, and the procedurally warped metaball is guided along its branching, winding path to dig what results in our tunnels, chambers and ravines.

The metaball was procedurally warped, because a spherical shape, through its nature, would provide only smooth tubular tunnels and sculptures. Therefore an experimental analysis of different types of noise functions was required to ensure that the metaball brush creates natural shapes which match the patterns of scallops found on cave walls, and form vadose and phreatic passages by moving along the various combinations of L-system branches. We notice that the inner empty volume of caves present silhouettes which can be simulated by combinations of noise applied to a spherical volume, as seen on figure 38 on page 15.

As described in section II-B.3 on page 10, the classic metaball is a function which represents a spherical energy field. To replace the unsatisfactory detailes created by Cellular Automata, our solution was to perturb the metaball's energy field into specific patterns and shapes.

Each metaball is spawned at a structural point, and therefore has the structural point as the centre of its energy field. The energy field goes from 1 (empty space) at the heart, to -1 (full space) at its outer boundary horizon. It is then distorted by a series of noise functions as described below. The energy field is applied in parallel, to the whole voxel volume, in the compute shader, therefore it sculpts all its encompassed voxels at once.

The warped metaball function is implemented in HLSL as a modified function of distance, and composed of the following elements:

$$meta\_dist = Ed(vp_{xyz}, f_c(sp_{xyz})) * (f_{sn}(vp_{xyz}) + f_{combine}(f_{vL}(vp_{xyz}),\ f_{vM}(vp_{xyz}),\ f_{vH}(vp_{xyz}))); \quad (6)$$

where sp_{xyz} represents a Structural Point (with its 3D coordinates).

vp_{xyz} represents one of every Voxel which reside in the vicinity of the current Structural Point.

$f_c(sp_{xyz})$ represents the computation of a low frequency and low amplitude Curl noise value from the position of the current structural point (and is actually computed on the CPU side during the stage of tile list creation). It is responsible for skewing the sphere into an assortment of oval shapes. Curl was chosen over plain Simplex due to its nature of abruptly changing its values (having abrupt transitional spots among otherwise calm intervals), forming localised large air pockets: figure 83 on the following page. In order to allow for the creation of flat ground, this distortion is limited on the Y (up) axis (by the customizable value of a slider).

$Ed(vp_{xyz}, f_c(sp_{xyz}))$ is the Euclidean distance between the current Compute Thread's voxel's position, and the skewed version of the current Compute Shader's Structural Point's position.

f_{sn} represents a function of Simplex noise of low frequency and high amplitude. It is low frequency because the subsequent high frequencies of Voronoi noise are a better match for rocky scallops (fine concave detail on cave walls). This decision is explained with the experimental results further below.

f_{vL}, f_{vM} and f_{vH} are functions which compute Voronoi noise. f_{vL} computes a low frequency Voronoi noise function, and the other two respectively have increased frequencies by a factor of two each. The lowest frequency provides abrupt angle changes in the otherwise circular tunnel shape. The finer frequencies provide detailing.

The noise value for the vp_{xyz} (each voxel point under the metaball's influence), is computed to be somewhere between 1 and -1; 1 being the noise value at the centre of the metaball (at sp_{xyz}). More precisely, the noise value is $meta\_dist$ units away from sp_{xyz}, and always falls between 1 and -1.

In addition to what has been depicted in equation (6), each major noise component in the metaball distortion function has its weight (amplitude) modulated along one of three low frequency Simplex noise samples in order to facilitate an alternation between cave features (a slow alternation over a longer distance).
The main result of this is the alternation between smooth phreatic tunnels (figure 84 on the following page), and vertical vadose tunnels (figure 85 on the next page), as well as combinations of the two: figure 86 on page 38. Note that

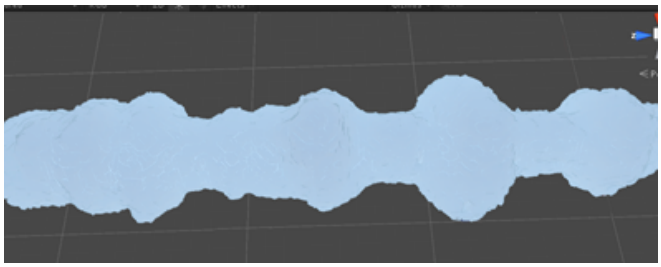


FIG. 83: Top view of the results of $f_c(sp_{xyz})$, where sp_{xyz} is sampled from an array of structural points which only contains a straight line. The amplitude and frequency have been increased for this figure, in order to easily and clearly illustrate a wider assortment of distortions in the limited figure space.

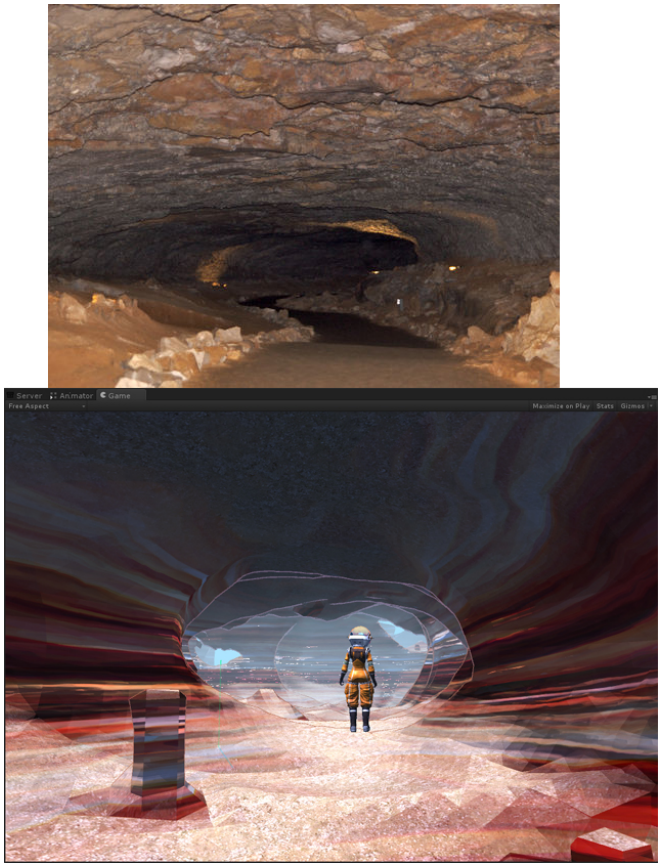


FIG. 84: Comparison between our caves and a section of Mammoth Cave National Park: short, wide and smooth corridors.

FIG. 85: Comparison between our caves and a section of Mammoth Cave National Park: narrow and tall corridors.

additional verticality for vadose structures, is obtained when two or more horizontal structural branches are found parallel and stacked on top of each other, as seen on figure 87 on the next page (and similarly for horizontally wide phreatic chambers).

With figure 77 on page 34 the generation process within a single voxel tile was explained, with focus on the Cellular Automata approach. We now present in figure 88 on page 40, the final form of the cave detail modelling solution, using the smooth metaball energy fields.

Multiple structural points (each of which creates a metaball) are sampled sequentially and applied to the voxel volume. These structural points are sampled sparsely from the otherwise dense L-System lines. One structural point is sampled every $metaballDiameter$ distance. Each metaball instance applies its energy value to the voxel field, and contributes to the carving of the 3D space. Due to the warped nature of the metaballs, multiple landscape patterns can emerge, such as hills or sharp peaks, cracks or windows through thin walls, plateaus, small mesas etc.

Below we briefly describe the process which has lead to the solution described above (namely the choices in noise functions). Multiple noise types have been considered (as discussed in section II-A.2 on page 4, and referring to Ebert et al [35] and Lagae et al [38]), but only a few have been focused upon as they were both recommended and widely used. Furthermore, preliminary empirical observations clearly showed a close relation to the cave shapes we sought to simulate.

The initial experiment was to warp the metaball though Simplex noise and Curl noise: figure 89 on page 40. The results proved unsatisfactory (figure 90 on page 40). Simplex noise is well suited for large scale mountain ranges (figure 19 on page 10), but unable to provide blocky stone structures for cave walls (figure 91 on page 40). While the addition of curl noise resulted in natural water-vortex-carved pockets in the cave, the curves were perfectly smooth and organic; better suited for a macro environment. It exhibited a distinct lack of the sharp turns and angles characteristic to real caves.

To achieve a sharper perturbation, different frequencies of Voronoi noise were tested. A low frequency and high amplitude Voronoi noise function would clearly show its cellular nature, and create rift-like partings in the geometry, as seen on figure 92 on page 40. Such strong amplitudes were ultimately scrapped, however, due to unreliability (e.g. not desirable for walls, constant rifts in the tunnels similarly not desirable). We leave the research of spawning deep cracks in

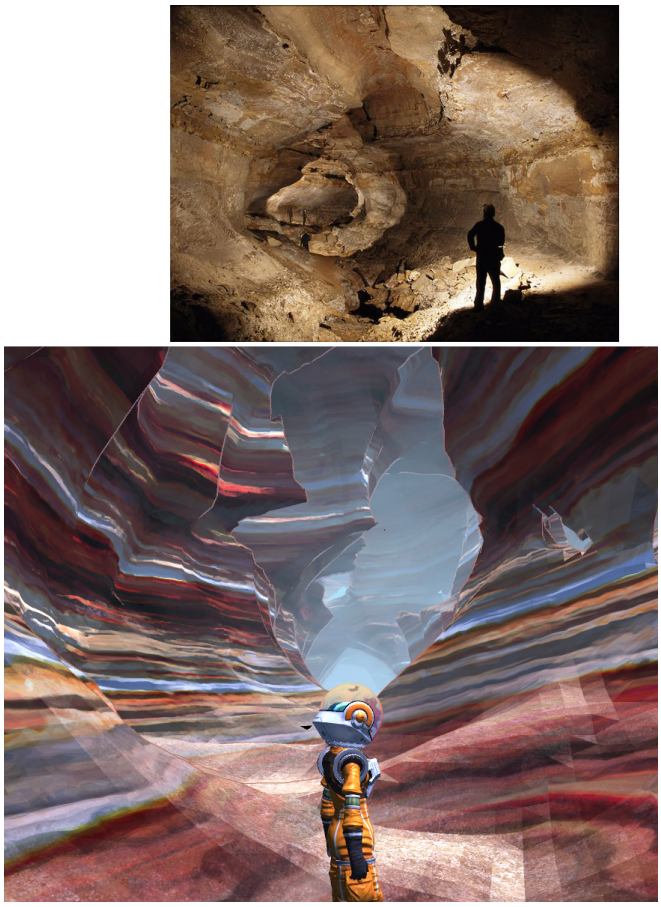


FIG. 86: Comparison between our caves and a section of Mammoth Cave National Park: vadose and phreatic combination.

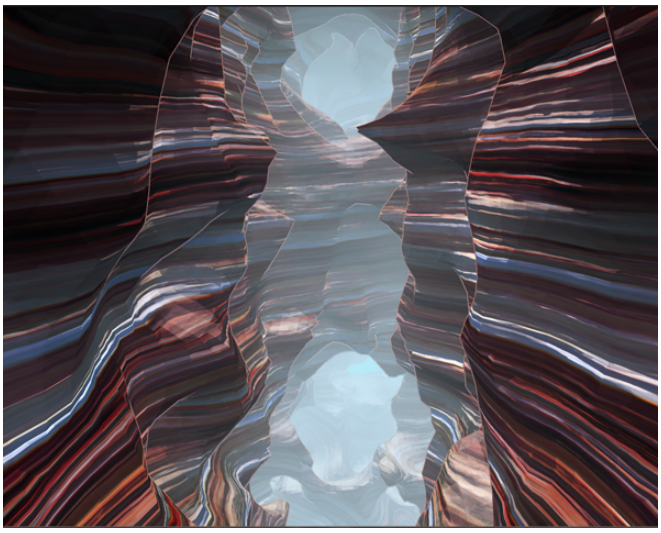


FIG. 87: Verticality due to stacked structural lines, creates more accurate Vadose passages.

the ground open for future work.

In the following diagram (figure 93 on the following page) we show a range of shapes that the metaball 3D brush can take, while warped by different types of noise. These variations can be selected and customised in our project via sliders.

We note that further detailing and granularity are required if it is decided to produce cave meshes of higher resolutions, and that for that fine level of detail, new types of noise should be researched in addition to those currently used. However, due to performance constraints for real-time rendering, this step can only be added as a procedural tessellation post-processing step (instead of increasing the polycount of the raw mesh), which was beyond the scope of this project. As it can be seen in figure 94 on page 41, attempting to apply any higher frequency noise than the ones already used, on our polygon density, would result in incomprehensible and undesireable noisy patterns.

After the implementation of a robust and user-friendly interface, the specific parameters of our cave detailing system would become open to a game design artist to explore and tweak. Furthermore, a node based interface can be added to allow an artist to plug different types of noise into the metaball warping equation or even to use a visual programming scheme to alter the equation itself. Regardless, the balance of functions and parameters that we have achieved serves as proof that our approach is very effective at modelling virtual cave interiors.

3) Features Inside the Cave

Features, such as stalactites, stalagmites and columns, can appear prominently in natural caves, as seen on figure 95 on page 41. While the current rendering techniques employed for this solution cannot render these features at the level of detail shown in the picture, it is important to note that our solution supports these attributes.

The features are added in a separate step after the initial tunnel has been carved out with the metaball, as described in section III-C.2 on page 36. The placement of stalactites are similar to the approach shown by Yurovchak[44] (described in section II-B.3 on page 10), but differs in the initial selection phase. Instead of picking points at random, the potential spawn points for features are selected based on low frequency simplex noise, by marking every point with a noise value within a certain range as potential spawn points for features. This ensures that these features are spawned in clusters, instead of randomly throughout the cave. The actual spawn points of these features are then chosen based on very high frequency simplex noise in the same way as the spawning clusters and Yurovchak's approach. An example of this selection in action can be seen on figure 96 on page 41. Due to the fact that every voxel in a tile gets processed in parallel on the GPU, this selection process can be completed within a single frame.

With a number of points now selected for feature creation, the floor and ceiling above and below each point is detected, and these points are designated as anchor points which will be grown into features. Two types of these features exist,

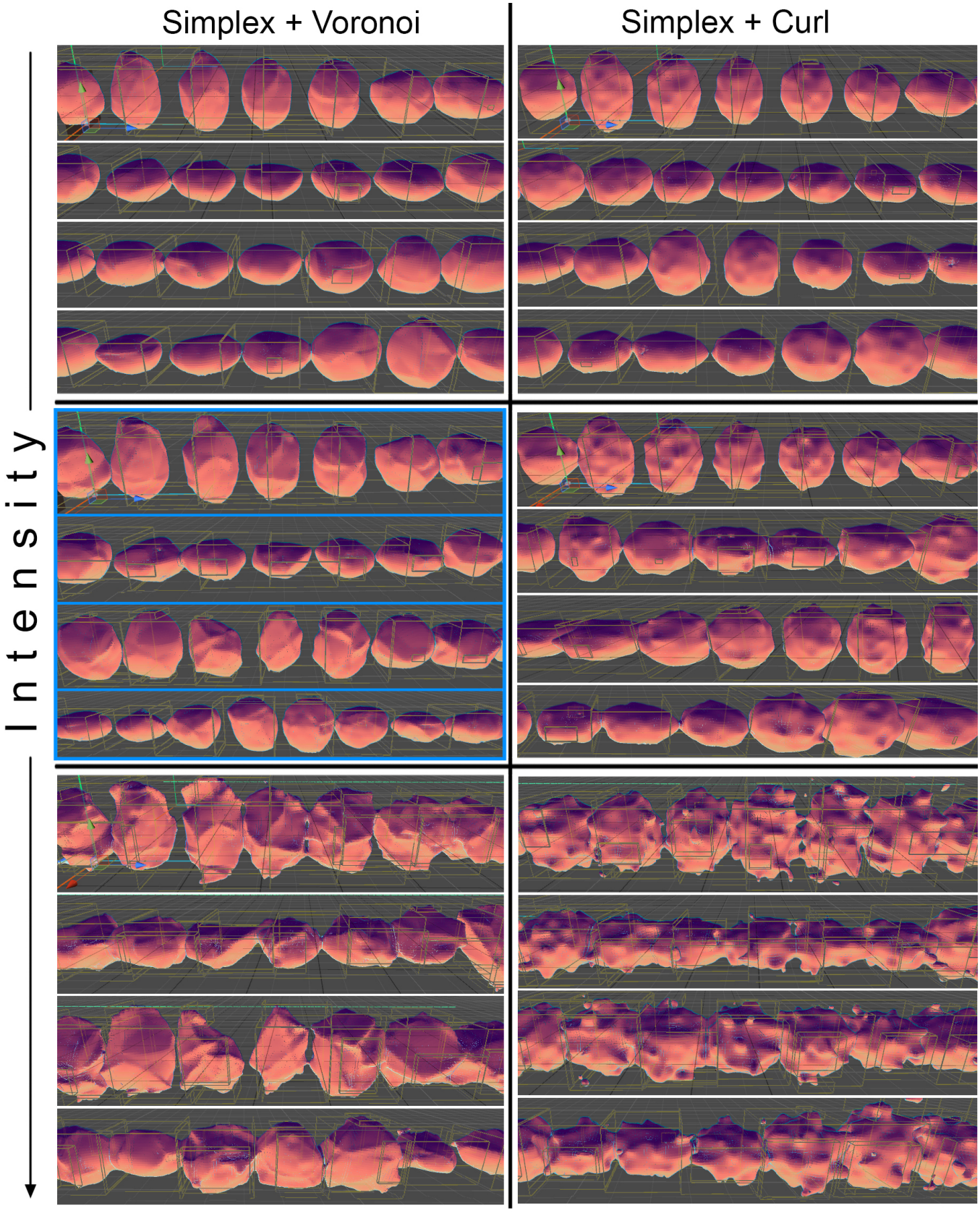


FIG. 93: This diagram shows experimental results of applying different noise intensities to a series of metaball brushes in our two main approaches: Voronoi noise vs Curl noise. The tile marked with blue is the setup currently used by our solution.

