2011

Procedural cave generation

Juncheng Cui

University of Wollongong

Recommended Citation

UNIVERSITY OF WOLLONGONG

COPYRIGHT WARNING

You may print or download ONE copy of this document for the purpose of your own research or study. The University does not authorise you to copy, communicate or otherwise make available electronically to any other person any copyright material contained on this site. You are reminded of the following:

Copyright owners are entitled to take legal action against persons who infringe their copyright. A reproduction of material that is protected by copyright may be a copyright infringement. A court may impose penalties and award damages in relation to offences and infringements relating to copyright material. Higher penalties may apply, and higher damages may be awarded, for offences and infringements involving the conversion of material into digital or electronic form.
Procedural Cave Generation

Juncheng CUI

"This thesis is presented as part of the requirements for the award of Master of Computer Science - Research from the University of Wollongong"

August 2011
Dedicated to

Shiqing Cui and Shujuan Shi
DECLARATION

I, Juncheng Cui, declare that this thesis, submitted in partial fulfilment of the requirements for the award of Master of Computer Science – Research, in the School of Computer Science and Software Engineering, University of Wollongong, is wholly my own work unless otherwise referenced or acknowledged. The document has not been submitted for qualifications at any other academic institution.

__________________

Juncheng Cui

Date:
ABSTRACT

Procedural content generation is becoming an increasingly popular research area as a means of algorithmically generating scene content for virtual environments. The automated generation of such content avoids the manual labour typically associated with creating scene content, and is extremely useful in application areas such as computer graphics, virtual reality, movie production and video games. While virtual 3D caves are commonly featured in these virtual environment applications, procedural cave generation is not an area that has received much attention among the research community to date. The research in this thesis investigated and developed an approach to automating the process of generating visually believable 3D cave models.

To generate 3D cave structures with diverse characteristics, the use of different noise functions with different parameters was examined. This thesis presents experimental results showing the relationship between these factors and their influences on the resulting cave structure. Furthermore, the construction of an efficient spatial data structure had to be developed as part of the cave generation process. This research proposed a unique bottom-up voxel-based octree building approach that was specifically designed for the purposes of constructing the cave structure.

In order to be able to render the resulting 3D cave model in real-time, a polygonal mesh of the interior cave walls was obtained from the data stored in the octree. To increase the realism of the cave model, a polygonal mesh smoothing technique was used. However, the naïve smoothing technique caused cracks to appear at certain sections of the cave walls. An effective solution to patch the cracks was subsequently developed to produce a smooth crack-free mesh.

In addition, the procedural simulation of two common cave features, namely stalactites and stalagmites, was explored. The generation of these cave characteristics was seamlessly added to the procedural cave creation process. The end result is a procedural approach capable of generating visually believable cave models automatically on a computer, and being able to render the resulting model in real-time.
ACKNOWLEDGEMENTS

I would like to thank my supervisor Dr. Yang-Wai Chow. He is a master in the area of computer graphics and game development, and has ushered me into the topic of procedural content generation. I could not have completed my research and this thesis without his great guidance and encouragement. I also want to express my gratitude to Associate Professor Minjie Zhang. She has showed me the essence of what research is about, and has taught me the methodology of research and the general approach to it. She has also helped me with my English and the formatting of my thesis.

On a personal note, I would like to thank my parents Shiqing Cui and Shujuan Shi for their all-around support, encouragement and constant trust. Learning abroad has been a precious experience for me, and I would like to thank them again for their understanding and love.

Finally, I would like to express my appreciation to Xing Su, Dayong Ye and Zhe Sun, who gave me good suggestions and helped me with writing this thesis.
# TABLE OF CONTENTS

DECLARATION .............................................................................................................. ii  
ABSTRACT .................................................................................................................... iv  
ACKNOWLEDGEMENTS ............................................................................................... v  
TABLE OF CONTENTS ............................................................................................... vi  
LIST OF FIGURES ....................................................................................................... vii  
LIST OF TABLES ......................................................................................................... ix  
LIST OF ABBREVIATIONS ......................................................................................... xii  
PUBLICATIONS .......................................................................................................... xiii

Chapter 1: Introduction ................................................................................................. 1  
  1.1 Major Issues ........................................................................................................... 3  
  1.2 Objectives ............................................................................................................ 3  
  1.3 Contributions ....................................................................................................... 4  
  1.4 Publications ......................................................................................................... 4  
  1.5 Structure of the Thesis ......................................................................................... 5  

Chapter 2: Literature review ......................................................................................... 6  
  2.1 Procedural Content Generation ........................................................................... 6  
  2.2 Procedural Terrain Generation ............................................................................ 8  
    2.2.1 Height map algorithm .................................................................................. 9  
    2.2.2 Layer data representation .......................................................................... 10  
    2.2.3 Voxels .......................................................................................................... 10  
  2.3 Challenges of Procedural Terrain Generation .................................................... 11  
  2.4 Spatial Object Modelling .................................................................................... 12  
    2.4.1 Octree structure .......................................................................................... 12  
    2.4.2 Point-region octree .................................................................................... 14  
    2.4.3 Kd-tree ....................................................................................................... 15  
  2.5 Spatial Object Rendering ................................................................................... 16  
    2.5.1 Solid modelling .......................................................................................... 16  
    2.5.2 Surface modelling ...................................................................................... 17  
  2.6 Flood Fill Algorithm ........................................................................................... 21  
  2.7 Procedural Noise ................................................................................................. 21  
    2.7.1 Lattice noises .............................................................................................. 24  
    2.7.2 Explicit noises .............................................................................................. 27
REFERENCES ................................................................................................................................. 97
APPENDIX A: SUPPORTING PAPERS ......................................................................................... 104
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Procedural terrain generation based on a height map</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>Representation of a quadtree</td>
<td>13</td>
</tr>
<tr>
<td>2.3</td>
<td>Morton order for a quadtree</td>
<td>13</td>
</tr>
<tr>
<td>2.4</td>
<td>Representation of a point-region quadtree</td>
<td>15</td>
</tr>
<tr>
<td>2.5</td>
<td>Representation of a kd-quadtree</td>
<td>16</td>
</tr>
<tr>
<td>2.6</td>
<td>White noise (1D and 2D case)</td>
<td>23</td>
</tr>
<tr>
<td>2.7</td>
<td>Perlin noise calculation in 3D space</td>
<td>25</td>
</tr>
<tr>
<td>2.8</td>
<td>Pseudo code of a bias function</td>
<td>29</td>
</tr>
<tr>
<td>2.9</td>
<td>Pseudo code of a gain function</td>
<td>29</td>
</tr>
<tr>
<td>3.1</td>
<td>Process for constructing the material distribution function</td>
<td>34</td>
</tr>
<tr>
<td>3.2</td>
<td>Pseudo code for the DISTRIBUTION_FUNCTION</td>
<td>34</td>
</tr>
<tr>
<td>3.3</td>
<td>Relative positions of children node for of an octree in 3D space</td>
<td>35</td>
</tr>
<tr>
<td>3.4</td>
<td>Initial state of the quadtree</td>
<td>37</td>
</tr>
<tr>
<td>3.5</td>
<td>Process of getting the quadtree index</td>
<td>38</td>
</tr>
<tr>
<td>3.6</td>
<td>Pseudo code for the GETTREEINDEX function</td>
<td>39</td>
</tr>
<tr>
<td>3.7</td>
<td>Pseudo code for the CHECKEXISTS function</td>
<td>40</td>
</tr>
<tr>
<td>3.8</td>
<td>Pseudo code for the CHECKINSERT_OR_NOT function</td>
<td>40</td>
</tr>
<tr>
<td>3.9</td>
<td>Pseudo code for the INSERTNODE function</td>
<td>42</td>
</tr>
<tr>
<td>3.10</td>
<td>Pseudo code for the MERGE function</td>
<td>42</td>
</tr>
<tr>
<td>3.11</td>
<td>Pseudo code for the TRIM function</td>
<td>42</td>
</tr>
<tr>
<td>3.12</td>
<td>Key steps in the process of building the quadtree</td>
<td>43</td>
</tr>
<tr>
<td>3.13</td>
<td>Pseudo code for the CONTAINS and FIND_CONTAINS_NODE functions</td>
<td>45</td>
</tr>
<tr>
<td>3.14</td>
<td>Pseudo code for the FIND_CONTAINS_NODEWITHDEPTH function</td>
<td>46</td>
</tr>
<tr>
<td>3.15</td>
<td>Facing directions of the nodes</td>
<td>47</td>
</tr>
<tr>
<td>3.16</td>
<td>Steps of finding neighbour nodes on a quadtree</td>
<td>48</td>
</tr>
<tr>
<td>3.17</td>
<td>Flood fill process and determining the boundary surfaces on quadtree</td>
<td>50</td>
</tr>
<tr>
<td>3.18</td>
<td>Pseudo code for the FLOOD_FILL algorithm</td>
<td>51</td>
</tr>
<tr>
<td>3.19</td>
<td>2D depiction of the boundary surfaces</td>
<td>53</td>
</tr>
<tr>
<td>3.20</td>
<td>Model complexity comparison of octree structure</td>
<td>54</td>
</tr>
</tbody>
</table>
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Structure of octree node</td>
<td>36</td>
</tr>
<tr>
<td>3.2</td>
<td>Child indices for the six facing directions</td>
<td>47</td>
</tr>
<tr>
<td>3.3</td>
<td>Boundary properties of octree node</td>
<td>49</td>
</tr>
<tr>
<td>3.4</td>
<td>Enumeration values for the boundary properties</td>
<td>49</td>
</tr>
<tr>
<td>3.5</td>
<td>Face triangles indices of the octree nodes</td>
<td>52</td>
</tr>
<tr>
<td>3.6</td>
<td>Model complexity comparison of octree structure</td>
<td>54</td>
</tr>
<tr>
<td>3.7</td>
<td>Model complexity comparison of triangle mesh</td>
<td>55</td>
</tr>
<tr>
<td>3.8</td>
<td>Comparison of sampling times in space</td>
<td>57</td>
</tr>
<tr>
<td>3.9</td>
<td>Comparison of sampling times in octree structure</td>
<td>58</td>
</tr>
<tr>
<td>4.1</td>
<td>Before and after smooth model complexity comparison</td>
<td>78</td>
</tr>
</tbody>
</table>
### LIST OF ABBREVIATIONS

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPS</td>
<td>First Person Shooter</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>HLSL</td>
<td>High-Level Shading Language</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>LOD</td>
<td>Level Of Detail</td>
</tr>
<tr>
<td>PCG</td>
<td>Procedural Content Generation</td>
</tr>
<tr>
<td>RTS</td>
<td>Real-Time Strategy</td>
</tr>
</tbody>
</table>
PUBLICATIONS


CHAPTER 1: INTRODUCTION

Procedural Content Generation (PCG) techniques have been widely used to generate scene content for virtual environments. This brings about many benefits, such as avoiding manual labour in creating virtual content. Applications of procedural content generation can be found in video games and movie production.

In general, there are four major challenges in the current game development industry [Parberry and Roden 2004]. First of all, as the development of graphics technology and the demands of the players increase, huge but detailed scenes need to be created, such as mountain terrains, large cities, and star fields. Some of these scenes also need to be simulated and rendered interactively in real-time, as these are special requirements for video games and simulation systems. The entire scene content needs to be done by artists, which creates an exponentially large burden on the development cycle.

Secondly, contents created by hand are hard to modify once completed. In the normal game development environment, game content is usually created using tools, such as editors and game engines, which need to be kept consistent by the programmers. However, changes in the game development framework would result in incompatibility between the already produced game content and the new framework.

Thirdly, many tools which are used to develop scene content have completely different formats from the game engine’s requirements. This causes programmers and artists to have different views about the final result, and this difference forces the development cycle to include more create-convert-test cycles. Finally, according to game designer Will Wright [Gamasutra 2005], who is the designer of game Spore, people prefer to make their own content, in other words, they like to customise their own experiences.

PCG technique is a good solution that meets these challenges. Thus, the benefits of PCG are quite obvious. First of all, in the PCG process, contents are generated algorithmically rather than manually. Only rough data descriptions are needed to generate all content with great detail, so it can avoid a large amount of manual labour, and the repetition task of creating content using editing tools by hand [Smelik et al. 2011].
Secondly, PCG can potentially give better performance in terms of run time speed. Since 1985, the development of computational ability has increased much faster than the speed of memory access [Lagae et al. 2010]. PCG can rapidly reduce memory access times since content is generated by the program directly. This is a clear advantage in procedural mapping techniques. For procedural textures, the program can be written in High Level Shading Language (HLSL) and be run on the Graphics Processing Unit (GPU), which is designed for parallel processing.

Thirdly, PCG can save on the amount of space required to store data required for software applications. For most modern games, the resources which include images, textures, sound, models and other related content usually takes up large proportions of storage space. However, PCG uses code to generate most of them, so the size of the distributed software files can largely be reduced. One example is the First Person Shooter (FPS) game ‘.kkrieger’. This game was developed by a group named ‘.theprodukkt’, and all content such as textures, levels, audio, animations and character models in this game are procedurally generated (http://www.theprodukkt.com/kkrieger). The most impressive aspect of this game is its size. It only takes 96-kilobytes of disk space, which is even smaller than most 2D textures. Compared with some modern 3D FPS games which take more than 2 gigabytes space and need to be released on several CDs/DVDs, this game is amazingly small. While game features in terms of complex storylines, game plots, etc. cannot be represented in a 96-kilobyte game as compared with a full 2 gigabyte game, nevertheless, the quality of the graphics in the game is relatively comparable with most full featured games. The small game size means that it can simply be distributed online because players can easily download the complete game, instead of the traditional way of having to purchase the game media from a physical shop.

Lastly, PCG can be used to generate special content which are hard for artists to create. Some content are elusive to describe and hard to be simulated, such as smoke, fire and clouds. Modern PCG techniques can generate perfectly realistic models and animations for these natural phenomena, which are difficult for artists. This is because it will take many hours for a human to manually model and animate the complex details of naturally occurring phenomena like fire and clouds. However, if this were to be simulated on a computer using PCG algorithms, it would remove the need for an artist to perform this laborious and time consuming task.
The benefits offered by PCG techniques have resulted in a growing interest among the research community in developing different PCG techniques for a variety of different areas. While there is much research effort on areas like terrain generation, texture synthesis, plants and trees, etc. there is relatively little research on the generation of cave models. This thesis focuses on the procedural generation of visually plausible 3D cave models.

1.1 Major Issues
While virtual 3D caves are commonly featured in virtual environment applications, procedural cave generation is not an area that has received much attention among researchers to date. There are several challenging issues faced in the procedural generation of 3D cave models.

In particular, cave generation is very much related to terrain generation. However, traditional procedural terrain generation algorithms are typically based on height map techniques [Smelik et al. 2009]. These techniques have the limitation in that they can only handle basic terrains and are not suitable for storing cave models since a height map can only store a single height value for each coordinate on the horizontal plane. Thus, height map techniques cannot handle specific types of terrains with overhangs and arches, which is a fundamental requirement in the representation of caves [Gamito and Musgrave 2001].

Another big challenge in procedural cave generation lies in the efficient storage of the resulting 3D cave model data. Although some approaches attempt to use voxels, or volumetric elements, to store cave model data, the memory cost required to store large cave data is huge [Peytavie, et al. 2009; Boggus and Crawfis 2010]. As such, better methods have to be developed in order to address these cave generation issues.

1.2 Objectives
The main objective of this thesis is to investigate and develop a procedural approach to generating visually believable 3D cave models and to be able to present the resulting model to the user at interactive rates. The results of this research could potentially benefit areas like interactive computer games and virtual reality.

In order to achieve this, this research will have to develop a new approach to automate the process of generating caves. The cave model data will have to be generated and stored using an effective and efficient data storage structure. In
addition, the resulting cave model will have to be rendered and displayed in real-time in order for the user to be able to interact with the cave model. Finally, to increase the realism of the cave model, certain cave characteristics like stalactites and stalagmites will have to be simulated in the generation of the model.

1.3 Contributions

This thesis develops and examines a novel procedural approach to generating virtual 3D cave models. The 3D cave models are generated using noise functions, and by specifying certain parameters randomly generated cave models with different characteristics can be created automatically on the computer. This research also proposes a unique bottom-up voxel-based octree construction approach that can efficiently store the 3D spatial data required as part of the cave creation process. Once the boundary surfaces of the cave are determined from the octree, a method for smoothing the resulting cave walls is introduced along with a technique for patching any cracks that may arise as a result of the smoothing process. In addition, this research also presents an approach to procedurally add stalactites and stalagmites to the 3D cave model. The resulting 3D cave model is a triangle mesh that can be rendered in real-time using any polygon based rendering technique.

The novelty of this research work lies in the fact that to date, there is no current work in academic literature on the procedural generation of 3D cave structures using noise functions. This research presents an approach of creating believable cave structures with realistic cave features such as surface roughness, stalactites and stalagmites. Unlike previous work where some manual work by a human is required to generate such 3D models, this work is completely automated on a computer. In addition, this thesis compares the individual techniques that have been used in this research with related work in the respective areas. It highlights the difficulties that the proposed methods seek to overcome and present experimental results obtained from the implementation of the various techniques.

1.4 Publications

The research conducted for this thesis has resulted in the following journal publications, which are provided in the Appendix section of this thesis:


1.5 Structure of the Thesis

The rest of the thesis is organised as follows:

Chapter 2 contains a comprehensive literature review on relevant research work in the area of procedural content generation. This includes an overview of current PCG techniques, the status and challenges of procedural terrain generation techniques, along with other topics related to this research.

Chapter 3 discusses the four steps adopted in this research to procedurally generate the foundation of the cave model. This chapter also presents the voxel-based octree construction approach developed in this research.

Chapter 4 describes the method used in this work to perform smoothing operations on the boundary surfaces of the cave obtained from the technique discussed in Chapter 3. This chapter also presents a novel algorithm for fixing potential cracks that may occur in the polygonal mesh after the smoothing process.

Chapter 5 shows a procedural approach to simulating two typical phenomena that occur in natural caves, namely, stalactites and stalagmites. This chapter also shows how to merge these in the final cave model.

Chapter 6 concludes this thesis and provides direction for future work.
CHAPTER 2: LITERATURE REVIEW

This chapter presents a comprehensive literature review on related research work as well as on various topics that are relevant to this research.

The chapter starts with a survey of current PCG techniques, which includes the definition and advantages of procedural content generation, its classification and recent applications. Since caves are a specific type of terrain, the related techniques developed for procedural terrain generation are discussed. This also includes the main challenges faced in procedural terrain generation and the current solutions for these challenges.

In order to store the cave model, 3D data structures should be used. Thus, a review of spatial data structures is provided. This chapter also discussed the merits and disadvantages of various data structures. This is followed by an overview of flood fill algorithms that will be used in this research. In addition, noise is an important aspect of procedural content generation, and therefore three types of procedural noises are discussed. Finally, this chapter goes through the related work in the area of cave generation and the current research in the simulation of cave characteristics.

2.1 Procedural Content Generation

Procedural Content Generation (PCG) is a vital component in the automated construction of virtual worlds. PCG refers to the automated generation of scene content on a computer. This can be achieved using a variety of different procedural algorithms and techniques, which has been the topic of much research.

In general, there are two types of PCG. For the first type, all the content has to be generated before running the application, and the user cannot interact with these applications or software. An example of this can be seen in movie production. In these applications, the most important requirement of the generation algorithm is the quality and accuracy. Therefore, the final results should be as realistic as possible, while the performance in terms of time and memory consumption is given a much lower priority. The other type of PCG is where content is generated interactively in real time, such as in video games. In these applications, the generating algorithm should be fast enough for the user to perform interactive operations within the virtual
environment. As such, the balance between realism and performance is a key problem for these applications.

There are many different kinds of content that can be generated procedurally. In general, procedural generation techniques related to textures and models are among the most widely discussed areas [Ebert et al. 2003]. To date, many techniques and approaches have been developed to generate these computer graphics content.

Procedural modelling is an exciting research area. Natural and artificial content generation are two types of procedural modelling. Many naturally occurring phenomena and content can be simulated procedurally, including clouds [Ebert et al. 2003], fire [Fuller et al. 2007; Smith 2008], and smoke [Stam 2000] which can be simulated purely based on different procedural techniques. Many of these approaches can also be rendered interactively in real-time. As far as the generation of natural content is concerned, procedural terrain generation is one of the most successful and widely used areas of PCG [Olsen 2004]. Other natural content generation techniques including the automated generation of trees [Smith 2008] and ocean waves [Hinsinger et al. 2002; Jeschke et al. 2003] can also be simulated by a computer program using PCG techniques.