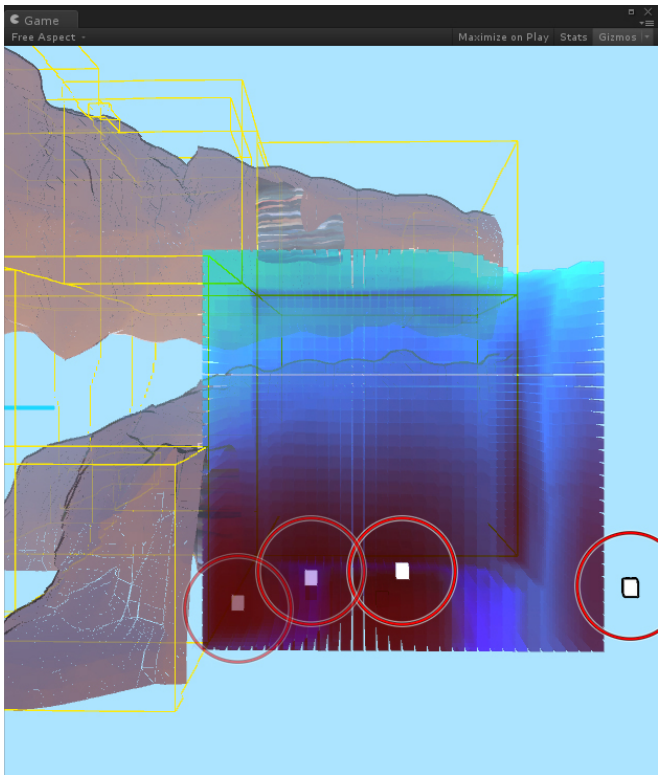


FIG. 88: Blue volume represents the visualisation of noise values in all the individual voxels which fell within the range of a metaball. Multiple metaballs have been applied to this volume, 4 of which can be seen, their centres marked as white dots (white dots circled in red).

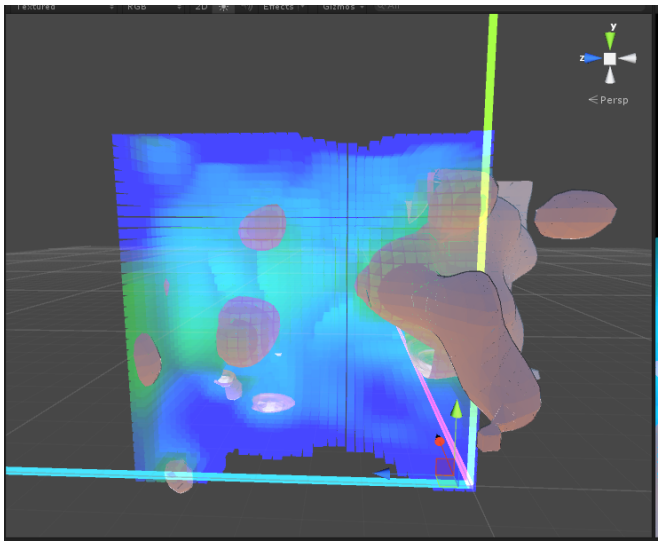


FIG. 89: Metaballs distorted by Curl noise + Simplex noise, building a cave section.

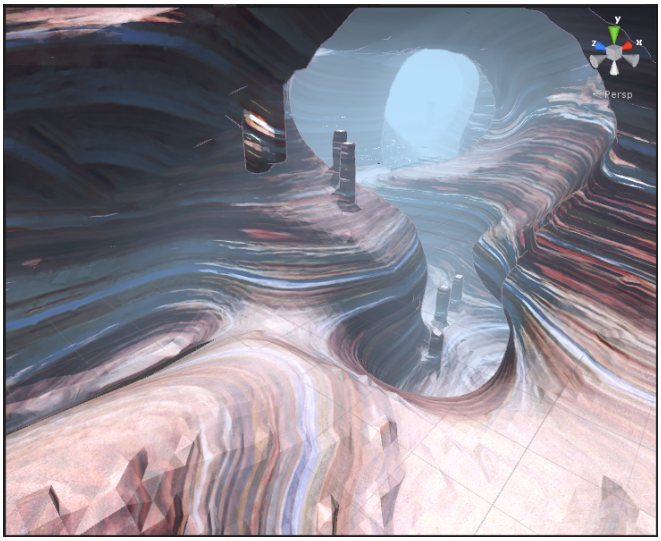


FIG. 90: Highly organic and perfectly smooth cave insides. Useful flow but lacking in sharp features.

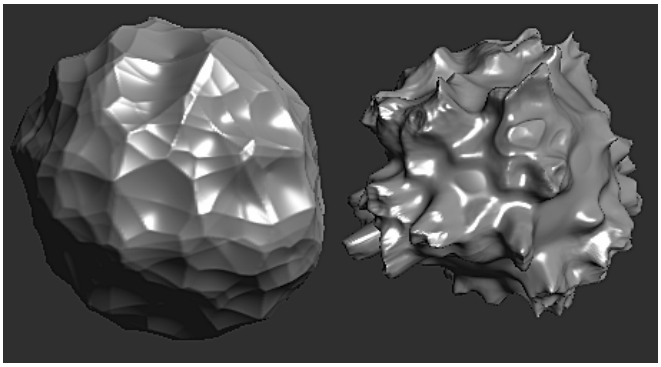


FIG. 91: A sphere distorted by Worley (Voronoi) noise (left), and a sphere distorted by a combination of Worley and Simplex noise[41]

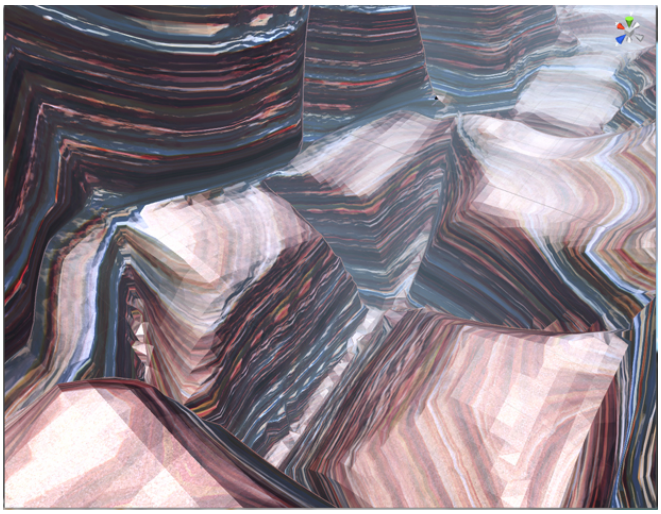


FIG. 92: Low frequency and high amplitude Voronoi noise parting the voxels into rift-like cells.

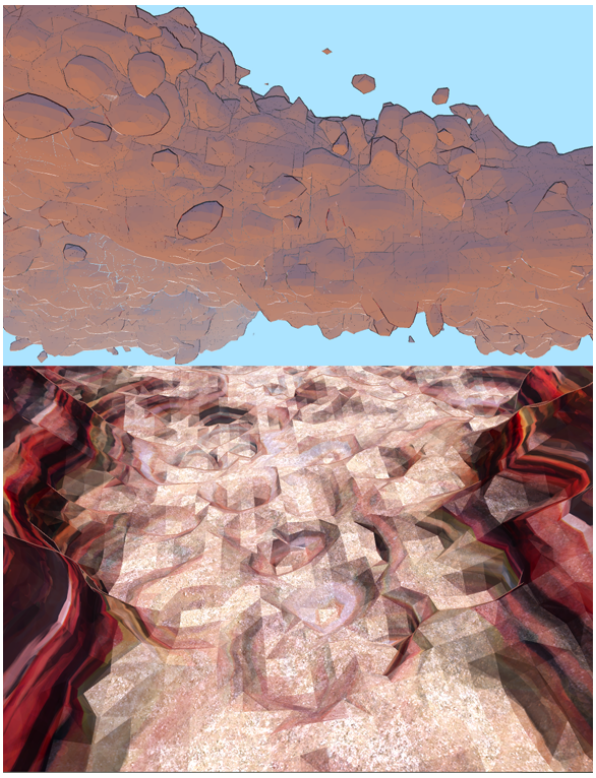


FIG. 94: The result of applying a layer of Voronoi noise, twice higher frequency than the smallest already used by the metaball warping equation. Regardless of amplitude, this layer is not desirable.

FIG. 95: Stalactites and stalagmites in a natural cave.

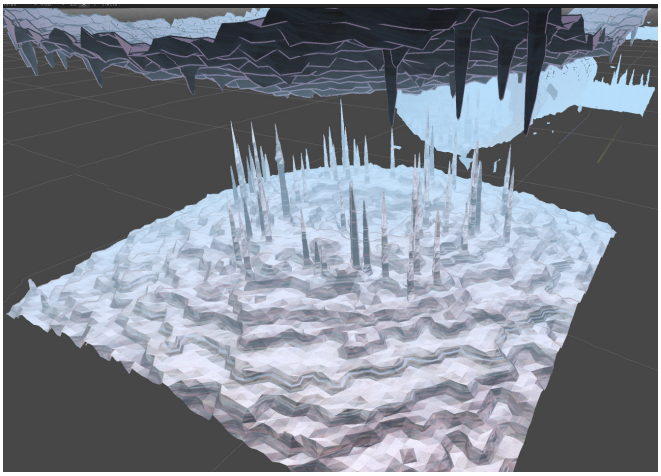


FIG. 96: Stalactite and Stalagmite distribution and construction test run.

namely stalactites and stalagmites, which are one column of voxels thick. Higher density stalactites, dubbed ”hoodoos”, instead consist of four columns of voxels next to each other, making them thicker and less rigid as seen on figure 97.
If a floor or ceiling is not found within the current tile, a feature will simply not be spawned in the direction where the edge of the tile was found.

After the anchor points have been selected and marked for expansion, they are grown with simple Cellular Automata. This automata checks if a voxel above or below it, is an anchor point, and if that is the case it itself becomes an anchor point and solid.
The length of each of these features is determined by the noise value of the initially chosen point as well as the distance of the anchor point from this initial point. This means that if the noise value of the initial point is large enough, a stalactite and a stalagmite can grow together to form a complete column. Furthermore the features are thinned out as they grow, by gradually increasing the noise value of each subsequent voxel towards zero, which can be seen on figure 77 on page 34.

While the feature spawning functionality is only used for stalactites and hoodoos in the current solution, it is possible to use the same selection process to procedurally place any prerendered (or procedural) objects in the caves. This could be used to spawn anything from rocks to enemies, by the noise-based selection method in conjunction with various

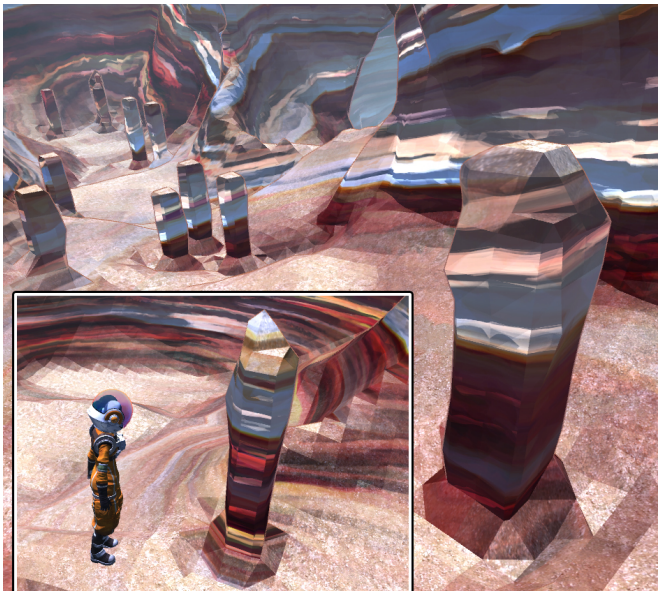


FIG. 97: Hoodoos are less rigid and their shapes are left more open to be influenced by the neighbouring volume of noise values.

heuristics to keep them from spawning in unwanted locations. Because the potential spawn volumes are dictated by a 3D noise function, they can be saved and reproduced at any time given the same noise seed.

The stalactites are tweakable via sliders (figure 98), though at this time they are not user friendly; due to the relative unpredictability of noise, and for reasons of scope, there are certain combinations of sliders which are known to cause erroneous results (exhibited in the form of mesh tearing or unrealistic distortion).

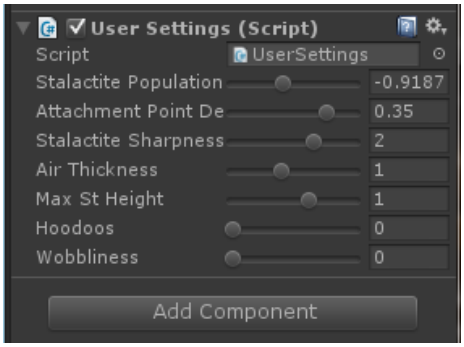


FIG. 98: Pannel in the Unity interface which exposes settings for our features. (In this particular case, configured to produce (sharp)Stalactites)

4) Technical Considerations

During the generation process, each tile starts out as a cube of 32^3 voxels. The choice of the number of voxels theoretically depends on the number of Compute Threads a GPU can spare while it also renders the scene. It is difficult to determine and explain what that number of threads should be, however. The ATI 7750 videocard, for instance, has 8 Stream Multiprocessors, each with 64 units, meaning a total of 512 Stream Processors. Each of them has a "Wavefront size": a block of 64 threads that reside on the videocard and can conceivably run concurrently for that Stream Processor, along with all the others. However, depending on memory requirements and the complexity of the operation, not all threads will actually be able to run the same kind of complex task, and typically, multiple cycles of the GPU are required to generate one frame [75]. Moreover, Nvidia's equivalent of a Stream Processor is a CUDA core, split into "Warps", which have 32 threads each, [74], and neither the cores nor the warps can be directly compared to ATI's "equivalents". The architectures are complex, different, and full of trade secrets; the way thread groups are managed, streamed and run is not straightforward. Therefore it is impossible to say that "a videocard can run x amount of threads in parallel, and we should aim to have y amount of voxels being processed on the GPU at the same time".

It is more useful for us to do benchmarks. The ATI HD 7750 is the lowest end gaming graphics card from the AMD 7k series, released in January 2012. We will do our benchmarks on this card, and whenever we mention a framerate or a number of seconds spent generating a tile, we will be referring to it.

64^3 voxels were arranged in a cubic grid, and then distance-based velocities (and colours) were applied to them as a test of processing speed. Figure 99 shows a random n-body simulation running at 135FPS. This proves that 262144 independent Compute Shader functions can be assigned to the hardware, and run at 135FPS, while the GPU also renders the $64^3 * 6 * 2$ (3145728) additively alpha-blended triangles which make up the procedurally generated displayed cubes.

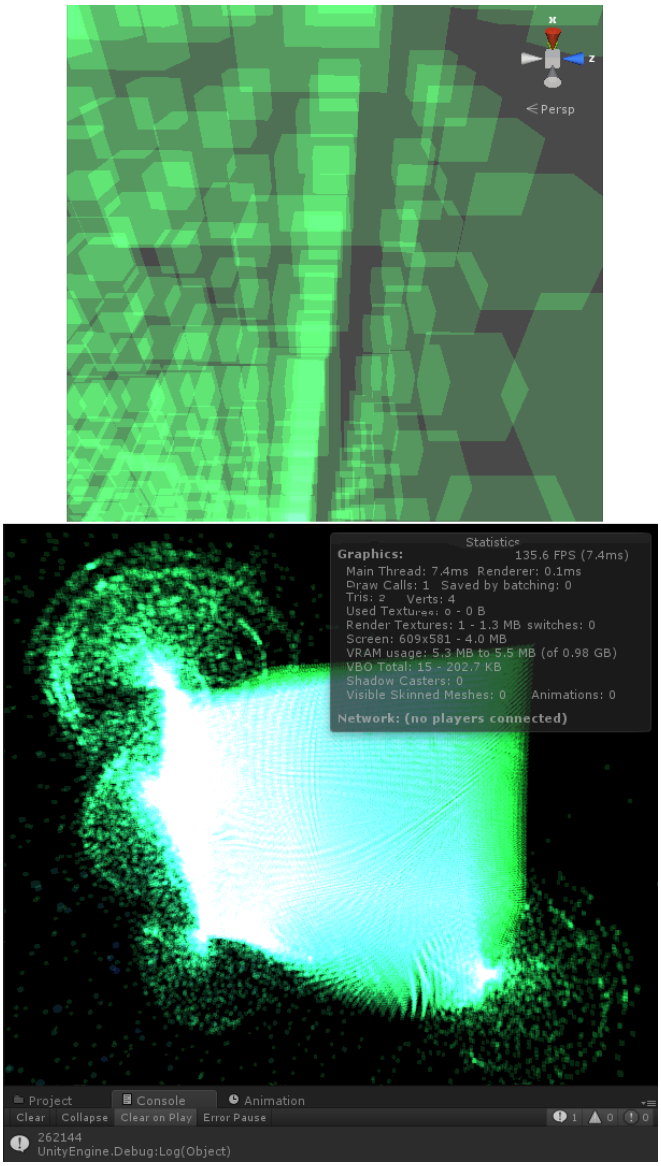


FIG. 99: Figure on the top is a zoomed-in screenshot showing that each voxel is represented as a cube. On the bottom is a screenshot of the whole volume of 262144 voxels.