For artificial content, large urban model generation is a hot research topic. Kelly and McCabe [2006] gave a survey of the recent procedural techniques developed for the generation of virtual cities. Greuter et al. [2003] present their work on how to generate cities in real-time, while other articles discuss how to generate road networks [Galin et al. 2010; Sun et al. 2002], and buildings [Müller et al. 2006]. In addition, Shamus [2009] did a fantastic job in his creation of a pixel city.

Procedural texturing is another important part of PCG. Perlin [1985] made his famous marble texture using Perlin noise in 1985. To date, many realistic textures can be generated procedurally and applied to different application areas, such as wood, marble, stone, and metal textures [Ebert et al. 2003]. Since procedural texturing can be written in a shader language like HLSL and run on a GPU, lighting and shading calculations can always be merged together with procedural textures. Therefore, the performance in terms of speed can be improved compared to the normal techniques.
In fact, other scene content can also be generated procedurally, not just content related to computer graphics. For example, PCG techniques have also been used in crowd simulation [Maïm et al. 2008].

PCG techniques are also heavily used in computer games. Games like Diablo and Diablo II both use procedural techniques to generate maps and dynamically created levels. Torchlight which is a Diablo style game, also uses dynamically generated maps in flexible ways. In the game Spore, all maps and scenarios are generated randomly by the computer program, and the user can define their own characters with editing tools. This supplies players with a special and unique in-game experience. In addition, the game Majesty: The Fantasy Kingdom Sim presents players with a different style of Real-Time Strategy (RTS) game, where all the heroes in the game are not directly controlled by the player. Instead, the player needs to post a reward for a target in order to drive the heroes into action. In this game, procedural techniques are used to generate all the levels and scenarios. Minecraft is a sandbox building game, where almost everything is generated procedurally.

### 2.2 Procedural Terrain Generation

Procedural terrain generation is one of the most popular and successful instances of PCG, and is very much related to cave generation. Huge and detailed terrains can usually be seen in movies and real-time games. There are various techniques for procedural terrain generation ranging from physically-based techniques to purely artificially derived methods have been explored [Smelik et al. 2009].

Natural phenomena such as erosions can be simulated based on height map algorithms. Olsen [2004] discussed two normal types of terrain erosion, which are named thermal erosion [Musgrave et al. 1989] and hydraulic erosion [Kipfer and Westermann 2006; Benes et al. 2006]. Not only did Olsen [2004] present a comprehensive survey of existing erosion algorithms, but he also raised several methods to accelerate and improve the performance of the erosion algorithms.

There are many specific terrain characteristics that can also be created procedurally based on height maps and different procedural techniques. Smelik et al. [2008] showed an intuitional and interactive approach to generate terrains procedurally by combining different procedural techniques, Doran and Parberry [2010] proposed a novel procedural approach to generate terrain characteristics using different functioned intelligence agents, and Zhou et al. [2007] showed how to
generate terrains procedurally based on digital photographs or sketch drawings. Smelik et al. [2009] gave a good survey on the current development of procedural terrain generation techniques.

2.2.1 Height map algorithm

Most terrain generation algorithms are based on the height map approach [Musgrave 1993]. A height map is a 2D array (for instance $m \times n$) where each element of this array represents a height value. Normally, a greyscale image with the dimensions of $m \times n$ is used to store height map, and the value (ranging from 0 to 255) of each pixel on the image represents a height value. The next step is to create a triangle mesh based on height map by separating every rectangle into two triangles. Finally, the mesh is rendered with textures and lighting onto the screen. Figure 2.1 gives a depiction of this process.

Figure 2.1a shows a 2D greyscale image of a height map, where each pixel’s value represents the height value for a particular terrain coordinate, these height values range from 0 to 255. Figure 2.1b shows a top down depiction of a triangle mesh. From the top down view, it can be seen that the triangle mesh is simply a regular triangle grid. Thus, if the vertices of the triangle mesh do not have any height values, the terrain would simply be drawn as a flat plane. To construct a proper terrain, the values of the pixels in the image shown in Figure 2.1a are used as the height values for the vertices of the triangle mesh shown in Figure 2.1b. Therefore, the height values of the vertices of the triangle mesh will now range from 0 to 255.
This results in a 3D terrain model with proper contours. Figure 2.1c shows an example of a 3D terrain model that was constructed using this approach. Thus, the problem of how to generate this height map is the key to normal procedural terrain generation algorithms. Many different and efficient methods had been developed, such as midpoint displacement [Wing-Cheong et al. 1995] and generation algorithms based on procedural noise functions (such as Perlin noise [Perlin 1985]).

2.2.2 Layer data representation

One of the obvious disadvantages of procedural terrain generation techniques based on the height map algorithm is that it is hard to handle cliffs, arches and overhangs. This is because there can only be a single height value for each coordinate on the horizontal plane. Furthermore, to simulate a cliff, the surface and textures will not be realistic after being stretched based on height map techniques.

The layer data representation was proposed by Benes and Forsbach [2001] to store terrain structure. Compared with the 2D height map, the layer data representation uses more than one height value on each two dimensional terrain position. Although layer data representation uses more space to store terrain models, as it is more convenient to describe steep terrains features such as cliffs, because rock strata can be simulated using different but associated types of material. However, the layered data representation was developed to simulate thermal and hydraulic erosion on the surface of a terrain. In view of the fact that it can only deal with the surface of a terrain, it cannot be used to represent complex terrains with overhangs such as caves and arches. Therefore, a different approach will have to be used to represent cave structures, which is the basis of this research.

2.2.3 Voxels

Voxels, or volumetric elements, are volumetric data structures that are designed to store 3D data and can also be used to store terrain models [Samet 1990]. Compared with height map and layer data representation, voxels cost much more space to store the final result. However, voxels can handle all kinds of terrains, especially terrains with cliffs and overhangs [Greeff 2009]. So it is convenient to use voxels to store cave models.

An ideal data structure should be suitable to store and represent a volumetric object flexibly and efficiently. In general, a 3D array can directly be used to store
voxels. The advantage of this is obvious: a 3D array is clear and simple enough to represent, and it is convenient to do interactive operations, such as insertion and deletion operations. However, the disadvantages of using 3D arrays are also distinct: the memory requirements are very large to handle, and the cost grows exponentially with the size [Laine and Karras 2010]. Consider a 3D array with length 256, width 256 and height 256, even when using one byte to store each element, the array still takes up 16 MB of memory, which is extremely large to handle.

2.3 Challenges of Procedural Terrain Generation

Despite the progress and development of techniques in the area of procedural terrain generation, there are still a number of challenges that have to be overcome and various approaches have been proposed to solve these challenges.

A big challenge in procedural terrain generation is the lack of control in the resulting appearance of the terrain [Parberry and Roden 2004]. This issue not only occurs in procedural terrain generation, but is also a challenge in the whole field of PCG. Since terrain models are generated according to several rough and fuzzy parameters, it is hard to control the final output accurately. Moreover, it is also impossible to meet some requirements like particular characteristics at specific place, such as a mountain or a valley at a specific location on the terrain. Due to this challenge, several solutions have been proposed.

One solution is to generate terrains based on the real terrain heights or a rough sketch of the terrain [Zhou et al. 2007]. This provides a way to place all specific terrain features which you want to exist in the final model, in a feature tree at beginning, then to add noise and turbulence to alter this rough sketch and to increase the amount of detail.

The other approaches to dealing with this challenge include the use of agents separately to change results generated by traditional procedural terrain generation algorithms. These agents can be classified into different functions, such as smoothing agents, mountain agents and beach agents. These agents are controlled by input parameters, so the final terrain can meet the requirements of specific properties [Doran and Parberry 2010].

Another challenge of procedural terrain generation is caves and overhangs. This issue exists in most procedural terrain generation algorithms, since they usually
generate terrains based on a height map [Smelik et al. 2009]. For caves and overhangs, there is more than one height value at the same height map coordinate.

Additionally, realistic texture mapping is another challenge faced by procedural terrain generation algorithms. Since the height differences between adjacent sampling points can have large difference, a 2D texture map would make the resulting textures in the scene look non-uniform. One typical example of this is in the case of a cliff. Most of the time, the texture on cliff is stretched dramatically.

2.4 Spatial Object Modelling
A cave structure is a special case of a terrain. While some research work deal with how to use 2D height maps to generate and store cave models, it is better to store and render a cave model using 3D data. However, as previously discussed, the huge demand for memory costs greatly limits the usage of 3D volumetric data arrays.

In order to simplify the storage and reduce memory costs, tree structures are widely used in computer graphics. Tree structures are inherently defined to keep consistency and homogeneity. Compared to 3D data arrays, the memory cost of tree structures can be reduced, since operational complexity is directly based on a logarithmic ratio. Many different tree structures can be used to store and represent voxel data [Samet 1990], and three popular and distinct tree structures are the standard octree, point-region octree and kd-tree [Greeff 2009].

2.4.1 Octree structure
An octree is one of tree structures that can be used to store and represent volume data. In an octree, each internal node has exactly eight children. The root of octree usually represents the entire volume, and the leaf nodes of the octree are either 1 or 0, which indicates whether this node is totally inside or outside of the object (homogeneous), and there is no need to separate this node any more. The other nodes in the internal level have values between 0 and 1, and all of them contain eight child nodes [Samet 1990].

The typical octree construction method follows a top-down sequence. The root node stands for the whole octree, then the algorithm needs to check whether or not the objects intersect with this node. If not, assign this node to 0 or 1 (which means totally inside or outside of the objects), otherwise split this node into eight children and do the checking recursively until the maximum depth of octree is
reached or until it cannot be split any more, in other words, it is homogeneous. Figure 2.2 shows the representation of 2D case of octree. Figure 2.2a shows how to store a single point by subdividing a standard quadtree and Figure 2.2b shows the tree structure, where the point is stored in the coloured leaf node.