The reason we chose 32^3 voxels however, was to guarantee that running Marching Cubes on that volume of voxels (while having noise in them) would not reach the 65000 vertices limit that Unity has for a single mesh. It would now be theoretically possible to process $2x32^3$ voxels at the same time, however,

this would require us to build a multithreaded application within Unity, which directly interacts with Unity components. Unity is not scheduled to officially support multithreading and to be thread safe until version 5.0 (as of yet unrealeased). We therefore leave this possibily for optimisations open for future work.

In section II-D.1 on page 17 we argued that the best approach to voxel data management for PCG caves would be a Point Region Octree structure. As we have shown in section II-D.2 on page 18, in order to handle voxel processing on the GPU, the octree structure must reside on the GPU and therefore must be maintained in a 3D texture. An issue that was discovered during the development stage is that the free version of the Unity engine does not support (allow) the use of 3D textures (and the 4096x4096 2D texel limit does not suffice). This essentially meant that there was no choice but to downgrade from the initial plan of using an octree, to using a virtual grid of 3D tiles, each being represented by a cube of voxels. This cube of voxels always points to the same 1D buffer residing on the GPU, and has no knowledge of its neighbours. The reason why the tiles always match, is that the functions applied to the voxels are based on noise generated from (seeded) worldspace coordinates.

Each Compute Shader layer which references the 1D array of voxels, holds an id of the current tile, its position and a list of the current tile's structural points. The 1D array contains the following data for each voxel:

- Its world position
- A noise value
- The previous noise value (Used to avoid problems when using cellular automata in parallel)
- A flag field (determines if the current voxel is to be ignored by the isosurface extraction, or whether it is a structural point, or is a spawn point for a stalactite etc.)

Each time that a new tile is sent to the pipeline, the initialisation stage sets each voxel's world position according to the current tile. This results in the observable blue 3D volume with a moving brush, which is teleporting around the game world "printing" tiles.

Another limitation triggered by the lack of 3D textures, is that it makes it impossible to store any procedural noise information in order to speed up the generation (and rendering) process. GPU Gems 3 [17] is capable of creating its detailed 3D terrain on GPUs from 2007, and the generation happens every frame, from a mixture of 10 noise functions. This is because instead of calculating the noise on the GPU for those 10 noise types, it instead uses lookup textures. Storing the results of a demanding noise function (such as 3D Curled Simplex noise) as a 3D point-cloud in a 3D texture would relieve a lot of strain from the hardware.

In contrast, our metaball is distorted by 6 separate noise function calls, all calculated on the spot on the GPU. In addition, our procedural texturing shader, needs to calculate 3 Simplex noise functions and the Curl noise procedure, for every vertex of the terrain, every frame of the game.
The advantage is that a computed worldspace-based noise function has infinite detail compared to a finite 3D texture. However, the finite volume can effectively be made exponentially larger by sampling from it in a random way (for instance, warping the lookup by one, simple, generated Simplex value).

As a closing statement to this section, we would like to note that due to these technical aspects, and also reasons of scope (ie not focusing on finding technical workarounds), our focus has been on researching our methods of modelling compelling caves, instead of fast runtime. Our technical demo is therefore aimed at the models, not at generation speed, coherence of tile distribution, and other technical aspects such as LOD.

We stress however, that our solution has no limitations that would keep it from being easily adapted to use 3D textures, and ported to an octree structure, given a Unity Pro license. At the same time we see no reason why, given enough technical knowledge, that our method could not also be ported to OpenCL or CUDA. We therefore also leave these aspects open for future work.

Even with the aforementioned inefficiency and overhead, we can output the following (on the ATI 7750):

- 831 cave sections were generated at an average speed of 0.228 seconds per mesh (˜7000 vertices per mesh).
- a total number of vertices of: 5.744.610
- a total amount of mesh space of: 213 MB

At the same time, a game world with all our meshes, the procedural texturing and an additional animated character model of 69763 triangles, is rendered at over 60FPS, at 1080p resulution on the ATI 7750, an ATI 5870, or an Nvidia GTX 660m. An unusually large open area inside a cave can cause a slight framerate drop (to ˜48FPS) due to the lack of LOD, and this overburdening will also slow down the generation process. As we discovered from the user tests shown in the Results section, people with high end machines (the videocards we have exemplified so far are all below average) experienced no slowdown at all neither during generation nor after completion.

D. Interactive Evolution

As mentioned in section III-B.1 on page 26, to increase the expressivity of the solution, it is possible to randomly generate production rules. This allows the L-system to generate a large variety of structures without any user input. This breadth of structures means, however, that it is possible to create caves which potentially do not fit the needs of the user. Furthermore the stochastic nature of the macros also means that the same production rule might take on many different forms, some of

which may not be desired.

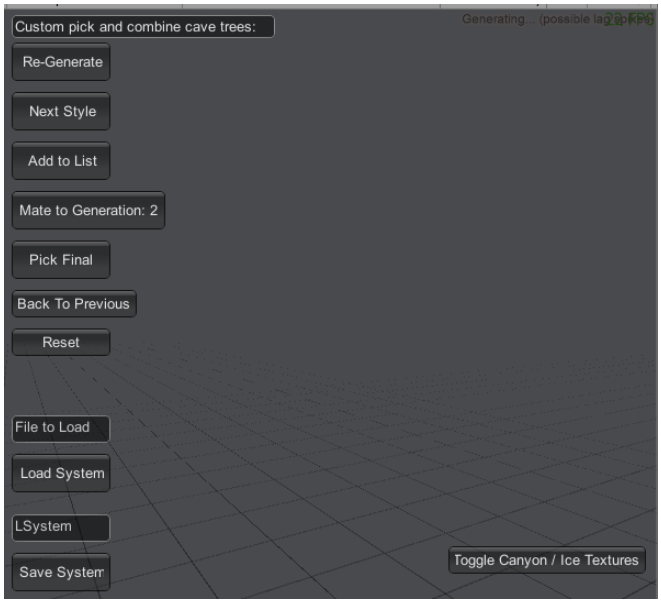


FIG. 100: The interface used to control the generation and evolution of caves

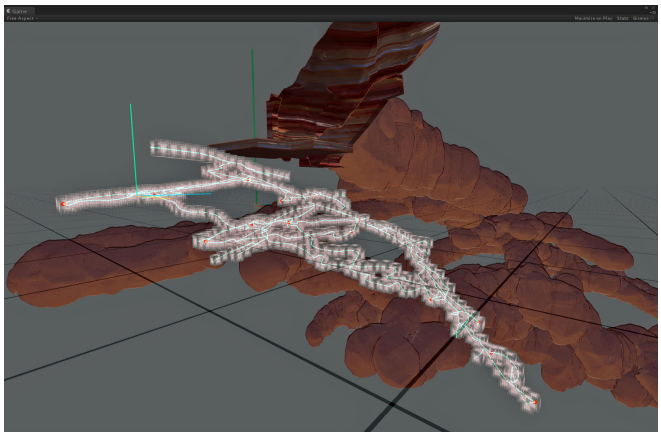


FIG. 101: The preview mode of our application simulates the shape of the final cave by illustrating the L-System structural points and the cave radius around them (white cubes).

To try and remedy this situation, an interface to control the generation and management of caves has been implemented, and seen on figure 100 and in figure 101, along with a system for interactive evolution.
The main use for the interface is to allow the user to generate more caves with the same production rules as well as ones with newly generated rules. This allows the user to quickly preview a lot of caves in order to pick the correct one for the current application. It is also possible to save and load caves from and to an XML file. Once saved to a file, a cave can be completely replicated, as the L-system is saved as a fully expanded and deterministic axiom along with any connected endpoints and the random seed that initializes any noise based generation.
Finally, it is then possible to pick candidates for evolution as well as move on to the next generation in the evolution process.

Due to the complex nature of cave systems, as well as no clear indicators as to what makes a good cave, finding a fitness function for an L-system that is representing a cave is not a trivial task. Furthermore, evaluating an L-system would involve traversing the final axiom of each individual in every generation, taking a heavy toll on performance. These two facts do not make traditional evolution a good fit for this type of cave generation tool.
L-systems are, however, uniquely suited to evolution, as their axioms can be treated directly as a genome and when working with bracketed L-systems, whole branches can be taken from one production rule and grafted onto another, in a way similar to the grafting performed on actual trees.
Figure 102 on the next page shows this process in action. (A) and (B) represent the individuals which are going to be mated together, and are drawn using the production rules seen in grammar (7).

$$\begin{aligned} Z &\rightarrow FF[RFLFRF[LFFFFZ]FZ]F[LFZ] \quad (A) \\ Z &\rightarrow FF[RFZ]F[LFLFRFFLFRFZ] \quad (B) \end{aligned} \tag{7}$$

By replacing the contents of a bracket in one rule with the contents of a bracket in another rule, it is possible to combine the features of the two rules into the newly grafted one. This is again shown on figure 102 on the next page where the spliced together L-systems, (C) and (D), are drawn with the production rules shown in grammar (8).

$$\begin{aligned} Z &\rightarrow FF[RFLFRF[LFFFFZ]FZ]F[LFLFRFFLFRFZ] \quad (C) \\ Z &\rightarrow FF[RFZ]F[LFZ] \quad (D) \end{aligned} \tag{8}$$

It is clear to see that these two offspring are combinations of their parent systems, with features from both clearly visible in both instances.

While traditional evolution would not work well with our solution, as mentioned above, the ease with which L-systems can be combined into new L-systems that share features with both of its parents, makes them an ideal candidate for interactive evolution. In lieu of an actual fitness function, interactive evolution uses the user as a fitness function to decide which individuals should be combined. In this case, this is done through the interface described above. It is possible for the user to pick L-systems to be combined later. As soon as two have been picked, it is possible to advance the interactive evolution to the next generation, after which every new L-system will be a combination of the ones that were picked in the previous generation. The user can simply keep doing this until a desired L-system is found, or go back to a previously picked L-system. When the user wishes to stop combining L-systems, it is possible to pick the current system as a final choice, in which case it will be saved and the interactive evolution will end.
The result of this evolutionary process is not always as clearly visible as above, due to the stochastic nature of the

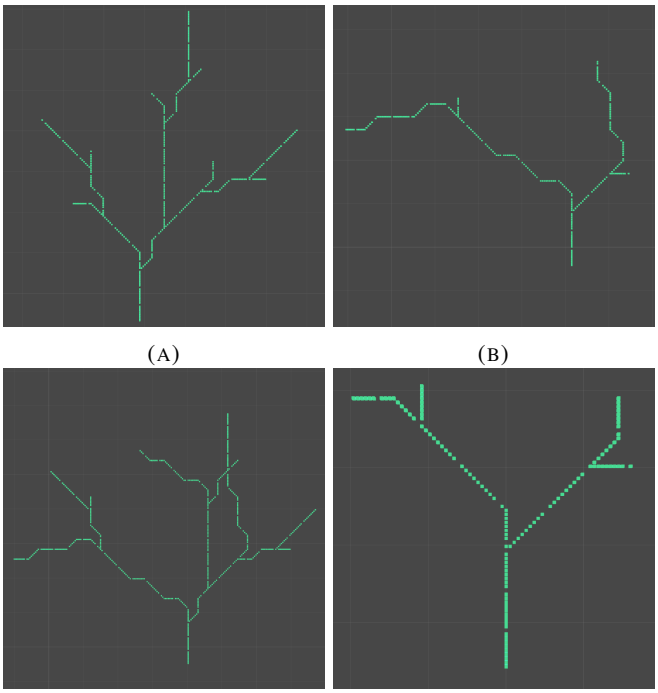


FIG. 102: An example of evolution in action, showing two individuals ((A) and (B)) as well as their offspring ((C) and (D)), all expanded twice

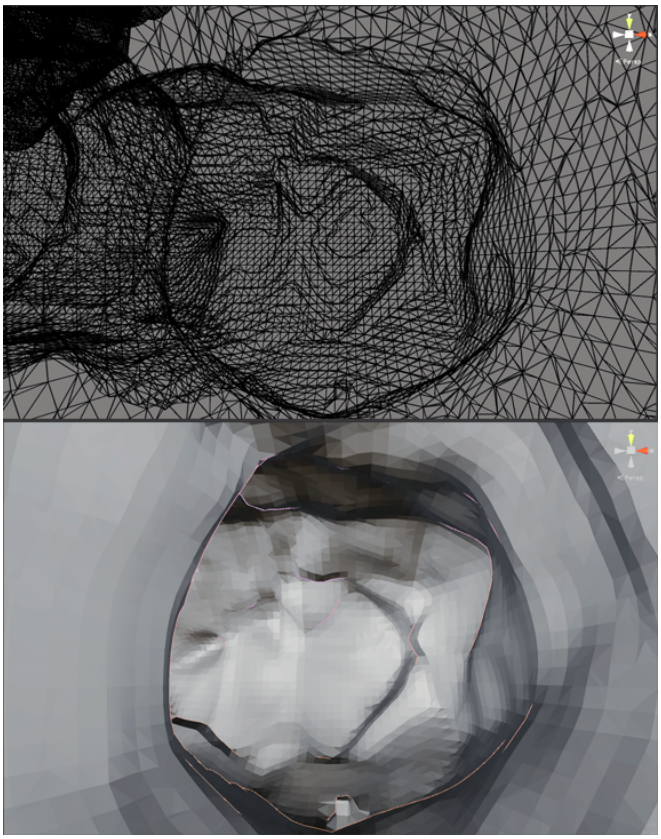


FIG. 103: Top: the mesh density of the Marching Cubes mesh. Bottom: the raw mesh with lighting and smooth normals.

macros, however aspects of both of the combined rules show up in the newly created one.

E. Mesh Extraction and Rendering

1) Isosurface extraction and smooth Normal calculation

As argued at the end of section II-E.2 on page 19, and coupled with the limitations outlined in section III-C.4 on page 42, a decision was made to employ the use of primal isosurface extraction methods, namely the classic Marching Cubes algorithm on our voxel data.

Figure 103 shows the results of marching cubes applied to the voxel data generated by our smooth metaball function (section III-C on page 33). We note that the depicted mesh's normals have been already smoothed with the method described in the following paragraphs.
As previously shown in figure 88 on page 40, the noise values within the voxels are perturbed by a moving smooth metaball. The smoothness comes from the fact that the metaball has a value of 1 (air) at its centre, and a value of -1 (rock) at its horizon. The transitional threshold between air and rock is at precisely 0, and every value between -1 and 1 is of floating-point precision. The isosurface extraction is applied only to groups of voxels which exhibit a change of sign (intersect the surface (threshold)).

Section II-E.2 on page 19 details how the algorithm works. As Marching Cubes leaves us with a mesh which has flat normals (perpendicular to the triangle face), metods must be researched to smooth these normals, as we elaborate in the coming paragraphs.

The practice in videogame development when it comes to obtaining smooth efficient meshes, is as follows. The artist creates a low-poly model to be used as the game asset, as having less triangles translates into better performance. A low resolution mesh however appears blocky: figure 104 on the next page. To alleviate this, an artist would also sculpt a very high definitiion version of the same model, and bake that model's surface normal information onto the lower polygon mesh. This effectively fakes the appearance of more detail than there actually is, due to the information extracted from the detailed version of the asset.