![Figure 2.2 Representation of a quadtree](image)

However, for some special cases, we do not know the final objects at the very beginning, before construction of the octree. All we have is a method which can be used to check whether a specific location of space is inside or outside the tree node. In such cases, the bottom-up construction strategy is much better than the top-down algorithm. In practice, there are two popular approaches. The first is to visit each tree element in the Morton order [Morton 1966]. The Morton order follows a ‘Z’ sequence. Figure 2.3 shows a 2D quadtree case which is visited in Morton order at a tree depth of 1 (Figure 2.3a), 2 (Figure 2.3b) and 3 (Figure 2.3c) respectively.

![Figure 2.3 Morton order for a quadtree](image)
For a 2D image, this algorithm checks each pixel once and only once to build the quadtree, as such no temporary node is required [Samet 1980]. By using this method, the leaf node is only created when it is needed. The second algorithm is the bottom-up construction approach and is suitable for handling the processing of huge data. For an image, the pixels of the image are processed one row at a time [Samet 1981], which is also called a row or raster-scan order. However, this solution costs more time because of some additional merging and node insertion operations.

Other than for the storing of spatial data, octrees can also be used to accelerate scene rendering. For example, some unnecessary objects or polygons can be rejected if they are not in the field of vision by applying octree techniques, ray casting algorithms [Laine and Karras 2010] and level of detail techniques [Wu et al. 2010].

2.4.2 Point-region octree

For a standard octree, the child nodes are always subdivided through the parent nodes’ geometric centre. However, for some cases, the standard octree is not efficient enough. For example, if we use a standard octree to store a terrain model, the only non-homogeneous part is focus on the bottom of octree, since most of the top of octree is blank, so it could be more efficient if the splitting position were to be moved downwards in the tree.

Point-region octree [Finkel and Bentley 1974] is an enhancement of standard octree, and for each node which has children, an extra property of point type is needed to hold the splitting position, and eight children could be uniquely determined based on this point. So in some sense, the standard octree is a special case of the point-region octree which always splits from the parent nodes’ geometric centre. Figure 2.4 below shows the 2D case of the different splitting method of standard quadtree and point-region quadtree. For the same point, a standard quadtree needs 3 levels to reach the point, and point-region quadtree needs only 2 levels to reach the same result. This advantage is more obvious for multiple objects and complex models. However, as previously mentioned, it needs more space to store the splitting points and needs more calculation to determine the positions of them. Figure 2.4 shows the 2D case of point-region quadtree, Figure 2.4a shows the splitting scenarios
2.4.4 Point-region quadtree

Figure 2.4 Representation of a point-region quadtree

2.4.3 Kd-tree

The kd-tree [Bentley 1975] is a kind of binary tree, where $k$ stands for the dimensionality of search space. Unlike the standard octree and point-region octree, each node of kd-tree has only two children no matter what the dimension, and these children are split based on a one dimensional axis. For the 2D case, the split sequence could be $x, y, x, y, \text{ etc.}$ or $y, x, y, x, \text{ etc.}$ and for 3D case the sequence could be $x, y, z, x, y, z, \text{ etc.}$ Figure 2.5 below shows a 2D example of kd-tree. Figure 2.5a shows the splitting scenario of kd-tree and points 1, 2, 3 and 4 are four separating points on each level. Figure 2.5b shows the resulting kd-tree structure and the letter L (left), R (right), U (up) and D (down) indicates the relative position of the two children on each level.
Compared to the standard octree and point-region trees, kd-trees use the least amount of memory since it has only two children for each non-leaf node. However, it has the deepest tree level which results in additional calculations to locate the specific node or to do interactive operations.

2.5 Spatial Object Rendering
In general, there are two kinds of spatial object rendering. One type is to represent the spatial objects as the entire volumes, which is called solid modelling, and the other type is to treat the spatial object as the iso-surface between solid and space which is called surface modelling.

2.5.1 Solid modelling
Solid modelling is a branch of geometric modelling. In solid modelling, any specified point in space should be part of the solid object or should be outside the object [Samet 1990]. Till now, six representations can be used to modelling solid objects. Since they are good in different aspects, the final representation is always a combination of them. These six representations are: the Boundary Model (BRep), Sweep Methods, Primitive Instancing, Constructive Solid Geometry (CSG), Spatial Enumeration and Cell Decomposition.

A popular and effective technique of solid modelling is the ray tracing algorithm [Levoy 1990]. The main idea of the ray tracing algorithm is to generate a 2D image by back tracing light rays from eye (or camera) through each pixel in the
The predecessor of ray tracing is ray casting, which was proposed in 1982 [Roth 1982] and can be considered as an abridged version of ray tracing, since the main principles between them are similar. Ray casting determines the colour on the image by back tracing the light paths from eye through each pixel on image and fetching the colour value of the first object which intersected the light ray. On the other hand, for ray tracing, the colour value could be calculated by combining light of absorption, reflection and refraction. Many models and techniques had been proposed to simulate realistic lighting models in nature [Lee et al. 1990], such as the Phong reflectance model [Phong 1975].

Compared to ray casting, the ray tracing algorithm generates more realistic images. In ray casting, the rays are traced from the eyes through the image pixels until they are intersected by objects. The main difference of ray tracing is that it considers the reflection and refraction of lights off objects, which means it traces light recursively, and this progress can be controlled by parameters such as maximum recursion count [Glassner 1989].

A major disadvantage of the ray tracing algorithm is the performance. Although commodity graphics rendering hardware is becoming more and more powerful, and some functions and APIs had been developed for ray tracing (such as NVIDIA Optix), it is still hard to implement ray-tracing algorithm in real-time, especially for complex models consisting of mirror-like materials. So for some cases, ray casting or surface modelling is a better choice, as it is simpler and faster.

2.5.2 Surface modelling

Surface modelling treats spatial objects as iso-surfaces between space and solid, and this iso-surface is a mesh based on triangles. Surface smoothing is discussed in the next section. The key is how to convert spatial objects into mesh based iso-surfaces.

The marching cubes algorithm [Lorensen and Cline 1987] is a famous algorithm to generate a seamless mesh based on volume data, and was first used in the visualisation of MRI scan results. This algorithm can transfer multiple 2D slices or volume elements into 3D surface polygons made up by triangles. Before implementing the marching cubes algorithms, the spatial object should be stored using 3D array with binary values (inside or outside of the objects). The word ‘cubes’ stands for the sub volume in this 3D array which consists of eight corners, and these corners are sampling points with adjacent coordinates, and the cubes is
confirmed with boundary points \((i, j, k)\) and \((i + 1, j + 1, k + 1)\). The marching cubes algorithm tries to fetch the whole iso-surface by going though all the cubes. According to the status the eight corners (inside or outside) for each cube, there are a total of 256 possible different surface configurations, and after topological analysis, 15 different types of cubes [Lorensen and Cline 1987].

These types merge 256 configurations by rotation and symmetry operations, and except for the case of 0, the triangle(s) should be a part of the final iso-surface. Then a lookup table can be constructed to match 256 configurations into these 15 types of surfaces. To save memory, a byte with 8 bit is suitable to describe 256 configurations as input of lookup table, and the lookup table returns a 12 bit number, with each bit corresponding to an edge. 0 means that the edge is not cut by the iso-surface, and 1 means the edge is cut by the iso-surface. The intersection points can be calculated by linear interpolation based on iso value and scalar values at each vertex if the iso-surface cuts the related edge. The marching cubes algorithms is efficient since most of the work has been done in constructing the lookup table.

Recently, many improvement and enhancement had been proposed to optimise the classic marching cubes algorithm from two different aspects. One is to reduce the triangle count generated by the marching cubes algorithm. Schroeder et al. [1992] proposed an efficient algorithm to decimating triangle meshes after the triangles had been generated using the marching cubes algorithm. According to the relationships of adjacent vertices, each vertex is assigned with one of five possible classifications (simple, complex, boundary, interior edge and corner). Then the decimation can be done based on different types and controlled by parameters. Results show that the number of triangles can be reduced by as much as 90 percent without obvious distortion. However, the disadvantage of this algorithm is that it needs to generate the triangles first, and the calculation is also huge to evaluate whether vertices should be deleted from the mesh. The octree structure can be used to improve the marching cubes algorithm by reducing the complexity of final triangle mesh [Wilhlems and Gelder 1992; Shekhar et al. 1996].

Another enhancement to the classic marching cubes algorithm is an attempt to eliminate the ambiguity error [Nielson and Hamann 1991]. For classic marching cubes topologies, seven cases are ambiguous as they have more than one solution to generate triangles. These can lead to holes and cracks on the final mesh. One novel solution is using tetrahedrons instead of cubes as the unit of generating iso-surface
[Bourke 1997]. In this algorithm, each cube is separated into six tetrahedrons, and each tetrahedron is assigned to one of eight types according to the vertices’ status (inside or outside of the object). Generating iso-surface polygons using tetrahedrons can remove the ambiguity of the classic marching cubes algorithm, and the result would look smoother for the same model.

Compared to solid modelling algorithms, surface modelling algorithms are much faster, and more convenient to implement in graphics rendering hardware. However, the lack of photorealism is a disadvantage of surface based modelling, especially for curve surface rendering.

2.5.2.1 Surface smoothing
A direct approach to surface smoothing is to calculate the vertex normals for each vertex on the surface. Since most vertices on a surface are shared between several triangles, surface will look smoother when only one normal vector is used for each vertex during lighting computations. Vertex normals can be calculated by averaging the normals of all the triangles that share this vertex.

Another approach is to use a weighted average of the normal vectors in order to calculate a single normal vector for that vertex. The assigned weights should be inversely proportional to the area of the triangle. The reason for this is because small triangles often occur at high curvature sections of a surface. The third way to generate vertex normals is based on the weighted sum of the normals of all the triangles that share that vertex, with the weight proportional to the arc cosine value of the angle shared by the two adjacent edges. No matter which approach is used to calculate the normal vector, the first step is to normalise all vectors.

Calculating vertex normals only makes the final surface ‘look’ smoother, as this only affects the lighting computations. In order to actually smooth the surface, the vertex positions need to be changed. The Laplacian smoothing function is widely used in surface smoothing [Buell and Bush 1973; Field 1988]. Equation 2.1 below shows an example of the Laplacian smoothing function.

$$v' = \frac{1}{2}v + \frac{1}{2n} \sum_{i=1}^{n} v_i$$

(2.1)

In this formula, $v$ stands for the original vertex, and $v_i$ stands for all the vertices adjacent to $v$, and $v'$ is the position after the change. So each adjacent vertex contributes the same weight to the final result.
2.5.2.2 Patching cracks

As previously mentioned, the ambiguity of the classic marching cubes algorithm can cause cracks to appear in the final surface. The cracks or overlaps may also occur after smoothing the surface or decimating the count of triangles [Phongthanapanich and Dechaumphai 2004]. To date, many algorithms and techniques had been developed to fix or to patch these cracks.