Because in real-time procedural geometry modelling there is no time to waste with generating a high definition version of the same mesh, different approaches must be explored. The classic solution, as described by Bourke [1], is to iterate through the generated geometry and average the normals of all the triangles sharing a vertex. This is however a naive normal smoothing approach which doesn't match or suit the topology. This is because there is no knowledge of how the mesh should be smoothed (ie there is no higher resolution model to compare to). Furthermore, the results make the mesh

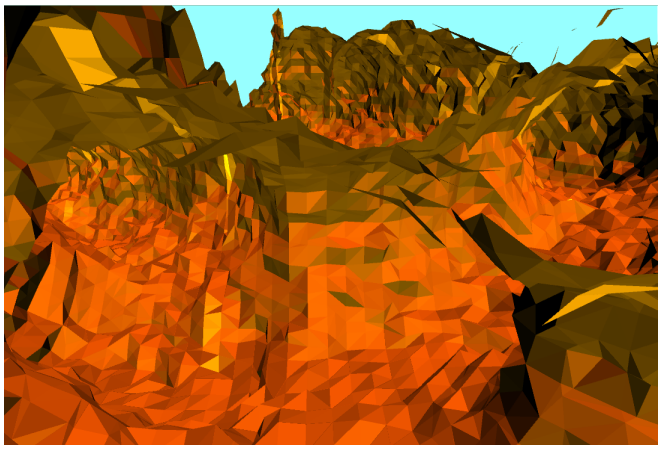


FIG. 104: An early attempt at a mesh extracted through Marching Cubes. No smoothing was applied to the normals.

appear to be vertex lit, even though it is pixel lit: figure 105. Attempting to further improve this naive method would mean

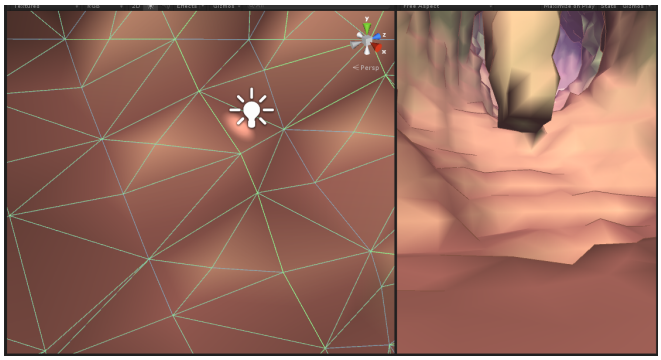


FIG. 105: Our early attempt: classic normal calculation by averaging neighbours. The naive method does not match or suit the topology. Notice that it appears to be vertex lit, even though it is pixel lit, as it can be seen with the small spotlight under the white lightbulb icon.

doing a distance-weighted normal blending calculation over a wider surface (a larger amount of neighbouring vertices) around the current vertex. It is not trivial, however, to take a flat 1D array of vertices from the generated mesh, and then quickly figure out, for each vertex, which are its x closest neighbours and then fetch their normals for weighted smoothing.

A better approach to this problem would be to attempt to fetch the higher-detail mesh information from the voxel data itself (or more broadly, from the procedural generation function). Bourke also points to a solution presented at SIGGRAPH 1987, which computes the normals by interpolating the values at the corners of the cubic voxel grid which encompasses the current vertex, relative to the vertex position, as well as from the values at the relevant neighbouring voxels, as seen on figure 106.

The solution employed for our normal calculations is similar to the one above. The exception is, however, as is pointed out in GPU Gems 3[17], that doing a grid-neighbour

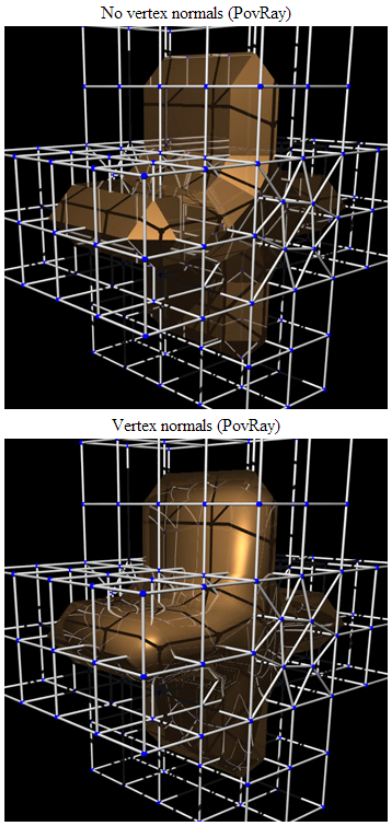


FIG. 106: Top image shows a low-poly model with flat surface normals. Bottom image shows the same model, made to look very smooth by calculating vertex normals from the grid corners (blue dots).

lookup within all 26 neighbours of the current cubic-grid cube and processing all of the relevant corners to calculate the smooth normals of the current vertex, is slow and unnecessary. GPU Gems suggests an alternative solution which computes a good quality normal, a normal relevant to its surroundings, from just 6 neighbouring voxels. In short, the x coordinate of the normal vector is calculated from the density difference between the neighbouring voxel at the x-1 coordinate, and the voxel at the x+1 coordinate. Similarly y and z are computed. Essentially, it bends the default perpendicular normal of the vertex, depending on which of the neighbouring voxels are ”air” and which are ”ground”.

The results of applying this method to our extracted meshes can be seen in figure 107 on the following page. It did not appear to be well suited for a low resolution mesh, particularly at transitions between faces with slopes of higher angle differences.

Instead of attempting to calculate more precise normals to improve the result, which would have considerably decreased performance, we sought to further smooth the mesh normals on a different scale. By averaging the triangle face normal (the default perpendicular normal) of each triangle in the existing 6-neighbour lookup in our grid, and blending the result to the smooth normal we are calculating through the

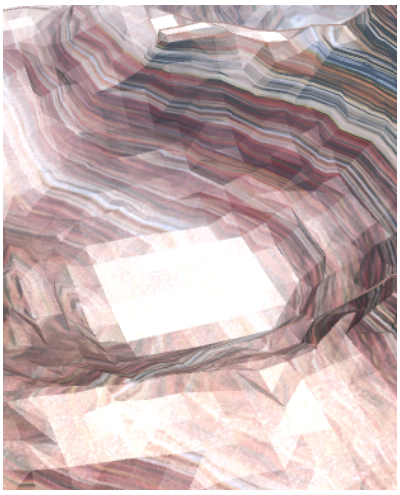


FIG. 107: Implementing the normal calculation method suggested in GPU Gems 3. It is not smooth and precise enough for a low resolution mesh.

Gems approach, we noticed a remarkable improvement in mesh smoothness, without adding any extra iteration steps (ie fetching more than 6 neighbours for normal computation).

The final results can be seen in figure 108. To prove that our methods are in line with the state of the art of real-time PCG in the industry, we have compared this figure to screenshots of terrain from No Man's Sky (2014)[82].

There evidently is a compromise to this approach of only fetching normal information from up to 6 grid-aligned neighbours instead of 26. For the less common cases in which the polygon has most of its relevant density neighbours not among those 6 voxels (e.g the top of a diagonal and thin wall), the generated normals will be less accurate, or closer to the default perpendicular normal.

FIG. 108: The normals and general smoothness of geometry compared to screenshots from No Man's Sky [82]. Beyond tessellation, there has been little done in the industry to produce primal marched terrain in real-time which is smoother than this.

2) Procedural Canyon Material and Triplanar Projection

As reviewed in section II-E.3 on page 23, triplanar projection must be used on procedurally generated meshes, as they are incapable of providing UV information. In the same section, we described the benefits to the technique of Solid Texturing, which is what we have used to create a coherent, three-dimensionally sample-able canyon rock material. 3D procedural noise is very powerful when sampled as a 3D volume in world-space, as it gives the material the appearance of a carved solid object, such as a wood sculpture.
The choice of creating a canyon material was made due to the fractal nature of the canyon rock stratification, as seen on figure 109 on the next page. This meant that we could obtain a high quality and compelling material without the need of much "artist work" put into painting textures for example. All that was required, were a few base textures which featured parallel horizontal lines of various orange shades, after which the 3d Curl noise function could provide complex information for perturbing the parallel lines.

The implementation of a triplanar projection scheme required us to write a full Lambert lighting model shader from scratch. Every element in the shading procedure needed to be custom, the data structures, the procedurally warped textures, the custom normals and bumpmaps. The way they all must be blended together and included into the final lighting equation meant we could not make much use of helper functions or pre-written shaders.
The shader is implemented in CG, ("C for Graphics") a common shading language developed by Nvidia and Microsoft, compatible with both the OpenGL API and HLSL.
We feature forward-rendered, multi-pass, per-pixel, lighting of all dynamic types (directional, point, and cone). We however have not implemented shadows for this project (and Unity Free also does not allow most shadows).
This material shader also features variable-thickness outlines, triplanar texture projection and triplanar bumpmap projection which are perturbed in real-time by Curled Simplex noise.

The final shader takes three simple textures as input for texturing the walls, and the floor and ceiling. A similar second set of textures provides bumpmapping information. In order

FIG. 109: Mini imageboard of fractal phenomena exhibited by natural canyon (cave) stratification. Mammoth Cave National Park, Antelope Canyon, Grand Canyon.[105][103].

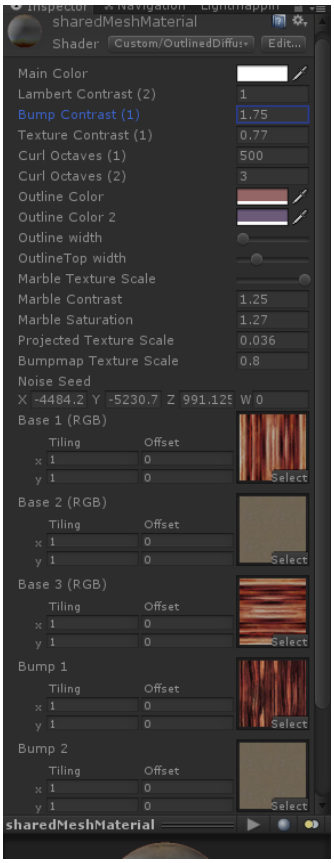


FIG. 110: The parameters exposed by our procedural triplanar projection shader. Communication between the GPU and the CPU can be done through these parameters, e.g. setting (or animating) the initial Seed for the noise.

to produce the canyon-like bended texture in the results, the textures were curled and combined at different scales. A third component, listed as "marble" in the shader, is actually a copy of the base texture, but scaled up (based on a slider) and blended to the final mixture of textures and bumpmaps in the fashion of a classic "Overlay" image filter:

$$1 - 2 * (1 - base) * (1 - marble) \tag{9}$$

where the variables are normalised to $\{0,1\}$. This allowed more complexity in the texture, and more importantly, alternated between very thin horizontal stripes, and very large, thick, stripes to provide multiple scales of detail. This can be clearly seen in figure 117 on page 51, on the lower versus the upper half of the column.

All of the input parameters of the shader can be seen in figure 110. The shader is highly customisable; it can dynamically alter any of its parameters, like the frequency and amplitude of the noise and its initial noise seed (also used on all other noise operations in our application, allowing us to save a specific pattern of cave). Since this seed value can be animated, it can be used to warp the texture in real-time, which is useful for simulating reflection and refraction for an ice material for example.

The post-work done by our procedural shader improves the perceived shape and resolution of our geometry considerably. The following figure shows versions of the same scene, with the meshes having parts of the triplanar projection removed: figure 111 on the following page. The material provides an important layer of finer detail which enhances the believability of the environment. Its impact has been tested and observed in our user survey (section IV-B on page 52).

Figure 113 on the following page shows that the properties of Curl noise complement the organic flow of the terrain, providing the illusion of a tilted canyon formation, or otherwise simulating the way natural canyon rock stratification is modulated in a similar fractal manner, as it was shown in figure 109. This element is a very important step to allow the user to suspend disbelief and to enhance the variation in a video game environment.

The range of features of the Curl noise also exhibits regions of unusual behaviour in our textures. Knots in the Curl noise function's output present emergent aberrations on the cave walls, as seen on figure 116 on page 50. The low amplitude of the noise ensures that these distortions does not become exaggerated, but remain believable.

The noise-based method of distributing stalactites throughout the world presented in section III-C.3 on page 38, can also be extended to the procedural material. Therefore, the final feature of the canyon material is that parts of the solid 3D volume described by the Curl noise function, are marked as regions saturated in minerals, as seen on figure 112 on the following page. Within these marked volumes, a simple equation is applied by the fragment shader, which blows up the contrast of a small fixed color range from the final

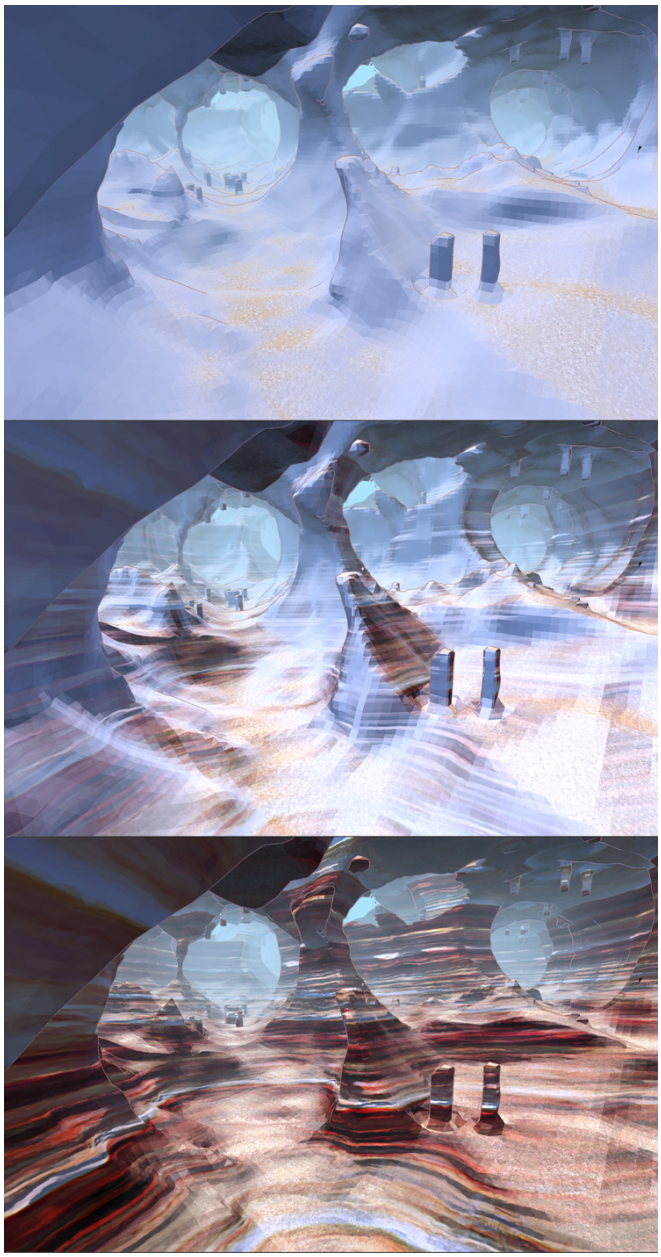


FIG. 111: The three images above show the same scene with progressively more layers of texture projection. Top image only has the Y coordinate texture projection on (ceiling and floor). Middle image also adds the X coordinate projection. Note the smooth blending between projections.

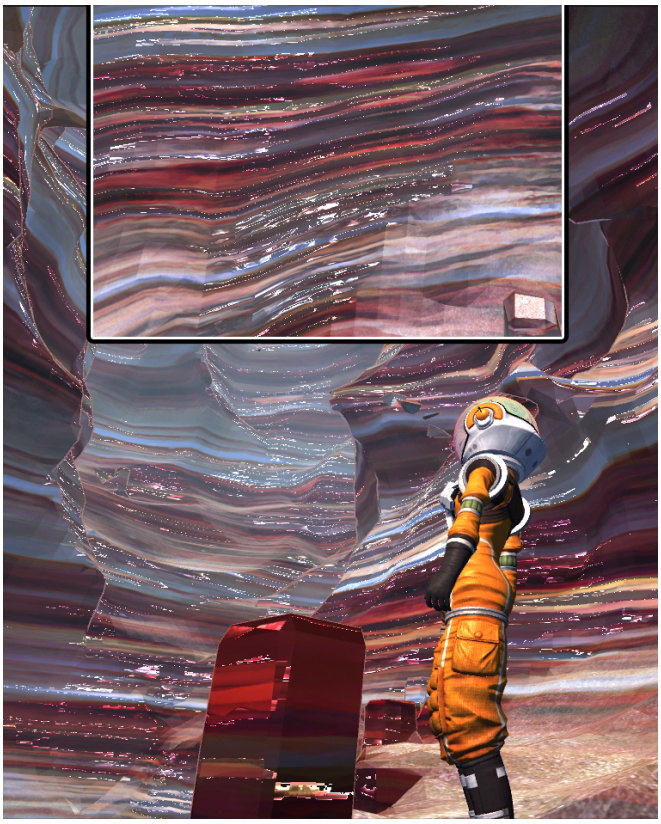


FIG. 112: A rare region which exhibits a ”sparkling minerals” patch in the procedural texture.

FIG. 113: Canyon stratification shader. Note the tilt and low scale curl. (Curl Noise)

fragment color.
This is a simple demonstration done to show that the procedural texture has the potential to be further varied, and that specific effects or decals can be spawned at certain locations.

Our procedural shader and its assortment of exposed parameters proved to exhibit remarkable versatility. It allowed us to easily derive an additional material, used to provide an alternative to the users evaluating our application. This second material was ice, and was created purely by tweaking the parameters of the existing canyon material. The frequency of the Curl noise was widely increased, and its amplitude decreased, in order to simulate tiny random reflections and refractions found on glacier walls, as seen on figure 115 on the next page.

The random seed of this material was also animated in real-time based on the player character’s position in the world and the dot product of their view angle, animating and further enhancing the reflection and refraction effect.
Further settings such as the Lambert contrast, the Bumpmap contrast and texture saturations were edited (their values were

flipped), in order to obtain the blue colouring of the material, without even having to change the textures. A sample of the result can be seen in figure 114.

FIG. 114: Our procedural shader with its parameters proves remarkable versatility: ice texture created without even changing base (orange canyon) textures.

FIG. 115: Texture on the walls of a glacier cave [104].

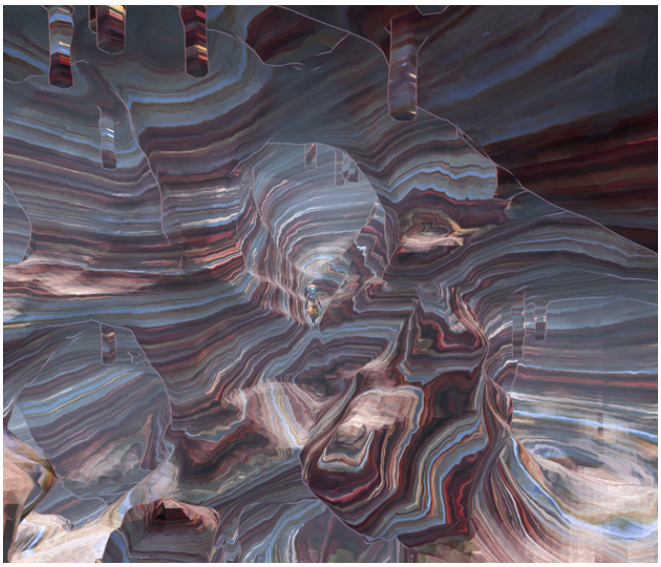


FIG. 116: Curl noise exhibits emergent aberrations. Desirable, especially in a procedurally generated game.

F. Collecting User Feedback

In order to better gauge the value of our solution, user feedback was collected from a number of random people. The users were prompted to play in a version of our cave generator running via the Unity web player and then asked a number of questions about their experiences. These questions are shown on table IV on page 52.

The tested application was in the form of a simple exploration game. The user started the game on a platform in the from of a pre-rendered cave entrance, from which point on the caves would generate. The main character was a third-person astronaut, which could run around and had a jetpack-like jump boost. The second player character was a space ship, which had the ability to roll 360 degrees on its forward axis (this was part of its steering mechanism), thus allowing for the testing of the caves from all possible orientations. The users were also encouraged to press a button which would switch in real-time between the canyon materials and the ice materials. This was done to compare the effectiveness of each material, both in relation to each other, as well as in relation to whether they improved on the mesh geometry.

The survey was designed to get a general idea of the user's opinion of the tool, as well as an idea of what kinds of applications they think our results would be suited for.
In order to reach out to users, our questionaire[10] was posted to the following websites:

- http://www.reddit.com/ (To various subreddits)
- http://forum.unity3d.com/
- http://www.gamedev.net/

[10]https://docs.google.com/forms/d/1_H9_oGVC9r0zN-Z8kkneQrhx4lgtDImn32ggXCv0c3o/viewform

- https://www.facebook.com/groups/game.students/ (Games Students at the IT University)
- https://www.facebook.com/groups/234823394212/ (MTG at the IT University of Copenhagen)

The users were asked to play through the generated caves, and provide feedback via the questionaire.
The questionaire can be found in the accompanying files as: SurveyPage.htm.

IV. Results

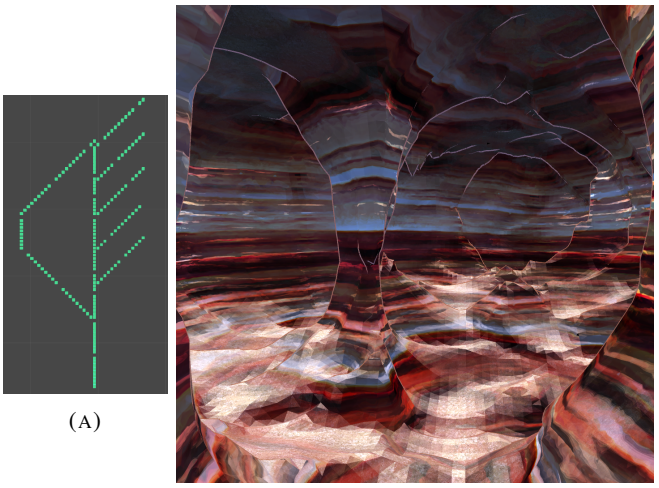
Evaluating a content generator is somewhat problematic. The end result, emergent procedural models, are often hard to describe and gauge in anything but a subjective manner. This is witnessed in literature, where many approaches to content generation are simply described and presented, and only evaluated briefly or not at all. At the same time, Ebert's, Musgrave's, Peachey's, Perlin's and Worley's book on procedural modelling and texturing via noise [35], is full of subjective terms such as "really beautiful", "really cool", "looks great" when it comes to describing their achievements. This book is found quoted in various academic work that we reviewed, as a benchmark comparison to the state of the art simulations. However, there is obviously no scientific evaluation function to quantify that "beauty".

To get around this problem, our content generator will be evaluated in two different ways.
Firstly, the emergent shapes of caves and the features that result from the metaball interacting with the Structural lines will be presented and compared to similar structures in nature. This will give an idea of the expressive range and believability of the generated caves.
Secondly the results of the user evaluation will be presented. This will show if the players, at which the generated content is targeted, find any value in the final product of our generator. Overall these two metrics should show if the goals mentioned in section I-B on page 2 have been achieved.

A. Expressive Range of the Generator

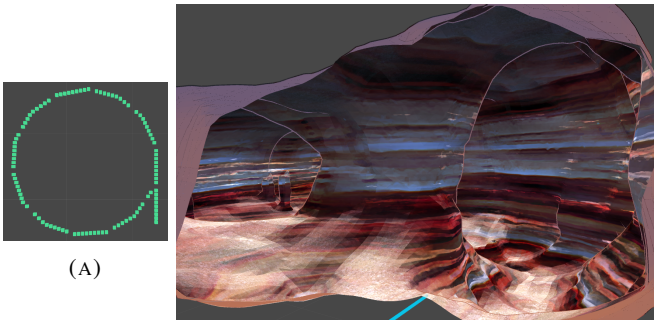
1) Interactions between L-system and Metaball

Section III-C.2 on page 36 explained what shapes the metaball can take, which can be seen in the experimental results of figure 93 on page 39. In this section we illustrate how the main source of interesting features and variety in our solution comes from the interaction between the L-system and the metaball. In particular the metaball can be manipulated to create various specific structures and features by using a series of patterns of structural lines that exploit the morphing volume of the metaball, and its size.
An example of this is shown on figure 117, where a room with a small pillar is created by concatenating many metaballs together with the pattern shown in (A). This creates an open



(A) (B)

Fig. 117: A room with a small column, created by exploting overlapping metaballs.



(A) (B)

Fig. 118: A large pillar, created by using a circular structural line

room with interesting features which emerge when the outer halves of the metaballs overlap. Natural looking columns, such as the one seen, can emerge when there is a spot in the structural pattern that is just barely out of reach of any metaball, resulting in a clump of solid terrain being left alone.
Another example of this is seen on figure 118. Here it is shown how a structural line forming a complete circle can create large pillars.

Tthe alignment of metaballs can also be exploited to create large scale landmarks such as bridges and ravines. Figure 119 on page 53 and figure 120 on page 53 shows two examples of how two structural lines on top of each other can interact to form these interesting features. Figure 119 on page 53 shows how two lines that diverge slightly can form a tall passage that splits in two when the metaballs stop overlapping, while figure 120 on page 53 shows the formation of walkways and overhangs when the two metaballs on top of each other only intersect slightly. These tall vertical structures create a good frame of reference for scale, and describe overarching large scale patterns in larger chambers, as seen in section IV-A.2 on the next page further below.

Q1	Do you play games with procedurally generated content?
Q2	Overall, how well did the shapes in our caves, match the reference photos above?
Q3	How natural did the canyon erosion look?
Q4	Albeit stylized, how much did the texture help sell the idea that this is some kind of canyon environment?
Q5	Did you try the ice texture? (pressing J)
Q6	Was the ice texture better or worse suited for the cave geometry?
Q7	Regardless of realism, how much did you like the shapes and art style?
Q8	How much time do you estimate you have spent playing around in the caves?
Q9	How interested would you be in playing a game which used our caves as (part of) its environment?
Q10	How well suited was the environment for a 3rd person character?
Q11	How well suited was the environment for a spaceship/flying game?
Q12	Did you find one of the two textures better suited for a certain play style?
Q13	What kind of game would you think our caves would best be suited for?
Q14	Did any of the environments / textures remind you of something (a game)?
Q15	What would you like to see added?
Q16	If you ran the program more than once, were any caves unsatisfactory / not fun to play around in?
Q17	Did you lose your way?
Q18	Do you have any general comments to add? Was there something in particular you liked / disliked?
Q19	Did you play until the entire cave finished generating?
Q20	Did you experience any lag (before and/or after) the cave finished generating?
Q21	Would you like to identify yourself?
Q22	What is your background?

TABLE IV: A list of all the questions that was asked the participants

The vertical element of the L-system can also create more low key features than massive rooms. Figure 121 on page 54 shows how a slightly raised structural line that intersects with a larger structure can create a raised walkway, and evoke an image of a dried out underground lake.

These examples of interactions between the L-systems and the metaball, as well as many others like them, are what creates the interesting and emergent features of our caves, which are shown in the next section. They also represent what is leveraged in the currently used macros to create, for instance, rooms with pillars and riverbed patterns via the Q macro.

2) Emergent Complex Features

Throughout the description of our methodology, we have already compared images of caves, canyons and ice textures from the real world, to our reproduction of the features exhibited by these elements.
Additionally, we have shown the specific results exhibited by a set of deliberate structural configurations. We now showcase the fact that more advanced landmarks can emerge from more complex structural configurations.

Figure 122 on page 54 illustrates our system's capabilities of reproducing realistic overarching riverbed networks throughout a cave, as seen in the top image of the figure. The centre image shows an island on a dry lake bed. More unexpected emergent landmarks can also appear, such as the mini underground mesa with an arch dug under it, displayed in the bottom image of the figure.

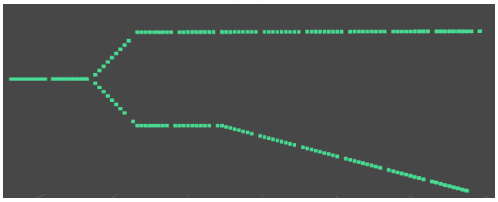
More instances of this emergent behaviour are demonstrated in

The Marching Cubes algorithm can exhibit occasional flaws. Our relatively low voxel resolution coupled with a very sharp transition in the noise, can result in floating geometry such as the thin columns in figure 123 on page 55. As Yurovchak [44] pointed out, the problem can evidently be eliminated by increasing voxel density, however, this is a brute-force solution which would decrease rendering performance (as it results in a higher polygon count). Cui[22] used a flood fill algorithm to eliminate potential floating geometry at the voxel level. This would however effectively lower the detail of the geometry even further. At the same time, an additional pass through the entire geometry is not desired in real-time content generation. We argue that the only option is to use a Dual grid isosurface extraction scheme, which, being based on an octree, would allow finer grid levels at regions of higher density (as reviewed in section II-E.2 on page 19). Otherwise, we have no choice but to recommend this error to be embraced as part of the art style.

B. User Survey

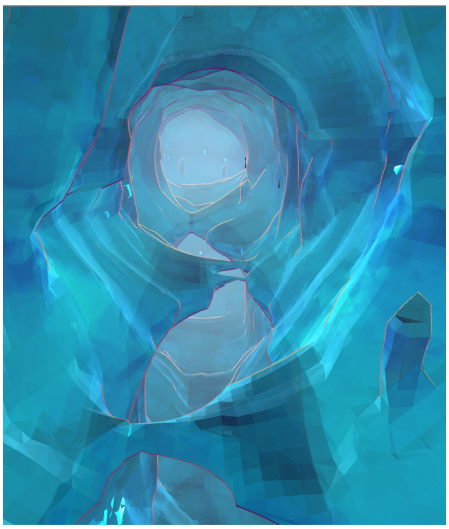
As described in section III-F on page 50, data was collected from users to gauge how well our solution would be received by a player. In total 30 users responded to the questionnaire, 26 of which had some sort of background in the software, fine arts, or gaming industry. Most also

(A)

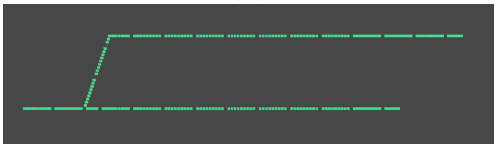


(B)

FIG. 119: A tall tunnel that split into two, created by having two structural lines slowly diverge. The L-system(B) is seen from the side



(A)



(B)

FIG. 120: Walkways and bridges in a tall passage, created by having two structural line run parallel at a distance where they only overlap occasionally. The L-system(B) is seen from the side

Question	Negative	Neutral	Positive
Q2	3	3	24
Q3	2	8	19
Q4	1	5	24
Q7	0	5	25
Q9	3	6	20
Q10	3	5	21
Q11	8	11	10

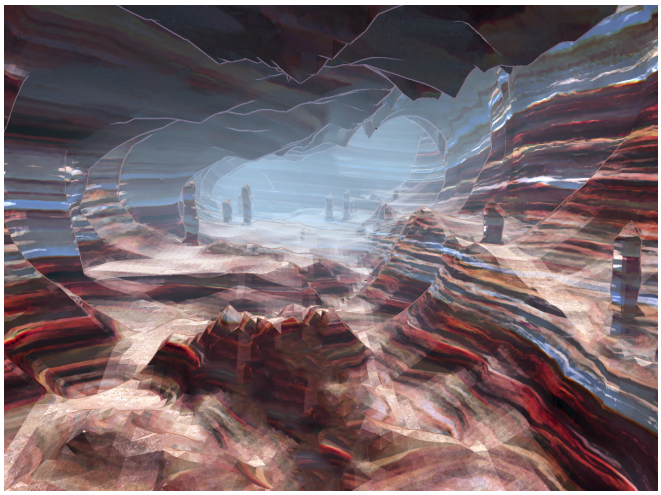
TABLE V: The sentiment of the answers given when the users were asked to rate various aspects of the caves. For the description of the questions, refer to table IV on the previous page. Answers were given on a scale from 1 to 10, with 1-3 being considered negative, 4-6 neutral and 7-10 positive

claimed to play a medium or high amount of games with procedurally generated content. Five of them claimed they only rarely play games with PCG, and one of them claimed to never do so. When asked if they play such games, they scored an average of 2 on a scale from 0(never) to 3(a lot). In this section an overview of their feedback will be presented. For the full results, see the accompanying spreadsheet. See the accompanying spreadsheet: UserSurveyResults.xlsx.

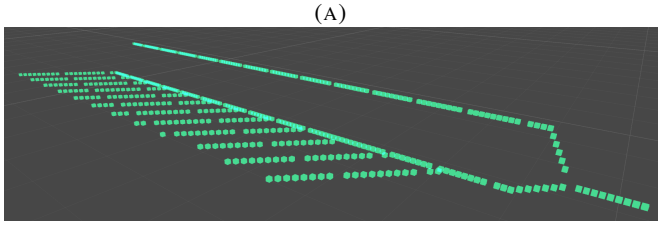
Overall the feedback that was received was very encouraging, with many positive answers recorded. As seen on table V, the answers to questions where the users were asked to rate various aspects of the caves on a scale of 1 to 10, were generally extremely positive. The only outlier was when users were asked how well the caves were suited for a spaceship game(Q11), where the users gave our solution an average score. This result is most likely caused by the fact that our spaceship did not have the control scheme of a standard Descent style game, where the mouse controls the facing, and the keyboard controls the speed in each direction. Instead the facing was controlled with the keyboard and the mouse wheel was used as a throttle to control the speed. This may have caused players that are not familiar with this type of control scheme to feel that the caves were not suited for flying. The controls of the ship and its associated camera also reportedly performed sluggishly (which was a deliberate feature intended to alleviate possible nausea).

The most positive response was given when users were asked if they liked the art style of the caves(Q7), with the most picked response being the most positive (10). This is a surprising response, as the art style that is employed in our solution is mostly a proof of concept and the detail and fidelity of the caves can be improved if more craftsmanship is employed, as well as if displacement mapping (tessellation) is added to the stratification. The answer is however in line with the fact that many games on the market are very popular despite employing a low fidelity art style, such as Minecraft, ShovelKnight and Journey among others.

Most users generally did not spend a very long time playing around in the caves, however. Two of them could not run our



(A)



(B)

FIG. 121: A room with a raised walkway (by the right wall), created by having one structural line slightly higher than the rest of the structure.

demo at all, due to their machines not supporting DirectX11, and answered the questions based on a large screenshot gallery instead. Of the users that could run the demo, 14 reported spending between 5 and 15 minutes exploring it, with 6 users claiming to have spent less than 5 minutes and a single user spending 30 minutes. The rest did not specify how long they played. Unfortunately there was no way for us to specifically time the users, and there might have been a disparity between their self-reported, perceived, approximate length of time, and the actual time they spent playing.
According to the responses, there were three reasons that users did not spend longer playing around with the caves:

1) The caves generated too slowly, causing players to either accidentally fall out of caves or not wanting to wait until the next section finished generating, commenting on how the caves are not spawned in the direction they are walking.
2) They expected a game, but there was no gameplay, resulting in boredom.
3) Performance issues made their experience un-enjoyable. This was reported mainly by users running laptops with low end mobile graphics cards.