Barequet and Kumar [1997] proposed an algorithm for repairing errors in polyhedral CAD models, where errors refer to cracks, holes, duplications and overlaps. The main idea of this algorithm is to change the polygons on surface from an unordered collection to a shared-vertex representation, and this is done by recursively merging two appropriate polygon edges into one by shifting the vertex positions. So after the whole process, each polygon edge is either shared with another polygon’s edge or is a boundary edge.

El-Sana and Varshney [1998] tried to remove the potential cracks in polygon surfaces by constructing alpha prisms in conjunction with simplified polygonal surfaces, and then attempt to eliminate cracks based on topological logic.

Fakir and Greg [2003] showed an algorithm for the simplification and repair of polygonal models using volumetric techniques. First of all, the model based polygon surfaces are converted into voxel-based models, so the issues of cracks were transformed into the volumetric domain. 3D morphological techniques could then be used to remove holes in the voxel representation. Finally, this algorithm converts the solid model back into a polygonal model as the output.

Ju [2004] proposed a robust algorithm for repairing arbitrary polygonal models, and which is somewhat similar to the previous algorithm. A volume representation based on an octree grid is constructed to show whether each spatial point is inside or outside the model, and then to reconstruct the polygon surfaces by contouring. So finally, a closed surface which separates space into disjoint internal and external volumes is generated.

Wu et al. [2004] showed a reason for cracks when rendering terrains with LOD technique, and proposed an algorithm to patch crack by storing all the vertexes of the cracks in a list, and drawing triangles based on the crack list. However, this algorithm cannot handle situations where there are more than one intersection point.
for one edge or cases where more than one edge has intersection points, which may cause degenerate surfaces, and overlaps.

2.6 Flood Fill Algorithm
The flood fill (or also called seed fill) algorithm is widely used in multi-dimension arrays to find all the nodes connected to the given seed node based on a target node colour. In general, it takes three parameters, a starting seed node, the target colour and a replacement colour. There are many different ways to do flood fill, such as the standard flood fill, scan line fill and quick fill; each type could be implemented recursively or non-recursively (using a queue or a stack). The main disadvantage of the recursive flood fill implementation is that it can easily lead to stack overflow when the arrays are large, whereas the non-recursive flood fill implementation can avoid this issue in most of cases.

In view of the fact that practitioners found that there were too many duplicate sampling on the same element in the array using the standard flood fill algorithm, thus, the scan line fill approach was introduced. For each loop in the standard flood fill algorithm, only one element is filled, and all the potential elements are pushed onto stack. Compared with the standard flood fill algorithm, the scan line fill algorithm fills all the potential elements on the same line during one pass and puts new seeds onto adjacent scan lines if needed, as controlled by boolean flag. Compared to the original flood fill approach, the scan line fill algorithm’s performance is much better due to sharply reduced number of sampling.

One important feature of the flood fill algorithm is that it finds all the connected neighbours. When used in multi-dimension arrays, the neighbouring elements are easy to find. However, when used in tree structures as opposed to simple arrays, the scenario become more complicated. As such, a number of researchers have written several articles that discuss how to find neighbouring elements in quadtree data structures [Samet 1982; Hunter and Steiglitz 1979].

2.7 Procedural Noise
Procedural noise is extremely important to Procedural Content Generation (PCG). Since PCG techniques generate content based on several intuitional or non-intuitional parameters, rich visual information is needed in order to fill in the content details. In most cases, such detail is complex and intricate to make the final result
look realistic. Procedural noise is a successful and effective approach to generating such detail.

Procedural noise means that the ‘noise’ is generated by program code, and not accessed from data structures directly. For instance, a 2D texture generated using procedural noise techniques means that the colour of each pixel is calculated using algorithms and mathematical functions but not from manual painting or digital photographs [Lagae et al. 2010].

In most instances, noise means completely random noise, called ‘white noise’. White noise is simply a set of random numbers within a certain range, and there is no regular distribution pattern and no correlation between adjacent numbers. Examples of white noise can easily be found in real life, such as the screen image when television cannot receive station broadcasts on the current frequency or when there is no programme being broadcasted. White noise can be generated by programs using random number functions.

However, most of the time white noise is not useful in PCG. One reason is that white noise is not consistent, and there is no regular pattern. Consistent noise is important in PCG. This means for the same input parameters and conditions, the result should always be the same. For example, for a 2D random texture, the colour of each pixel should always be the same when the same coordinates of the pixel are given as input. Otherwise, this texture would change if we were to look at it from another angle. White noise does not need any input, so it is truly random, which also means we cannot control it. Another reason is that values in white noise are completely independent and have no correlation with adjacent values. As such, when white noise is sampled at high frequencies, it gives rise to ‘aliasing’ artifacts. The reason for aliasing is because there is no relationship between the adjacent points of white noise, and its energy is spread equally over all frequencies. Figure 2.6 White noise (1D and 2D case). Figure 2.6a shows the white noise in 1D whereas Figure 2.6b shows a 2D case.
Although white noise and procedural noise use the same term ‘noise’, there are obvious differences between them. Moreover, these differences bring benefits to PCG, which are described as follows.

The most important characteristic of procedural noise is its steady and consistent properties. As such, procedural noise is always called “apparently random” or “pseudorandom”, which means that the generated results of the noise function should always be the same when the input parameters are the same, the input parameters could be the location coordinates, or the attributes of objects. Therefore, procedural noise can be randomly accessed, and does not need to be stored into memory for use later on. This characteristic of procedural noise brings two benefits to PCG. The first is that it can save the memory storage requirements of a program, and the other one is that its contents can be generated relatively fast.

Since 1985, the computation ability of computers has increased much faster than memory access speeds. Thanks to the rapid increase in computational speed, many details with complex and specific patterns can be evaluated in real-time and interactively, which is important to applications such as video games. In fact, many programs spend most of their time reading from and writing to disk memory, and other calculation may have to be suspended and delayed. Procedural noise can be generated and calculated in real-time instead of requiring memory accesses, thus, the speed of program can be accelerated.

Procedural noise can be generated based on a few initial values or patterns (in the order of a few kilobytes), but can produce large results (in the order of megabytes). Therefore, the design of a procedural function should be compact and
elaborate to achieve this. In the past two decades, many types of procedural noise have been introduced based on different theories to meet various requirements.

Procedural noise is continuous and consistent for multi-resolutions, and this continuity is not restricted to discretely sampled data. The continuity is reasonable, two sampling points nearby should have similar noise values and the difference should be large for two sampling points far away from each other. For multi-resolutions, procedural noise can adapt to any resolution, from a distant view with low resolution and a close view with high resolution. This nice feature makes procedural noise a good solution of Level of Detail (LOD) issues [Luebke 2003]. Also, beautiful fractal images and animations [Mandelbrot 1983] which are self similar can be generated based on procedural noise.

Procedural noise is non-periodic or its periodicity is not obvious. In real life, no two objects are ever exactly alike. So periodic repeating would cause results to look artificial. However, some foundational pseudorandom functions of procedural noise are periodic, so the period can be made to be large to make the periodicity not conspicuous. Procedural noise is parameterised. We can control the final result by changing the properties of procedural noise function such as amplitude and frequency, so one type of procedural noise can generate a group of related noise as desired. Other than amplitude and frequency, other parameters can also be set to manage the noise values of procedural noise functions.

An ideal procedural noise function should have the following properties: it should generate repeatable pseudorandom values based on its inputs; the values generated by it should be within a given range, such as from -1.0 to 1.0; it should be band-limited, where the largest frequency should be about 1.0; its periodicity should not be obvious or give rise to regular patterns, so the period should be large; and it should be stationary and isotropic [Ebert et al. 2003]. Based on the mathematical theory behind of procedural noise, it can be classified into three categories; namely, lattice gradient noise, explicit noises, and sparse convolution noises [Lagae et al. 2010].

2.7.1 Lattice noises

Lattice noises are the most popular and successful procedural noises, that are widely used in PCG, especially in applications for generating textures. The basic idea of generating lattice noises is to randomly generate values (scalar values) or gradients
(vectors) on vertices of an integer lattice first, and to generate the noise values of other points by convolving or interpolating the values or gradients of the integer lattice’s vertices, surrounding the sampling points. The most famous type of lattice noise is Perlin noise.

2.7.1.1 Perlin noise

Perlin noise was invented in 1985 [Perlin 1985; Perlin 2002], and has been widely used in recent decades. Perlin noise can be found in many areas of PCG, which include natural texture generation, procedural terrains and increasing the realism of models which are hard to manually define, such as fire, smoke and cloud simulation. Perlin noise is classified as belonging to lattice noise. More precisely, it is a type of lattice gradient noise.

The first step to generating Perlin noise is to build integer lattices by calculating pseudorandom gradients for all vertices of integer lattices. The integer lattices are square meshes for the 2D case, and a cubic network for a 3D case. The vertex gradients consist of n scales (n equals to the number of dimensions) with unit length. In order to save memory and keep consistency, the gradients of integer lattices are pre-computed and saved as a lookup table. For most cases, 256 gradients are enough to generate Perlin noise. Then for a given sampling point, use a hash function to choose a gradient from the lookup table according to the coordinates.

The next step is to generate noise values of other noises by doing splined interpolation of the nearest vertices in the integer lattice which contains the sampling point. For a 2D case, there are four vertices, while there are eight vertices for a 3D case. Figure 2.7 shows the steps of this process for a 3D case.

Figure 2.7 Perlin noise calculation in 3D space
In Figure 2.7, there are seven steps to calculate noise value on point \( P \) in lattice with vertexes \( ABCDEFGH \):

- Calculate noise value of point \( a \) by doing interpolation of point \( A \) and point \( B \);
- Calculate noise value of point \( b \) by doing interpolation of point \( C \) and point \( D \);
- Calculate noise value of point \( c \) by doing interpolation of point \( E \) and point \( F \);
- Calculate noise value of point \( d \) by doing interpolation of point \( G \) and point \( H \);
- Calculate noise value of point \( e \) by doing interpolation of point \( a \) and point \( b \);
- Calculate noise value of point \( f \) by doing interpolation of point \( c \) and point \( d \);
- Calculate noise value of point \( P \) by doing interpolation of point \( e \) and point \( f \).