These reasons match up well with the other responses. When asked if they played until the entire cave finished generating, only 4 users said yes. The rest gave up either because the generation took too long, the cave was too big or they got bored. Additionally 7 users reported having performance

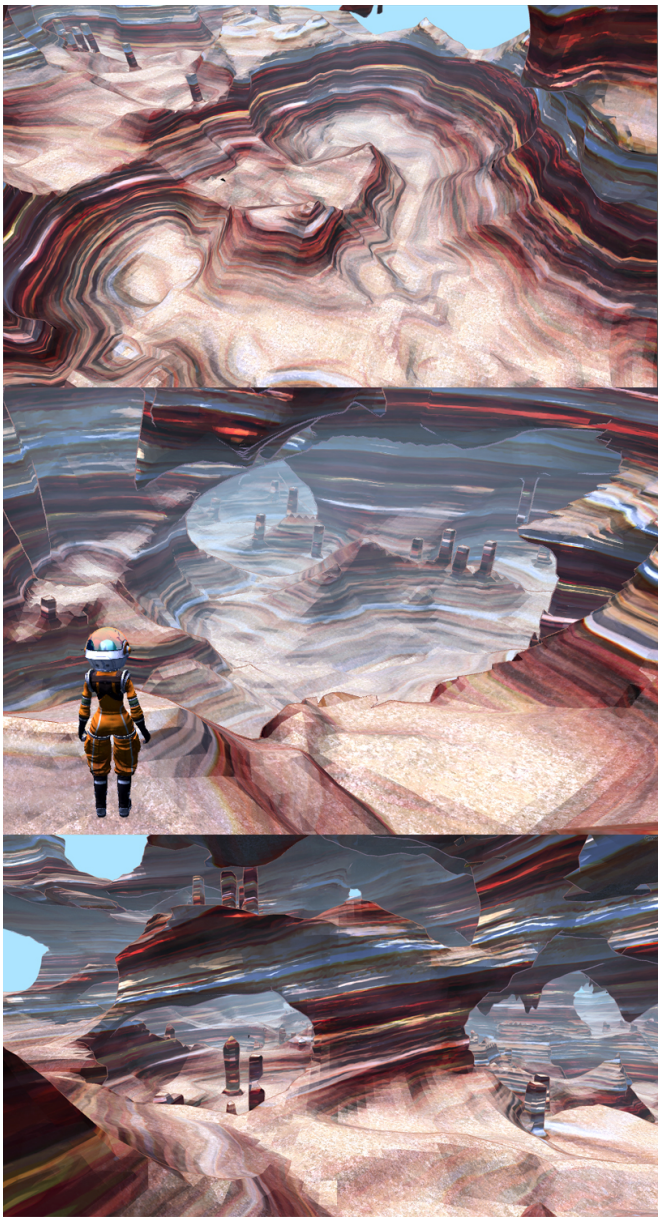


FIG. 122: Three instances of emergent landmarks and water-erosion simulation done by our procedurally generated combinations of stuctural points and metaballs.

issues while running our demo on various hardware. The performance was also the most common complaint when people were asked to give additional comments, with 4 people mentioning it.
These problems might have been the reason that most users did not seem to test more than a single cave, with only 3 users reporting to have tried multiple cave styles. This means that we do not get much feedback on the expressivity of our demo.

When asked what kind of game the caves in our demo would be best suited for, 10 users answered that they would like to see this kind of caves in an exploration focused game, and 4 users answered that they would like to see a mining

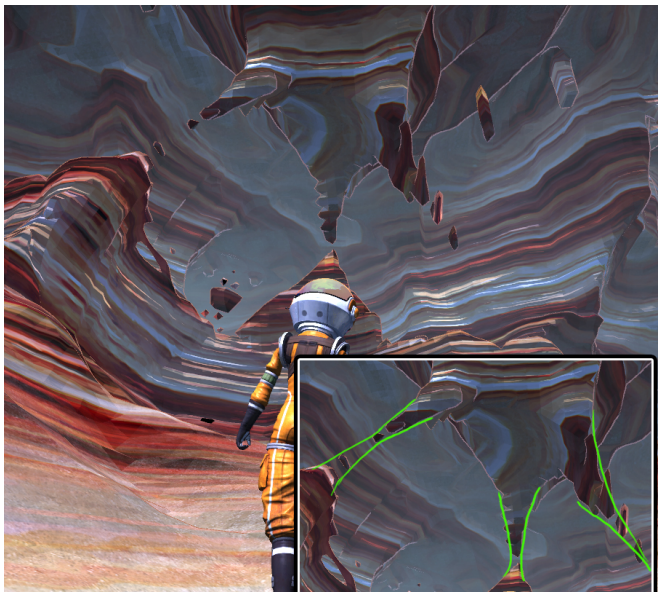


FIG. 123: Consequence of Marching Cubes precision on low resolution voxels. Green lines on the bottom-right illustrate the actual shape of the voxel information.

game. Other types of games, such as a third-person shooter, a canyon pod-racing or Descent style flying game, first-person exploration, 3D puzzle-solving, and an RPG, were mentioned. Interestingly though, when the users were asked if the caves and their style reminded them of anything, most people noted that they could not think of any games with similar environments. This shows how little procedurally generated underground landscapes have been tackled in the games industry. The items that were mentioned, were Minecraft, Waking Mars[107] (which is 2D), Gunvalkyrie[106] (for the canyons), Half Life 2 and Star Wars (pod-racing).

The users seemed to respond well to the alternate animated ice style texture, with 14 users liking it, 5 users not liking it and the rest being ambivalent. Users generally did not prefer one texture over the other, and scored an average of 3.142 on a scale of 1(canyon style) to 5(ice style) when asked which one they preferred.

Finally, when they were asked what they would like to see added, the main feature that users wanted was more variety in the caves with 6 users mentioning this. Some users gave a clearer explanation of what they meant, the main points being about the variety in the shape and size of the tunnels, pointing out the lack of very narrow or very tight tunnels, and the constant appearance of the textures (ie colour scheme). It is also possible that this perceived lack of variety could be caused not only by the actual environment, but also by the lack of objects that would generally be present in a cave in an actual game, such as rocks, vegetation, loot, buildings, or inhabitants. Our caves are also brightly lit to allow the users to best see the structures generated within the cave. In a game environment, the lighting would also vary, providing darker areas, torches etc.

Some users also wanted higher fidelity visuals with 3 users mentioning this.

V. Conclusions and Discussion

In section I-B on page 2 the overall goals for our cave generator were outlined, and as shown by our results above, we can conclude that this goal has been reached. The main properties of expressivity and believability, have been achieved in our landscapes, the results showing that the caves exhibit a wide range of shapes and structures. At the same time, they also share the same characteristics as natural cave patterns, enough to maintain immersion in the context of video games. In this section, conclusions will be drawn from the user feedback that was received, the viability and applications of our cave generation tool are discussed, and possible future improvements are presented.

A. Conclusions on User Feedback

The user feedback was very helpful in identifying problem areas of our solution.
While the current solution generates very interesting caves, according to users, the generation is not fast and well structured enough to be seamless, and many users disliked encountering holes in the world where tiles had not been generated yet. This, of course, breaks immersion, and the technical limitations presented in section III-C.4 on page 42, are to blame. However, it was also pointed out in that section that even though we were required to downgrade to a primitive static grid, the methods developed for cave structuring via L-Systems and cave modelling on the GPU, can be easily converted to a sparse voxel octree structure residing in a 3D texture. Furthermore we have shown that both the generation speed and the geometry rendering speed can be greatly increased by pre-generating the values of some of the noise functions and saving them in 3D textures.
Even without these improvements, though, our tool is still useful in a context where the caves can be generated before the player is able to enter them.

Another complaint was about the somewhat odd and unnatural shapes of the features inside the caves, namely the stalactites. While it is true that the current iteration of these features sometimes look out of place, this is a problem that can be solved. The reason for their odd shape is due to the limitations of the voxel grid, the voxel resolution not making it possible to achieve the level of detail that is required to create natural looking thin stalactites. Thin stalactites would be one voxel thick, and wold look like stretched pyramids. It is possible, however, to alleviate this problem, as well as well as to enhance the stratification on the walls, by using tessellation with a normal displacement map similar to our bumpmap. By applying tessellation to our 4-voxel thick, blunt stalactites, they can be easily and procedurally modelled into high quality smooth and thin stalactites.We have also

previously mentioned that pre-modelled assets can be spawned instead of the procedural stalactites.
The reason why tessellation has not been added in this project, is that it proved to be a very technologically complex task on a triplanar shader: the normally automatic tessellation geometry-pass, must be implemented manually to support procedural tangents and fake UVs.

Surprisingly, some people commented on the lack of variety even though as we described in section IV-A.2 on page 52, our caves present a wide spectrum of possible shapes. In section IV-B on page 52, several possible reasons for this complaint were explored. At the same time, some users themselves pointed out two reasons, the first being the lack of very tight passages.
We have demonstrated that our metaball can take short or narrow shapes, but the tunnels generally have a minimum radius. This was a feature to guarantee that the player cahracters never get stuck. But in a game with a character that can crawl for instance, it is definitely possible to distort the metaball radius even further.

The second problem pointed out by users was that they do not want the canyon or the ice textures to be the only thing they see. Therefore we hypothesise that the textures need more lighting transitions, decals, and an overarching progression to and from fundamentally different tones of rock texture. This can be solved with the aid of an artist, and with the additional procedural decoration potential described in the material method section III-E.2 on page 47, as well as in the procedural shading review of section II-E.3 on page 23.

Once these variations would be implemented, as well as decorative props such as rocks, vegetation, and more specific game related structures would be added, a new round of testing would be required to precisely gauge whether the structural flow of the caves is itself varied enough. And given its procedural nature, we see no reason why it would not be.

Apart from these complaints, as well as complaints about the lack of gameplay, most users were excited by our demo. They reportedly really liked the art style and were excited by the prospect of exploring caves like this in an actual gameplay setting, with one user answering the following when asked what type of game our caves would be suited for.

> *Please please PLEASE do some exploration thingy, it works great for that! The feeling that I just need to get to the end of the cave to see whats there is pretty strong. It reminds me of the caves in terraria or starbound, but these are just a thousand times more interesting in themselves!*

This, along with the fact that 66% of users answered positively when asked if they would be interested in these caves as part of a game, clearly shows that procedural caves are something that there is a demand for from players.
It also shows that an interesting environment is very important to the enjoyment of a game. As mentioned in section I on page 1, repeated and uninteresting environments are something that players notice and dislike. On the other hand, interesting and varied environments can actually be a good experience in and of itself. While there were some complaints about the lack of gameplay, some users actually enjoyed simply exploring the environment.

B. Viability and Applications of the Cave Generator

As seen above, our method of generating caves by using an L-system as a scaffold for a distorted metaball, is clearly a viable way to generate procedural caves. While the current solution is not optimal due to problems with generation speed and performance, solutions to these problems exist as shown in section III-C.4 on page 42.
With these problems fixed, there would be several possible applications for our solution, divided into two main categories: As part of a content creation pipeline, or as a way to procedurally generate caves during gameplay.

As part of a standard asset pipeline, our cave generation tool could be leveraged to speed up creation of caves for all types of games in a similar fashion to how middleware, such as SpeedTree, are used to procedurally generate vegetation in many current games. Designers could conceivably use our tool to speed up creation of caves in games where caves represent a prominent part of the world. It could also allow smaller developers with limited resources to incorporate detailed caves as part of their game without devoting many man-hours to manually creating them.
This is also where the interactive evolution presented in section III-D on page 43 would be useful. This, along with the many customization options for both the L-system and the metaball, allows designers to have a good level of control over the customisability of the generated geometry.

It would also be possible to allow designers to simply sketch the overall strucuture of caves, by providing a tool that lets them interact with the rule file in a visual and intuitive manner. The cave could then be generated to follow this designer specified guide lines, to quickly generate detailed caves that fit the required specifications.
Furthermore, the Metaball Approach section III-C.2 on page 36, concluded with discussing the affordance and potential of our tool to be extended with a visual programming frontend, where a designer could plug in, and experiment with, different kinds of procedural noise (and parameters for said noise) while previewing a live set of metaball shapes. This would essentially customise the nodes in the metaball distortion equation and allow for the shapes of the caves to be refined and improved by effectively using an artist as a fitness function.
This mixed initiative approach to cave generation would have great value for different game development studios. It could

speed up the asset creation process, while still maintaining designer control over the structure and feel.

The other possible, and main use for our cave generation tool, would be as a way to create procedural underground landscapes during gameplay. Games, such as the wildly successful Minecraft, already use procedural techniques to generate their entire world, but these techniques are all relatively simple. In the case of Minecraft, the technique is also designed for a world made of cubes. For a high fidelity 3D game, a tool like the one presented in this thesis, would help create a more immersive experience.
Our tool is well suited for this application, as it generates a wide variety of caves. Our solution also presents basic functionality to allow the generated caves to cater to certain gameplay needs. This functionality consists of the ability to limit the verticality of caves by restricting the maximum possible angles that a tunnel can move at vertically, as well as the ability to limit the amount of dead ends by connecting branches together. Together with the ability to edit the L-System rules themselves, it allows a designer to guarantee that every generated cave conforms to certain properties.

In terms of specific game styles our landscapes would be best suited for, users mentioned a first or third person exploration game as a possible match, and considered that the lower gravity, and the possibility to jump very high, fit the environment very naturally. This type of game would also benefit from the emergent nature of the world, to immensely increase replayability.

Another type of game that was mentioned, where a procedurally generated world and caves would enhance gameplay, is the first or third person survival genre. This genre is relatively new, and contains games such as DayZ[80] and The Forest[108]. The problem with these games, however, is that while some things such as the placement of loot and player spawn points are procedural, the world itself is not. This means that players quickly get to know what the world is made of, and the initial sense of wonder and discovery is gone.
With a procedural approach, that initial feeling can be maintained, by placing the player in a new procedurally generated world for every playthrough of the game. It also avoids another problem plaguing this genre, which is the limited size of the game world. With PCG, the game can simply generate more terrain as the player moves towards the edge.

Finally, and in contrast to the slower paced genres mentioned above, our environment was reportedly also a good match for more fast paced games such as Gunvalkyrie[106], where the third person character has a certain amount of air control through the use of brief jetpack bursts, and fall-cancelling attack moves. A third person shooter, hack-n-slash, or beat-em-up game, perhaps in multiplayer, would be a good possible match for our environments, which resemble those of Gunvalkyrie.

1) Comparison to Other Cave Generation Tools

We also note that our results are very practical and usable in real world scenarios, as opposed to the results presented by other academic work. We provide gameplay at (over) 60FPS and 1080p, with high quality meshes and materials.
For example, Cui's paper [22], which we have reviewed in the first section, while it claims to provide real-time PCG caves, the meshes generated are too high-poly, and also have two demanding additional processing steps: smoothing the mesh, and patching holes in the geometry; neither of which are needed in our solution. The results can be compared in figure 124.

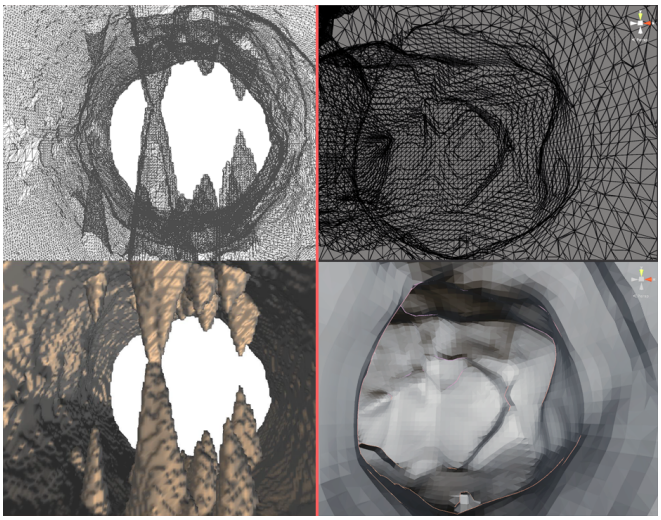


FIG. 124: Wireframe of a sample of Cui's mesh [22] on the left, compared to our more practical results on the right.

At the same time, our landscapes are much more reliable and coherent than the other noise based solutions: figure 125 on the next page.

Similarly, (these) other cave approaches, model walls only through one superficial application of Perlin or Simplex noise, and as such only provide random appearing noisy geometry. Our solution represents an improvement, due to its more advanced distortion function, and the metaball working together with the lsystem to also create larger features and landmarks.

Our solution is at the same time fully automatic, while also maintaining the possibility of user control, as described in the previous sections.