For the interpolation polynomial, \( 3t^2 - 2t^3 \) was initially used. However, \( 6t^5 - 15t^4 + 10t^3 \) was later used instead to generate smoother noise at the cost of more computation [Perlin 2002].

2.7.1.2 Simplex noise

Simplex noise is another type of lattice gradient noise [Perlin 2001] and Gustavson [2005] showed clearer descriptions of simplex noise in his paper [Gustavson 2005]. Compared to Perlin noise, simplex noise has a lower computational complexity and is easier to implement in hardware.

Simplex noise is similar to Perlin noise, in that it also involves the generation of pseudorandom gradients on integer lattices, and does cubic interpolation to get noise values of points within the lattices. Compared to Perlin noise, the greatest contribution of simplex noise is that it uses fewer vertices to do interpolation to get noise values for sampling points, because the basic shape of lattices for simplex noise are different from that of Perlin noise. The simplest and most compact shape of \( n \)-dimensional space is the shape with \( n + 1 \) edges, when the space is spread with these shapes, we call it a simplex grid. So in simplex noise, an equilateral triangle is the basic shape for 2D images instead of a square, whereas a tetrahedron is the basic shape for 3D space instead of a cubic. Due to this difference, the computational cost drops down to \( O(n) \) from \( O(2^n) \) for Perlin noise.

Another difference of simplex noise and Perlin noise is the way to calculate the noise value. Instead of doing interpolations of all integer lattices vertices gradients of Perlin noise, simplex noise generates noise values by using the summation of contributions of each corner vertex, which can also reduce the computational cost. So
generally speaking, the whole process of simplex noise is similar to Perlin noise. First, generate pseudorandom gradients for each vertex of simplex grid. Then for a given sampling points, determine which lattice (simplest shape) it belongs to, and make a summation of contributions of each vertex of this lattice as the final result.

2.7.2 Explicit noises

Explicit noises do not belong to procedural noises in the strictest sense. Explicit noises generate a mass of noise values in an explicit fashion at the same time, so these noise values are pre-generated and stored in memory. However, these types of noise are still different from fetching data directly from a data structure, so they can still be considered as procedural noises. Compared with normal implicit procedural noises, explicit noises cost much more memory to store the pre-generated noise values, in return for being less computational complex.

2.7.2.1 Midpoint displacement

Midpoint displacement is a typical technique of explicit noises [Fournier et al. 1982]. Olsen showed a clear description of this algorithm using 2D images, and showed how it could be used in procedural terrain generation [Olsen 2004].

The implementation of the midpoint displacement algorithm is to generate the values of all peaks of the whole area first (for 2D square space, there are four peaks to be generated, whereas the number is eight for a 3D case), and then recursively calculate the noise values of middle points between two points for which the noise values are already known. The new noise values are based on the average value of two peaks points, randomly offset by a turbulence factor within a certain range. According to the current recursive depth, this range is halved with that of the upper recursive depth. The midpoint displacement can generate noise which is an approximate $1/f$ noise. For 2D images, the midpoint displacement algorithm can be enhanced using the diamond-square algorithm [Koh and Hearn 1992]. Compared to the standard midpoint displacement algorithm, this can reduce the square shaped artificial effects.

2.7.2.2 Wavelet noise

Cook and DeRose [2005] invented wavelet noise in 2005 [Cook and DeRose 2005], and it quickly became a popular algorithm for generating noise procedurally. In their
paper, Cook and DeRose [2005] found that there are several disadvantages in Perlin noise, such as aliasing and the loss of detail.

There was another fundamental issue existing in most procedural noise functions that was also raised by Cook and DeRose [2005]. Texturing a 2D surface by sampling a 3D noise function is commonly used when rendering. However, the 2D texture would not be band-limited, even if the original 3D function is perfectly band-limited. So the balance between the loss of detail and aliasing cannot be solved by constructing a perfect band-limited 3D function.

Wavelet noise fixes these two major issues. To implement wavelet noise, a noise band tile $N$ is needed to be pre-generated. This is why wavelet noise belongs to explicit noises. This is done using four steps. For a 2D case, an image $R$ is created by filling random noise within a range, then down sampling $R$ to half size to create image $R_{\downarrow}$, then up sampling $R_{\downarrow}$ to full size, labelled $R_{\uparrow}$, then subtract $R_{\downarrow}$ from original $R$ to create the noise band image $N$. The wavelet analysis technique is used when doing the down sampling and up sampling process.

The next step is to use this noise tile $N$ to generate noise values for any position spatially by using any evaluation method of quadratic B-splines. Compared to Perlin noise, wavelet noise uses more memory to store the noise tile, which is acceptable by today’s memory standards. Wavelet noise also runs about 30% faster than Perlin noise to generate the same sized noise [Cook and DeRose 2005].

2.7.3 Sparse convolution noises

Sparse convolution noises are a type of procedural noise which is not based on pseudorandom numbers and lattice gradients or values. They generate noise values by summing several randomly positioned and weighted kernels. Lewis [1984] introduced the definition of sparse convolution noises [Lewis 1984, 1989]. Sparse convolution noises can change noise values by direct spectral control. Sparse convolution noises can be perfectly band-limited, and it costs less memory than lattice noises (such as Perlin noise) and explicit noises (such as wavelet noise). However, the speed of sparse convolution noises is its disadvantage. Two famous of sparse convolution noises are spot noise [Wijk 1991] and Gabor noise [Lagae et al. 2009].
2.7.4 Tools for changing noise distribution

2.7.4.1 Bias function

The bias function is a power curve defined over the unit interval, and can change the distribution of data [Ebert et al. 2003]. Bias functions are widely used in image processing, if you want to make something much darker and more transparent, set bias value small than 0.5; if you want to make something brighter and more opaque, set bias value larger than 0.5. Figure 2.8 below shows the pseudo code of a bias function.

```
//bias function
BIAS_FUNCTION (FLOAT b, FLOAT x)
    RETURN pow(x, log(b) / log(0.5))
```

Figure 2.8 Pseudo code of a bias function

2.7.4.2 Gain function

The gain function is an intuitive way of controlling the degree of a function’s distribution, of whether most of the time it is near the middle ranges or near the extreme ranges. By changing the gain value, the degree of fuzziness can be altered. We can set the gain value smaller than 0.5 to make the image or texture fuzzier or set gain value larger than 0.5 to sharpen an image or texture file. Figure 2.9 shows the pseudo code of a gain function.

```
//gain function
GAIN_FUNCTION (FLOAT g, FLOAT x)
    IF x < 0.5
        THEN
            RETURN BIAS_FUNCTION(1 - g, 2 * x) / 2
        ELSE
            RETURN 1- BIAS_FUNCTION(1 - g, 2 - 2 * x) / 2
        ENDIF
```

Figure 2.9 Pseudo code of a gain function

2.8 Procedural Cave Generation

Existing efforts in the construction and visualisation of 3D cave structures includes the use of scanning hardware to obtain accurate spatial data about actual cave structures [am Ende 2001]. The scanned spatial data can then be used to reconstruct a virtual representation of the real cave that can be visualised on a computer display. However, the 3D mapping of caves using this physical approach is an extremely
painstaking and time consuming process. Schuchardt and Bowman [2007] investigated whether the visualisation of complex 3D cave structures using immersive virtual reality provided a higher level of spatial understanding of such structures, which cannot be mentally visualised using traditional means such as 2D cave maps. The cave model used for their system was constructed from cave survey and measurement data obtained from an actual cave, and converting this information into a 3D cave model.

The procedural creation of synthetic 3D cave models has previously been investigated by Boggus and Crawford [2009a, 2009b and 2010]. Their work focuses on the procedural generation of solution caves, which are caves formed by rock being dissolved by acidic water. In their research, they applied knowledge about the formation of solution caves in order to create cave models for virtual environments. Their proposed method involved approximating water transport to create a coarse level of detail model for a cave passage. They also demonstrated methods of generating 3D cave models using cave patterns, and proposed that surface detail could be added using techniques like bump mapping and displacement mapping.

Johnson, et al. [2010] examined an approach of using a cellular automata-based algorithm for the real-time generation of 2D infinite cave maps, for the purposes of representing cave levels in video games. However, the generation of 3D caves maps using this approach was left for future work. Peytavie, et al. [2009] presented a framework for representing complex terrains, which includes caves, using a volumetric discrete data-structure. In addition, they proposed a procedural rock generation technique to automatically generate complex rocky scenes with piles of rocks. Their aim was to generate and display physically plausible scenes without the computational demand of physically-based simulations. Their approach mainly focused on using their unique data-structure for efficient interactive sculpting, editing and reconstruction using high level terrain authoring tools, as opposed to a purely procedurally driven approach.

2.9 Cave Characteristics

There are many different kinds of speleothems that can be found in natural cave, such as drapery, straws, column and flowstone, and two typical instances of them are stalactites and stalagmites. The growth of stalactites and stalagmites is due to water dissolving some of the limestone. Although the physical and chemical mechanisms
underlying the growth of stalactites are well known, yet research in the simulation of the stalactite shape itself is relatively unexplored.

Tutnov et al. [1997] described the simulation of power plant fuel elements behaviour in complex and accident conditions, such as heat, mechanical and hydraulic issues, and one part of the code could be used to simulate the stalactites. However, this paper is focusing on the simulation of destroyed rods imitators and the fuel element accidental deformation, and does not give a procedural approach to simulate stalactites.

Short et al. [2005] analysed the chemical factor of stalactites, and considered the growth of stalactite as a free-boundary problem. Based on this, a physical model was built to find a universal geometric equation to describe the shape of stalactites. Finally, this paper raises a formula, which indicates the relationship between the length and radius of stalactites, and the comparison between simulation results and the observed stalactite shows this formula is quite accurate.

Kim et al. [2006] invented a new approach to modelling ice dynamics based on thin-film Stefan problem, and many natural phenomena could be simulated, such as icicles and stalactites. The final result of this novel physically-based algorithm is perfect to pass the comparison test of realism. However, one big issue of this approach is the time cost. Since it is physically-based, a large number of iterations are needed to run in the previous, and it would cost half an hour to generate a 2D picture. So it is impossible to do it in real-time.

Tortelli and Walter [2009] presented an approach for modelling speleothems growth based on geological studies. In their work, they took advantage of the powerful computational capabilities of GPUs to model the genesis and growth of stalactites, stalagmites and columns, in real-time from a set of meaningful geological parameters. However, their work is not suitable for this research because the purpose of this research is not to grow speleothems, but rather to add visually plausible stalactites and stalagmites to a 3D cave model.
CHAPTER 3: GENERATING THE CAVE FOUNDATION

This chapter describes the approach developed in this research on how to generate triangle-based boundary surfaces of a cave foundation procedurally. First of all, a distribution function is constructed to check whether or not the material of a spatial sampling point is stone or air, which means inside or outside the cave model. Then a novel approach is proposed to build an octree structure, based on the distribution function and a scan line fill algorithm, which is used to store the cave model. In order to remove isolated floating stones outside of cave model and air bubbles inside of cave model, a flood fill algorithm is applied in octree structure, and the boundary surface of the cave foundation could be determined at the same time.

3.1 Constructing the Distribution Function

The fundamental idea behind this approach to generating the cave structure is to be able to separate air from stone in a given 3D spatial area. This means that for a given 3D coordinate in space, we need to get the material (stone or air) of that point using a distribution function that can give us the material at that point.

The 3D area in question will form the foundation of the cave structure as it will provide the overall cave shape. For simplicity, basic volumetric 3D shapes like spheres, ellipses, cuboids, cylinders, cones, etc. can be used as the cave foundation, and these basic shapes can easily be connected and composited together to form entire cave systems, with passages, crevices, caverns, etc. For a simple case, we can generate a cave based on spherical shape. If we want to build a sphere based cave model with centre \( C \) and radius \( R \), the material distribution function is defined as follows: For a given point \( P \), calculate the distance between \( P \) and \( C \). If the distance \( PC \) is larger than \( R \), then the material at \( P \) should be stone (inside of cave model), otherwise the material should be air, which means \( P \) is outside of cave model. Figure 3.1a shows the 2D case for this process.

To produce the randomness which is a key feature of procedural cave structures, a 3D noise function will be used to create the overall distribution of air and stone. A 3D Perlin noise function was employed for this work. Note that other 3D noise functions can also be used which will result in different distributions and may potentially give rise to different cave structures. Figure 3.1b shows the 2D cross...
section of Perlin noise with its values ranging from 0.0 to 1.0, which corresponds to greyscale value from 0 to 255.

In addition, in order to control the roughness of cave model, a bias function was used in conjunction with 3D Perlin noise to control and facilitate the smooth separation between air and stone at the walls of the cave. Thus, different cave structures result from adjusting the bias parameter $b$. For a specific point $P$ in space, bias value $bias(P)$ was calculated based on the result of dividing the distance $PC$ by radius $R$, and limiting its range from between 0.0 and 1.0. Figure 3.1c shows a 2D cross section with bias value $b$ equals 0.05, which changes the distribution of Figure 3.1a.

The next step is to combine the Perlin noise and the bias function together to add randomness to the sphere. For a specific point $P$ in space, calculate the bias value $bias(P)$ and Perlin noise value $noise(p)$. If the $bias(P)$ is greater than the $noise(P)$, the material of this point is stone (inside the cave model); otherwise it belongs to air (outside the cave model), Figure 3.1d shows the result of a 2D cross section.

At this stage, the distribution of stone and air already looks like a cave foundation. However, one can see small ‘floating islands’ in the resulting cave structure in Figure 3.1d, which will have to be removed. Additionally, there are also some small ‘air pockets’ in the stone which are redundant to the final cave’s display and should therefore be removed.
Figure 3.1 Process for constructing the material distribution function

Figure 3.2 below gives the pseudo code for the distribution function.

```
//bias function
BIAS_FUNCTION (T,B)
    RETURN pow(T, log(B) / log(0.5))

//this function is used to check a spatial point is belonging to
//stone or air
DISTRIBUTION_FUNCTION (point P, float BIASVALUE)
    SET the distance between point P and geometric centre of cave to DISTANCE
    SET the result of BIAS_FUNCTION(DISTANCE, BIASVALUE) to BIASRESULT
    SET the Perlin noise value at point P to NOISERESULT
    IF BIASRESULT > NOISERESULT
        THEN
            RETURN STONE
        ELSE
            RETURN AIR
```

Figure 3.2 Pseudo code for the DISTRIBUTION_FUNCTION
3.2 Building the Voxel-based Octree

Storing all the spatial information in 3D array directly would lead to large memory costs, therefore the next step was to develop an approach to storing the spatial information into a tree structure. As described in chapter 2, there are many types of tree structures for storing spatial information, such as octrees, point-region octrees and kd-trees. Although point-region octrees and kd-trees have the advantage in terms of lower memory storage requirements, the octree spatial data structure is more flexible and less complex when it comes to interactive operations, such as inserting or deleting nodes from the tree. Since the distribution of spatial data is initially unknown when constructing the cave model, this would make it difficult to construct a point-region octree and kd-tree. For these reasons, an octree was chosen as the data structure to store the cave model.

Before the whole construction process, the basic information about octree must be explicitly defined. Figure 3.3 shows the relative positions of children nodes for a non-leaf octree node (note that child number 4 is behind child number 7). Table 3.1 Structure of octree node in turn shows the internal structure of an octree node.

Figure 3.3 Relative positions of children node for an octree in 3D space
### Internal data stored by an octree node

<table>
<thead>
<tr>
<th>Name</th>
<th>Data type</th>
<th>Meaning</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>centre</td>
<td>VECTOR3</td>
<td>The centre of this node</td>
<td>(0,0,0)</td>
</tr>
<tr>
<td>length</td>
<td>FLOAT</td>
<td>The length of this node</td>
<td>0</td>
</tr>
<tr>
<td>depth</td>
<td>INTEGER</td>
<td>The depth of this node</td>
<td>0</td>
</tr>
<tr>
<td>children[]</td>
<td>octree node POINTER</td>
<td>Pointer to the children, if any</td>
<td>Null</td>
</tr>
<tr>
<td>material</td>
<td>ENUM type (SOLID, AIR or JUNCTION)</td>
<td>The material of this node</td>
<td>Null</td>
</tr>
</tbody>
</table>

Table 3.1 Structure of octree node

To describe the octree construction method, a 2D case using a quadtree is used for simplicity to illustrate the approach. This algorithm can easily be extended to a 3D case using an octree. To start the explanation, the initial point distribution is shown in Figure 3.4a, where S stands for stone element and A stands for air element. The composition of air and stone can be dynamically found on the fly, using the method described in the preceding section (i.e. section 3.1 of this thesis).

Since the final model is not known prior to constructing the octree, the normal top-down construction approach for building an octree runs into difficulties when trying to check whether a large node should be separated into smaller nodes, unless all (or most) of the unit elements included in this node have already been visited. This would mean that there will be a lot of duplicate sampling of the same spatial locations, thus wasting computation. So it is better to construct the octree using a bottom-up approach. To do this, for each point in space, create a leaf node at the deepest level of the octree, and insert them one by one into the octree. In order to determine which leaf node to insert into the octree, each node must be assigned a unique index. For example in a 2D scenario, if there is a quadtree with a maximum depth of 3, the index of a node is an array of length 3, with each value of the array ranging between 0 and 3 since each quadtree node can have a maximum of four children, provided it is not a leaf node. The child node indices of the quadtree follow the Morton order (or called Z order). So there are in total 64 nodes at the deepest level of the quadtree with indices from 000 to 333, as shown in Figure 3.4b.
Figure 3.4 Initial state of the quadtree

For the 2D quadtree case with maximum depth 3, all the nodes at the deepest level should not be separated any more, which means that the material of these nodes will either be stone or air, and not mixed. As the quadtree is to be built using a bottom-up approach, the relationship between the spatial points and the leaf node indices at the deepest level need to be found. Therefore a method must be implemented to return the leaf quadtree index at the deepest level for any given point. This process can be done from top-down by comparing the point coordinate with the centre coordinate recursively until the maximum depth, and transferring the centre to one of its child’s centre.

For the 2D quadtree with a maximum depth of 3 described above, assuming the root coordinate of the quadtree is (0, 0), the length of quadtree is 8. Therefore, given a point coordinate of (-2.5, 0.5), this must first be compared with (0, 0). Since -2.5 is smaller than 0 and 0.5 is greater than 0, the first element of the index array should be 0. Then offset the centre (0, 0) to the new centre (-2, 2), and make the comparison again. The final the quadtree index should be 023 for the sampling point (-2.5, 0.5). Figure 3.5 shows the whole process of quadtree and the pseudo code is given in Figure 3.6 for an octree which follows the right-handed coordinate system.
Figure 3.5 Process of getting the quadtree index

//Assume the octree with root CENTER, length LENGTH, and maximum //depth MAXDEPTH, this method would return the tree index of given //point P