Overall, it is clear that even though procedural cave generation has many benefits and could be used both as part of a content pipeline and as a way to generate caves during gameplay, not much work has been done to explore this area. Our solution shows that it is very much possible to generate 3D caves that are expressive and believable, and

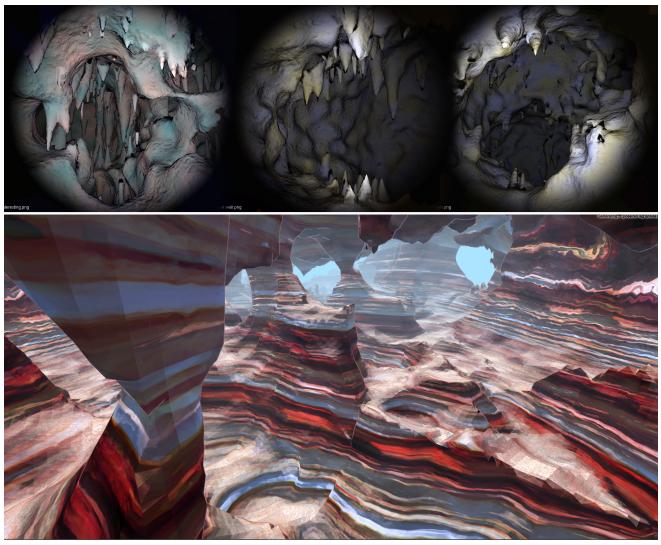


FIG. 125: The three images on the top are from Yurovchak's results [44]. The bottom is a large chamber from our solution.

that such a solution would have a place in the industry.

C. Future Work

Our current solution generates only finite pre-generated caves. However, it is possible to expand this functionality to allow for infinite caves. Instead of expanding the L-system all at once, it could be dynamically expanded in response to player movement. Doing this, it is possible to keep generating new tunnels ahead of the player in whichever direction he is going.
It is possible to go a step further, however, and take the players actions into account in addition to his position. For example, if a player is rushing through the cave, the L-system could start branching more heavily, creating larger areas for the player to explore, in an effort to slow him down.
The possibilities are many, and the specific alterations would depend completely on the game.

In section V-B on page 56, it was shown how our method could be used for various types of applications in the industry. Our current solution, however, only demonstrates the fundamental approach to generating the cave environments. To elevate our project from the status of a simple tech demo to a finished tool, a Sparse Voxel Octree must be used. Migrating our modelling methods to an octree structure stored as a 3D texture, will also easily afford destructible terrain or otherwise minable terrain, as is commonly found in games featuring emergent procedural landscapes.

Lastly, as discussed in section II-E.2 on page 19, in order to obtain higher quality terrain, the only available option is to switch the isosurface extraction algorithm from a Primal grid marching method, to a Dual grid method (such as Dual Marching Cubes). As we have also discussed, a Dual method depends on Hermite data. This, to our method specifically, translates into the need to obtain the gradient function of our metaball warping equation (section III-C.2 on page 36). This becomes a problem of performance, as our metaball does not conform to the shape of a primitive, and therefore cannot be used as a CSG operation [5] (Constructive Solid Geometry[11], used in Everquest Next[87]), or reproduced as a set of CSG operations (CSG tree) since none of its parts are primitives. The remaining option therefore is to compute the gradient of the metaball distortion equation directly, to obtain the normal. This however is slow, because it requires to identify and compute the density function (or, metaball warping function) on additional small deltas of the current point, along the voxel surface (to get a tangent and thus a normal).
Future research however might prove it viable to pre-compute a certain range of metaball behaviour (like the range shown in the grid figure 93 on page 39), and store it in a large volume of data such as a 3D texture, along with pre-computed gradients of the corresponding metaball function points.
Alternatively, drawing inspiration from the concept of a CSG Tree, each noise component of the metaball equation could potentially be pre-computed and saved to a 3D texture volume separately, along with each point's pre-computed normal. At runtime the equation could theoretically sample each component from the 3D lookups (instead of calculating them) and combine the corresponding normals into the final normal required by the Hermite data.
Pre-computed 3D textures would concievably diminish the range of the metaball variation, but as discussed in section III-C.4 on page 42 given a certain minimum 3D texture size, and lookup permutations, this is a non-issue.

D. Concluding Remarks

With this project, we set out to both research and review the possibilities of procedurally modelling caves and the technologies involved, as well as to develop an original and better approach of our own. Throughout this document, we have shown that by combining simple methods for generating both structure and detail, it is possible to generate expressive and believable caves. We have also shown that it is possible to obtain a great degree of control over this generation through various means.
Our solution fits into an empty niche in the current research on procedurally generated games, as there has been very little work done on the the topic of procedurally modelled 3D caves in both academia and the games industry. Only a few papers have been written on the topic and very few games use procedurally generated caves, most of them being based on simple methods, such as the modified Perlin worms used in Minecraft (see section III-B.2 on page 28.

While more work is required to ready our implementation for commercial production, we are able to demonstrate a solid approach to generating believable 3D underground caverns.

[11]The technique of subtracting a primitive volume, with known or easily computed normals, from a surface

We provide a much needed contribution to this otherwise sparse but important niche of research within the field of real-time PCG.

VI. Academic References

[1] P. Bourke, "Marching cubes," 1994. url: http://paulbourke.net/geometry/polygonise/

[2] W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3d surface construction algorithm," 1987. url: http://dl.acm.org/citation.cfm?id=37422

[3] Lawrence Johnson and Georgios Yannakakis and Julian Togelius, "Cellular Automata for Real-time Generation of Infinite Cave Levels," 2010.

[4] Jeremy Kun, PhD, University of Illinois, "The Cellular Automaton Method for Cave Generation," 2012. url: http://jeremykun.com/2012/07/29/the-cellular-automaton-method-for-cave-generation/

[5] Philip Trettner, RWTH Aachen Computer Science, Project Leader of Upvoid Studios, "Terrain Engine Part 2 - Volume Generation and the CSG Tree," 2013. url: https://upvoid.com/devblog/2013/07/terrain-engine-part-2-volume-generation-and-the-csg-tree/

[6] Ruben Smelik and Tim Tutenel and Rafael Bidarra and Bedrich Benes, "A Survey on Procedural Modeling for VirtualWorlds," 2009.

[7] Ruben Smelik and Tim Tutenel and Klaas Jan de Kraker and Rafael Bidarra, "Interactive Creation of Virtual Worlds Using Procedural Sketching," in *Proceedings of Eurographics*, 2010.

[8] R. Smelik, T. Tutenel, K. J. de Kraker, and R. Bidarra, "Integrating procedural generation and manual editing of virtual worlds," in *PC Games*, 2010.

[9] Jonathon Doran and Ian Parberry, "Controlled Procedural Terrain Generation Using Software Agents," *IEEE Transactions on Computational Intelligence and AI in games*, vol. 2, no. 2, 2010.

[10] K. Perlin, "An Image Synthesizer," in *SIGGRAPH Comput. Graph.*, 1985. url: http://www.cs.jhu.edu/~misha/Fall13/Readings/Perlin85.pdf

[11] Ken Perlin, "Improving noise." Media Research Laboratory, Dept. of Computer Science, New York University, 2002. url: http://mrl.nyu.edu/~perlin/paper445.pdf

[12] Alain Fournier and Don Fussell and Loren Carpenter, "Computer Rendering of Stochastic Models," *Communications of the ACM*, vol. 25, no. 6, 1983.

[13] F. Kenton Musgrave and Crain E. Kolb and Robert S. Mace, "The Synthesis and Rendering of Eroded Fractal Terrains," *Computer Graphics*, vol. 23, no. 3, 1989. url: http://cs.allegheny.edu/sites/cs230F2006/uploads/p41-musgrave.pdf

[14] G. Greeff, "Interactive voxel terrain design using procedural techniques," Master's thesis, Stellenbosch University, 2009. url: http://scholar.sun.ac.za/handle/10019.1/2631

[15] M. Cepero, "Procworld, voxel terrain engine," 2012. url: http://procworld.blogspot.dk/2012/11/a-system-of-caves.html

[16] H. Samet, *The Design and Analysis of Spatial Data Structures*. Addison - Wesley Publishing Company, 1990.

[17] T. Alexander *et al.*, *GPU Gems 3*. NVIDIA Corporation, 2007. url: http://http.developer.nvidia.com/GPUGems3/gpugems3_ch01.html

[18] B. Benes and R. Forsbach, "Layered Data Representation for Visual Simulation of Terrain Erosion," in *SCCG 01: Proceedings of the 17th Spring Conference on Computer Graphics*. IEEE Computer Society, 2001, pp. 80–86.

[19] M. Gamito and F. K. Musgrave, "Procedural landscapes with overhangs," in *10th Portugeuse Computer Graphics Meeting*, 2001, pp. 33–42.

[20] J. VanDerLei, "Challenges in procedural terrain generation." url: https://www.inter-actief.utwente.nl/studiereis/pixel/files/indepth/JoeriVanDerLei.pdf

[21] R. Geiss *et al.*, "GPU Gems part 3, Chapter 1: Generating Complex Procedural Terrains Using the GPU," 2007.

[22] J. Cui, "Procedural cave generation," Thesis for Msc. in Computer Science, University of Wollongong, Sidney Australia, 2011. url: http://ro.uow.edu.au/cgi/viewcontent.cgi?article=4495&context=theses

[23] A. Peytavie, E. Galin, J. Grosjean, S. Merillou, "Arches: a Framework for Modeling Complex Terrains," in *EUROGRAPHICS 2009 / P. Dutr and M. Stamminger*, 2009. url: http://liris.cnrs.fr/~egalin/Pdf/2009-arches.pdf

[24] Matt Boggus and Roger Crawfis, "Procedural Creation of 3D Solution Cave Models." The Ohio State University, 2009. url: ftp://ftp.cse.ohio-state.edu/pub/tech-report/2009/TR19.pdf

[25] A. N. Palmer, "Cave Geology." Dayton, OH: Cave Books, 2007.

[26] Ivo Mark, "Random Midpoint Displacement." url: http://www.cescg.org/CESCG97/marak/node3.html

[27] A. Fournier, D. Fussel, and L. Carpenter, "Computer Rendering of Stochastic Models." Communications of the ACM, 25:371-384, 1982. url: https://blog.itu.dk/mpgg-e2010/files/2010/09/1011853781.pdf

[28] J. Togelius, N. Shaker, and J. Dormans, "Grammars and l-systems with applications to vegetation and levels," in *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, N. Shaker, J. Togelius, and M. J. Nelson, Eds. Springer, 2014.

[29] A. Lindenmayer, "Mathematical models for cellular interactions in development 1. filaments with one-sided inputs," *Journal of theoretical biology*, vol. 18, no. 3, pp. 280–299, 1968.

[30] J. Knutzen, "Generating climbing plants using l-systems," Master's thesis, Chalmers University of Technology and University of Gothenburg, 2009. url: http://www.cse.chalmers.se/~uffe/xjobb/climbingplants.pdf

[31] P. Prusinkiewicz, M. Hammel, J. Hana, and R. Mech, "L-systems: From the theory to visual models of plants," in *Proceedings of the 2nd CSIRO Symposium on Computational Challenges in Life Sciences*, M. T. Michalewicz, Ed. CSIRO Publishing, 1996. url: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.22.6081&rep=rep1&type=pdf

[32] Y. I. H. Parish and P. Müller, "Procedural modeling of cities," in *Proceedings of ACM SIGGRAPH 2001*, E. Fiume, Ed. ACM Press, 2001, pp. 301–308. url: http://hal.archives-ouvertes.fr/docs/00/52/75/00/PDF/instant_architecture.pdf

[33] G. Danks, S. Stepney, and L. Caves, "Protein folding with stochastic l-systems," in *ALife XI*. MIT Press, 2008, pp. 150–157. url: http://alifexi.alife.org/papers/ALIFExi_pp150-157.pdf

[34] D. A. Ashlock, S. P. Gent, and K. M. Bryden, "Evolution of l-systems for compact virtual landscape generation," in *Proceedings of IEEE Congress on Evolutionary Computing*, 2005, pp. 2760–2767. url: http://eldar.mathstat.uoguelph.ca/dashlock/eprints/Lsystem2.pdf

[35] D. Ebert, K. Musgrave, D. Peachey, K. Perlin, and S. Worley, "Texturing and Modeling: A Procedural Approach, Third Edition," 2003, pp. chapter 2, p68.

[36] Stefan Gustavson, Linkping University, "Simplex noise demystified," 2005. url: http://webstaff.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf

[37] A. Lagae, S. Lefebvre, R. Cook, T. DeRose, G. Drettakis, D. Ebert, J. Lewis, K. Perlin, and M. Zwicker, "State of the art in procedural noise functions." url: http://graphics.cs.kuleuven.be/publications/LLCDDELPZ10STARPNF/LLCDDELPZ10STARPNF.pdf

[38] A. Lagae, D.S. Ebert, K. Perlin et al., "Survey of procedural noise functions." url: http://people.cs.kuleuven.be/~ares.lagae/publications/LLCDDELPZ10SPNF/LLCDDELPZ10SPNF.pdf

[39] R. Wolfe, "Teaching Texture Mapping Visually, DePaul University, Siggraph." url: http://www.siggraph.org/education/materials/HyperGraph/mapping/r_wolfe/r_wolfe_mapping_6.htm

[40] R. Bridson, J. Hourihan, and M. Nordenstam, "Curl-noise for procedural fluid flow." url: http://www.cs.ubc.ca/~rbridson/docs/bridson-siggraph2007-curlnoise.pdf

[41] A. Moorer, "Terrain generation basis functions." url: http://andy.moonbase.net/archives/410

[42] J. Kopf, C.-W. Fu, D. Cohen-Or, O. Deussen, D. Lischinski, and T.-T. Wong, "Solid texture synthesis from 2d exemplars." url: http://www.cs.tau.ac.il/~dcor/articles/2007/Solid-Texture.pdf

[43] David Forsyth, University of Illinois, "Procedural shading and texturing." url: http://luthuli.cs.uiuc.edu/~daf/courses/ComputerGraphics/Week8/Shading.pdf

[44] Andrew Yurovchak, Washington University, "Adventures in digital spelunking," 2009. url: http://courses.cs.washington.edu/courses/cse557/09au/projects/final/artifacts/yuro/index.html

[45] A. Jnsson, "Fast metaballs," 2001. url: http://www.angelcode.com/dev/metaballs/metaballs.html

[46] N. Shaker, A. Liapis, J. Togelius, R. Lopes, and R. Bidarra, "Constructive generation methods for dungeons and levels," in *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, N. Shaker, J. Togelius, and M. J. Nelson, Eds. Springer, 2014.

[47] H. Samet, "The design and analysis of spatial data structures," 1990. url: http://cdn.preterhuman.net/texts/math/Data_Structure_And_Algorithms/The%20Design%20And%20Analysis%20Of%20Spatial%20Data%20Structures%20-%20Hanan%20Samet.pdf

[48] Addison-Wesley, "Chapter 37. Octree Textures on the GPU," 2005.

url: http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter37.html
[49] J. Revelles, C. Urena, and M. Lastra, "An Efficient Parametric Algorithm for Octree Traversal." url: http://wscg.zcu.cz/wscg2000/papers_2000/x31.pdf
[50] S. Laine and T. Karras, "Efficient Sparse Voxel Octrees," 2010. url: http://www.nvidia.com/object/nvidia_research_pub_017.html
[51] K. Miller, Stanford, "KD-Tree," 2010. url: http://web.stanford.edu/class/cs106l/handouts/assignment-3-kdtree.pdf
[52] Mikola Lysenko, Wisconsin University, "Smooth Voxel Terrain." url: http://0fps.net/2012/07/12/smooth-voxel-terrain-part-2/
[53] Tristam, "Isosurface Extraction." url: http://swiftcoder.wordpress.com/planets/isosurface-extraction/
[54] M. Levoy, "Efficient Ray Tracing of Volume Data," 1990. url: http://webuser.uni-weimar.de/~weiszig/post/svo_diplom/raytracing/p245-levoy.pdf
[55] J. Carmack, "Ars Technica, Raytracing benefits and performance." url: http://arstechnica.com/gadgets/2013/01/shedding-some-realistic-light-on-imaginations-real-time-ray-tracing-card/?comments=1&post=23723213#comment-23723213
[56] John Carmack, via Sam Lapere, "Eventually ray tracing will win," 2011. url: http://raytracey.blogspot.dk/2011/08/john-carmack-eventually-ray-tracing.html
[57] John Carmack, via Lapere, "Physics of Light and Rendering," 2013. url: http://raytracey.blogspot.dk/2013/08/carmacks-physics-of-light-and-rendering.html
[58] S. Roth, "Ray Casting for Modeling Solids," 1982.
[59] A. Doi and A. Koide, "An Efficient Method of Triangulating Equi-Valued Surfaces by Using Tetrahedral Cells," 1991. url: http://search.ieice.org/bin/summary.php?id=e74-d_1_214
[60] C.-K. Shene, "Mesh basics." url: http://www.cs.mtu.edu/~shene/COURSES/cs3621/SLIDES/Mesh.pdf
[61] G. M. Nielson and B. Hamann, "The Asymptotic Decider: Resolving the Ambiguity in Marching Cubes." url: http://graphics.stanford.edu/courses/cs164-10-spring/Handouts/paper_p83-nielson.pdf
[62] Sarah F. Frisken Gibson, "Constrained Elastic SurfaceNets: Generating Smooth Models from Binary Segmented Data," 1999. url: http://www.merl.com/papers/docs/TR99-24.pdf
[63] V. Holmstrom, "Modified surfacenets: Smoothing a marching cubes mesh." url: http://www.cs.wm.edu/~nikos/cs420/projects/ModifiedSurfaceNets.pdf
[64] S. Schaefer and J. Warren, "Dual Marching Cubes: Primal Contouring of Dual Grids," 2004. url: http://www.cs.rice.edu/~jwarren/papers/dmc.pdf
[65] T. Ju, F. Losasso, S. Schaefer, and J. Warren, "Dual contouring of hermite data," in *Proceedings of ACM SIGGRAPH 2002*. Siggraph, Rice University, 2002. url: http://www.frankpetterson.com/publications/dualcontour/dualcontour.pdf
[66] L. P. Kobbelt, M. Botsch, U. Schwanecke, and H.-P. Seidel, "Feature sensitive surface extraction from volume data," in *Proceedings of ACM SIGGRAPH 2001*, 2001, pp. 57–66. url: http://mesh.brown.edu/DGP/pdfs/Kobbelt-sg2001.pdf
[67] Upvoid Studios, Germany, "Terrain engine part 1 - dual contouring," 2013. url: https://upvoid.com/devblog/2013/05/terrain-engine-part-1-dual-contouring/
[68] J. Gregson, "Dual contouring." url: http://jamesgregson.blogspot.dk/2011/04/dual-contouring.html
[69] M. Cepero, "From voxels to polygons," 2010. url: http://procworld.blogspot.dk/2010/11/from-voxels-to-polygons.html
[70] J. Togelius, E. Kastbjerg, D. Schedl, and G. N. Yannakakis, "What is procedural content generation? mario on the borderline," in *Proceedings of the 2nd Workshop on Procedural Content Generation in Games*, 2011.
[71] J. Togelius, N. Shaker, and M. J. Nelson, "Introduction," in *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, N. Shaker, J. Togelius, and M. J. Nelson, Eds. Springer, 2014.
[72] Jonathan Shewchuk, Berkeley, "Dual marching cubes." url: http://www.cs.berkeley.edu/~jrs/mesh/present/Andrews.pdf
[73] O. St'ava, B. Benes, R. Mech, D. G. Aliaga, and P. Kristof, "Inverse procedural modeling by automatic generation of l-systems," in *Computer Graphics Forum*, vol. 29, no. 2, 2010. url: https://www.cs.purdue.edu/cgvlab/papers/aliaga/eg2010.pdf
[74] "Whitepaper: NVIDIAs Next Generation CUDA Compute Architecture," NVIDIA. url: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf
[75] W. Engel, "Compute Shader Optimizations for AMD GPUs: Parallel Reduction." url: http://diaryofagraphicsprogrammer.blogspot.dk/2014/03/compute-shader-optimizations-for-amd.html

VII. Media References

[76] Cellar Door Games, "Rogue Legacy," 2014. url: http://roguelegacy.com/
[77] "Dragon age 2," BioWare, 2011.
[78] "Civilization series," Firaxis Games and others, 1991-2014.
[79] "Dragon age 2 review," Elder Geek, 2011. url: http://elder-geek.com/2011/03/dragon-age-2-review/
[80] "Dayz," Bohemia Interactive, TBD.
[81] "Minecraft," Mojang, 2011. url: https://minecraft.net/
[82] "No man's sky," Hello Games, TBD.
[83] "Descent," Parallax Software, 1994.
[84] "Permutation racer," Tom Betts, Nullpointer, 2014. url: http://www.nullpointer.co.uk/content/permutation-racer/
[85] "Interview of Hello Games' "No Man's Sky"," Polygon, 2014. url: http://www.polygon.com/a/e3-2014/no-mans-sky
[86] "Krautscape," Mario von Rickenbach , Playables LLC, 2014. url: http://www.krautscape.net/
[87] "Everquest next landmark," Sony Omline Entertainment, 2014. url: https://www.landmarkthegame.com/media/youtube
[88] "Everquest next," Sony Omline Entertainment, 2014. url: https://www.everquestnext.com/media?category=video
[89] "Phreatic passage in Mammoth Cave National Park." url: http://www.gogobot.com/mammoth-cave-national-park-ky_2
[90] "Great Orme Cave (Elephant's Cave)." url: http://www.ucet.org.uk/index.php/forum/21-knowledge-bank/812-great-orme-cave-elephant-s-cave
[91] "Isosurface, definition." url: https://en.wikipedia.org/wiki/Isosurface
[92] "Wikimedia, heightmap images." url: https://en.wikipedia.org/wiki/Heightmap
[93] S. Strandgaard, "Worley noise." url: https://secure.flickr.com/photos/12739382@N04/2564255134/in/photostream/
[94] "General-purpose computing on Graphics Processing Units." url: http://gpgpu.org/about
[95] "Unreal engine 3," Epic Games, 2004. url: https://www.unrealengine.com/products/unreal-engine-3
[96] Unity, "DirectX 11 Compute Shaders." url: http://docs.unity3d.com/Manual/ComputeShaders.html
[97] "Libnoise, example: Perlin worms," 2014. url: http://libnoise.sourceforge.net/examples/worms/index.html
[98] M. Smith, "Pixar's Lightspeed." url: http://thisanimatedlife.blogspot.ch/2013/05/pixars-chris-horne-sheds-new-light-on.html
[99] T. Betts, "Nullpointer," 2014. url: http://www.nullpointer.co.uk
[100] M. Hansmeyer, 2014. url: http://www.michael-hansmeyer.com
[101] hatmihp, "Terrain (heightmap) generation." url: http://hatmihp.wordpress.com/tag/heightmap/
[102] T. Merino. url: http://undergroundflashgun.esconatura.com/author.php
[103] D. stock photos. url: http://www.dreamstime.com/
[104] S. Thrainsson. url: http://www.skarpi.is/
[105] J. Groene. url: http://kevintbear.blogspot.dk/
[106] "Gunvalkyrie," (Sega, Xbox), 2002. url: https://en.wikipedia.org/wiki/Gunvalkyrie
[107] "Waking mars," Tiger Style, 2012. url: https://en.wikipedia.org/wiki/Waking_Mars
[108] "The forest," Endnight Games, 2014.
[109] N. Geographic, "Mammoth Cave National Park." url: http://travel.nationalgeographic.com/travel/national-parks/mammoth-cave-national-park/
[110] "The elder scrolls V: Skyrim," Bethesda Game Studios, 2011. url: http://www.elderscrolls.com/skyrim
[111] "Directcompute api," Microsoft. url: http://msdn.microsoft.com/en-us/library/windows/desktop/ff476331(v=vs.85).aspx
[112] "Cuda toolkit," Nvidia. url: https://developer.nvidia.com/cuda-toolkit
[113] Nvidia, "Tessellation." url: http://www.nvidia.com/object/tessellation.html

[114] “Opencl programming standard,” AMD. url: http://developer.amd.com/tools-and-sdks/opencl-zone/
[115] AMD, “Mantle API,” 2013. url: http://www.amd.com/en-us/innovations/software-technologies/mantle
[116] “Unity3d game engine,” Unity3D. url: http://unity3d.com/

VIII. Appendix

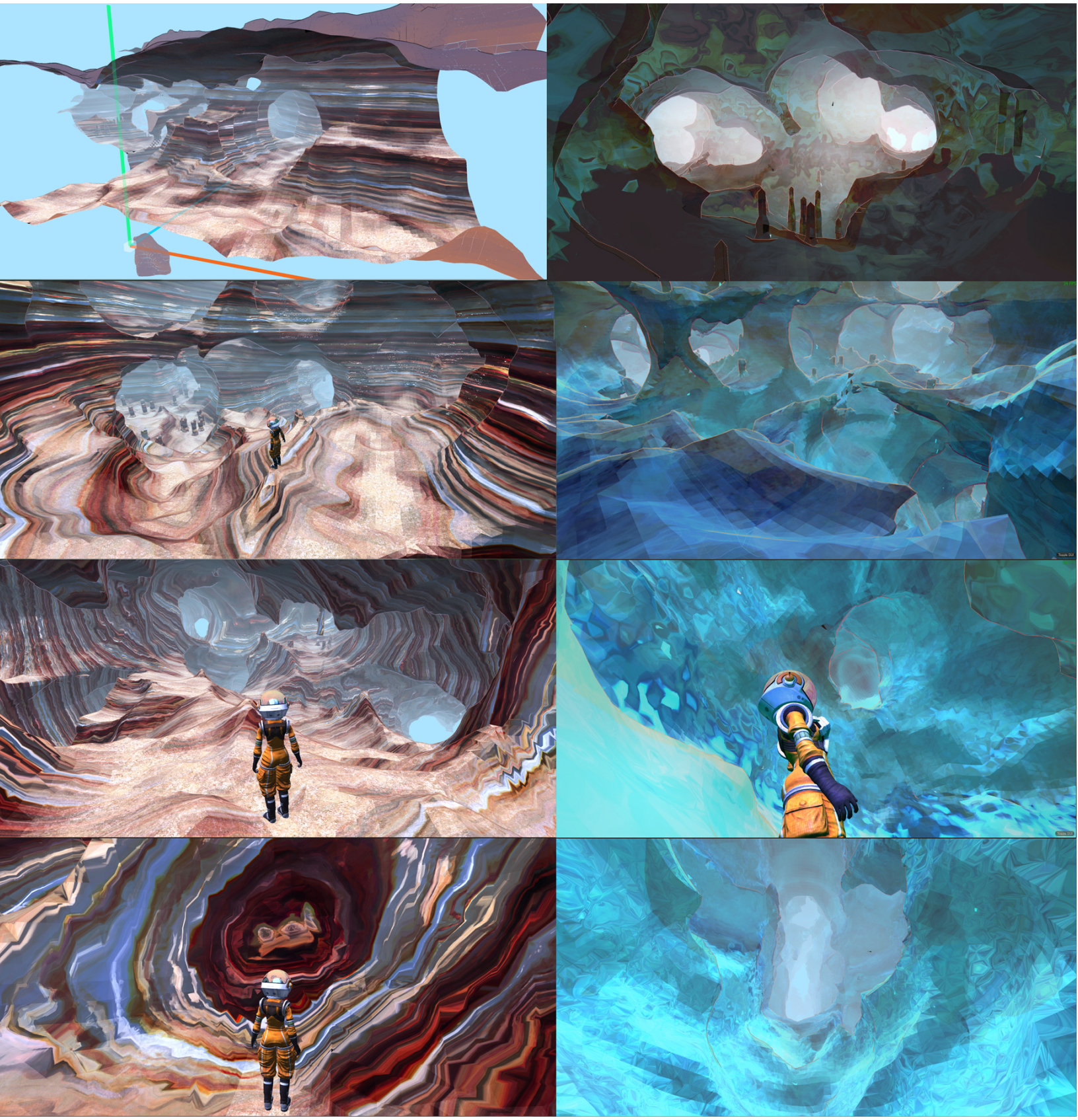


FIG. 126: Annex 1

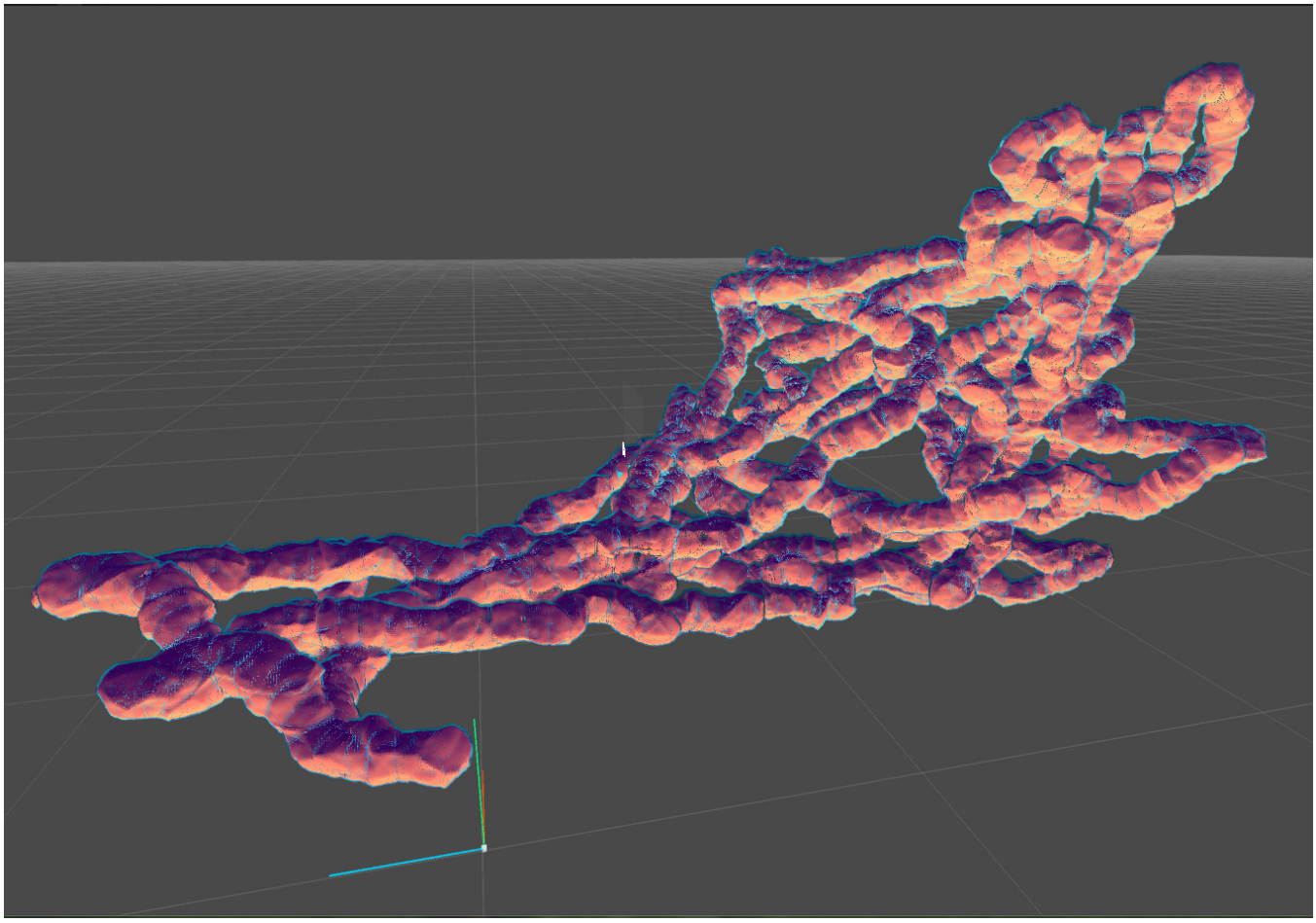


FIG. 127: Annex 2

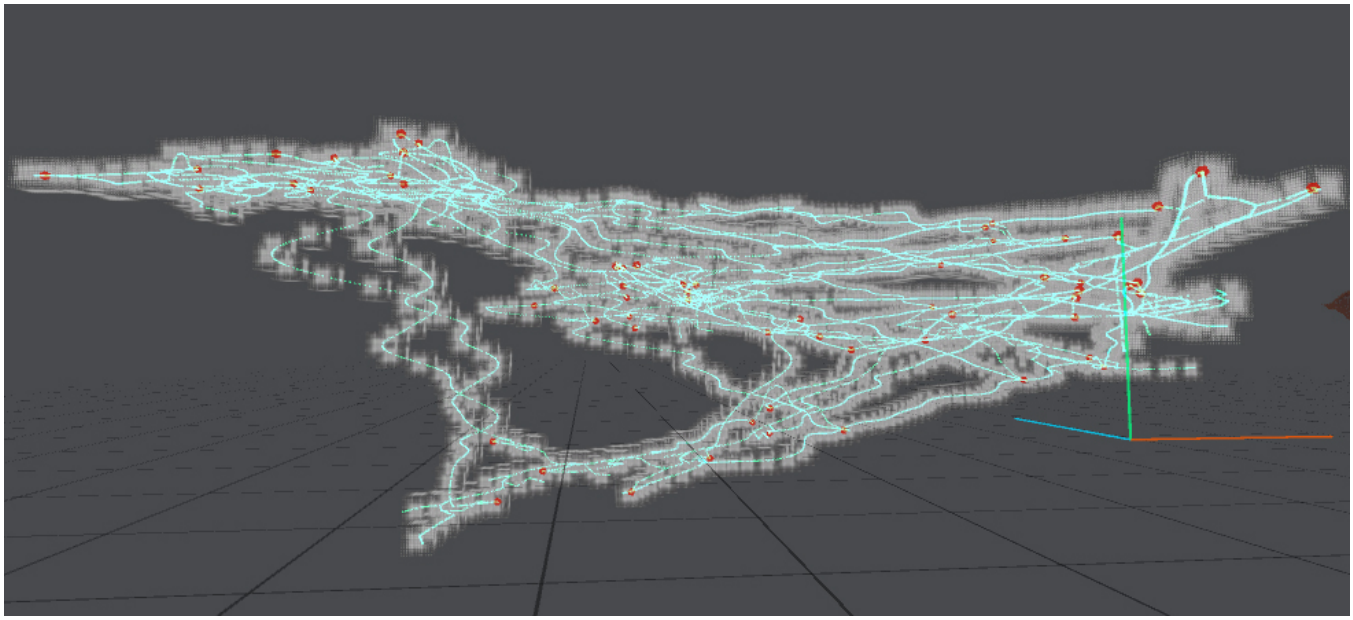


FIG. 128: Annex 3