GETTREEINDEX(point P)
  SET LENGTH / 2 to OFFSET
  SET CENTER to C
  Create integer array INDEX
  FOR LEVELINDEX = 0 to MAXDEPTH step 1
    SET OFFSET_LENGTH /2 to OFFSET_LENGTH
    //go to children 0 for next loop
    IF P.x < C.x AND P.y >= C.y AND P.z < C.z THEN
      SET C.x - OFFSET to C.x
      SET C.y + OFFSET to C.y
      SET C.z - OFFSET to C.z
      SET 0 to INDEX[LEVELINDEX]
      CONTINUE
    ENDIF
    //go to children 1 for next loop
    IF P.x >= C.x AND P.y >= C.y AND P.z < C.z THEN
      SET C.x + OFFSET to C.x
      SET C.y + OFFSET to C.y
      SET C.z - OFFSET to C.z
      SET 1 to INDEX[LEVELINDEX]
      CONTINUE
    ENDIF
    //go to children 2 for next loop
    IF P.x >= C.x AND P.y >= C.y AND P.z >= C.z THEN
      SET C.x + OFFSET to C.x
      SET C.y + OFFSET to C.y
      SET C.z + OFFSET to C.z
      SET 2 to INDEX[LEVELINDEX]
      CONTINUE
    ENDIF
    //go to children 3 for next loop
    IF P.x < C.x AND P.y >= C.y AND P.z >= C.z THEN
      SET C.x - OFFSET to C.x
      SET C.y + OFFSET to C.y
      SET C.z + OFFSET to C.z
      SET 3 to INDEX[LEVELINDEX]
      CONTINUE
    ENDIF
  ENDFOR
SET C.x + OFFSET to C.x
SET C.y + OFFSET to C.y
SET C.z + OFFSET to C.z
SET 2 to INDEX[LEVELINDEX]
CONTINUE
ENDIF
//go to children 3 for next loop
IF P.x < C.x AND P.y >= C.y AND P.z >= C.z THEN
SET C.x - OFFSET to C.x
SET C.y + OFFSET to C.y
SET C.z + OFFSET to C.z
SET 0 to INDEX[LEVELINDEX]
CONTINUE
ENDIF
//go to children 4 for next loop
IF P.x < C.x AND P.y < C.y AND P.z < C.z THEN
SET C.x - OFFSET to C.x
SET C.y - OFFSET to C.y
SET C.z - OFFSET to C.z
SET 0 to INDEX[LEVELINDEX]
CONTINUE
ENDIF
//go to children 5 for next loop
IF P.x >= C.x AND P.y < C.y AND P.z < C.z THEN
SET C.x + OFFSET to C.x
SET C.y - OFFSET to C.y
SET C.z - OFFSET to C.z
SET 0 to INDEX[LEVELINDEX]
CONTINUE
ENDIF
//go to children 6 for next loop
IF P.x >= C.x AND P.y < C.y AND P.z >= C.z THEN
SET C.x + OFFSET to C.x
SET C.y - OFFSET to C.y
SET C.z + OFFSET to C.z
SET 0 to INDEX[LEVELINDEX]
CONTINUE
ENDIF
//go to children 7 for next loop
IF P.x < C.x AND P.y < C.y AND P.z >= C.z THEN
SET C.x - OFFSET to C.x
SET C.y - OFFSET to C.y
SET C.z + OFFSET to C.z
SET 0 to INDEX[LEVELINDEX]
CONTINUE
ENDIF
ENDFOR
RETURN INDEX

Figure 3.6 Pseudo code for the GETTREEINDEX function

A scan line fill algorithm is then used to control the sequence of nodes to insert into the octree. The normal scan line fill algorithm visits the elements line by line, and replaces the old colour with a new colour. The insert algorithm developed in this research was redefined for the purposes of octree node insertion. The main idea is use the scan line fill algorithm to control the node insertion sequence. The sequence
is as follows: For a given seed representing a node containing stone, get the octree index according to the previous defined function (using the pseudo code given in Figure 3.6), then check whether or not this node already exists in the quadtree. If not, check the material of this point using the distribution function (using the pseudo code given in Figure 3.2) to determine whether this node should be inserted into the final octree structure or not. The check as to whether the tree index already exists in the octree structure can be done by trying to visit this node following the relative tree index. The pseudo code listed in Figure 3.7 and Figure 3.8 are defined for these checking processes, respectively.

```plaintext
//Assume the octree with maximum depth MAXDEPTH, this method would //return whether a given tree index INDEX is exists in the octree //structure or not
CHECKEXISTS(INDEX)
    SET the point of root to POINT_OF_OCTREE
    FOR LEVELINDEX = 0 to MAXDEP
        TH step 1
            IF POINT_OF_OCTREE->children[INDEX[LEVELINDEX]] is NULL THEN
                RETURN FALSE
            ENDIF
            IF POINT_OF_OCTREE->children[INDEX[LEVELINDEX]]->material is SOLID THEN
                RETURN TRUE
            ENDIF
            IF POINT_OF_OCTREE->children[INDEX[LEVELINDEX]]->material is JUNCTION THEN
                SET POINT_OF_OCTREE->children[INDEX[LEVELINDEX]] to POINT_OF_OCTREE
                CONTINUE
            ENDIF
        ENDIF
    ENDFOR
ENDFOR
```

Figure 3.7 Pseudo code for the CHECKEXISTS function

```plaintext
//This method is used to check whether a leaf node should be //inserted into octree structure
CHECKINSERT_OR_NOT(Point P)
    GET INDEX by call GETTREEINDEX with P
    GET RESULT of CHECKEXISTS with INDEX
    IF RESULT equals to TRUE THEN
        RETURN FALSE
    ELSE
        GET MATERIAL by call DISTRIBUTION_FUNCTION with P
        IF MATERIAL is STONE
            RETURN TRUE
        ELSE
            RETURN FALSE
        ENDIF
    ENDIF
ENDIF
```

Figure 3.8 Pseudo code for the CHECKINSERT_OR_NOT function
The insertion operation is to add a new leaf node according to the tree index. The implementation is to build a path from the root to the destination node, and assign all other nodes which are brother nodes according to tree index to null value if they do not exist. For the 2D case example above, if the quadtree index of the first insertion node is 000, then four children on the first level should be created with children of tree index 1, 2 and 3 with null value (node 0, 00 and 000), only allocation space for children with tree index 0 and the material of this node should be assigned with mixed, which means this node can be separated. Then children 01, 02, 03 should be assigned with null value, etc. So after insertion operation of tree index 000, children with index 0, 00 and 000 should be created with allocate space, and children with index 1, 2, 3, 01, 02, 03, 001, 002, 003 should also be created with null values.

After each insertion operation, a merge operation should also be done if needed. The merge operation is implemented by checking all the brother nodes at the deepest level according to the tree index first. The merge operation will be performed if all brother nodes are not null and their material, \( m \), are the same. Then these child nodes are deleted, and the material, \( m \), is assigned to its parent node. Then the brothers at the higher level are checked recursively, until the root node is reached or no other merge operations can be performed.

When all the insertion and merging operations are complete, a trim operation is done to replace all null nodes with a type of material. So at the end, the whole octree will be constructed with stone nodes, air nodes and mixed nodes, whilst at the same time the isolated floating stones would have been removed. The pseudo code for these operations is provided below; insertion operation (Figure 3.9), merge operation (Figure 3.10) and trimming operation (Figure 3.11).

```plaintext
//Insert node with tree index INDEX into tree structure with root ROOT and maximum depth MAXDEPTH, call merge function to check whether merge operation is needed
INSERTNODE(INDEX, ROOT)
SET ROOT to POINT_TO_OCTREE
FOR LEVELINDEX = 0 to MAXDEPTH step 1
  IF POINT_TO_OCTREE->children equals NULL THEN
    IF LEVELINDEX not equals MAXDEPTH THEN
      //not reached the lowest level
      //and Create children nodes pointers for this node
      Allocate space for children field of POINT_TO_OCTREE
    SET JUNCTION to POINT_TO_OCTREE->material
    FOR CHILDRINDEX = 0 to 7 step 1
      IF INDEX[LEVELINDEX] equals CHILDRINDEX
        //node
      ELSE
        SET CHILDRINDEX to POINT_TO_OCTREE->material
      ENDIF
    ENDFOR
  ELSE
    //reached the lowest level
    //Allocate space for children field of POINT_TO_OCTREE
    SET JUNCTION to POINT_TO_OCTREE->material
    FOR CHILDRINDEX = 0 to 7 step 1
      IF INDEX[LEVELINDEX] equals CHILDRINDEX
        //node
      ELSE
        SET CHILDRINDEX to POINT_TO_OCTREE->material
      ENDIF
    ENDFOR
  ENDIF
ENDFOR
```

After each insertion operation, a merge operation should also be done if needed. The merge operation is implemented by checking all the brother nodes at the deepest level according to the tree index first. The merge operation will be performed if all brother nodes are not null and their material, \( m \), are the same. Then these child nodes are deleted, and the material, \( m \), is assigned to its parent node. Then the brothers at the higher level are checked recursively, until the root node is reached or no other merge operations can be performed.

When all the insertion and merging operations are complete, a trim operation is done to replace all null nodes with a type of material. So at the end, the whole octree will be constructed with stone nodes, air nodes and mixed nodes, whilst at the same time the isolated floating stones would have been removed. The pseudo code for these operations is provided below; insertion operation (Figure 3.9), merge operation (Figure 3.10) and trimming operation (Figure 3.11).
Allocate space of this child
ELSE
SET POINT_TO_OCTREE->children[CHILDINDEX] to NULL
END
ENDIF
ELSE
SET SOLID to POINT_TO_OCTREE->material
ENDIF
ENDIF
ENDFOR
CALL merge function with INDEX
RETURN

Figure 3.9 Pseudo code for the INSERTNODE function

//Merge operation of index INDEX
MERGE(INDEX)
FOR BOTTOMLEVEL = MAXDEPTH -1 to 0 step -1
SET ROOT to POINT_TO_OCTREE //from root for each time
FOR LEVELINDEX = 0 to BOTTOMLEVEL step 1
SET POINT_TO_OCTREE to POINT_TO_OCTREE->children[INDEX[LEVELINDEX]]
ENDFOR
//check whether all brother nodes’ material is solid
FOR CHILDIINDEX = 0 to 7 step 1
IF POINT_TO_OCTREE->children[CHILDIINDEX]->material is not equals to solid
THEN
RETURN
ENDIF
ENDFOR
//delete children nodes and assign parent node material to solid
FOR CHILDIINDEX = 0 to 7 step 1
DELETE POINT_TO_OCTREE->children[CHILDIINDEX]
ENDFOR
DELETE POINT_TO_OCTREE->children
SET solid to POINT_TO_OCTREE->material
ENDFOR
RETURN

Figure 3.10 Pseudo code for the MERGE function

//Trimming operation, should be called at last
TRIM(POINT_TO_OCTREE)
IF POINT_TO_OCTREE equals to NULL
THEN
Allocate space of POINT_TO_OCTREE
SET air to POINT_TO_OCTREE->material
ENDIF
IF POINT_TO_OCTREE->material equals to junction
FOR CHILDIINDEX = 0 to 7 step 1
TRIM(POINT_TO_OCTREE->children(CHILDIINDEX))
ENDFOR
ENDIF

Figure 3.11 Pseudo code for the TRIM function
For the 2D quadtree example, if the tree index for the seed point is 000, then the indices of the insertion sequence should be 000, 001, 010, 011, 100, 101, 111, 113, 112, 131, 133, 132, 311, 313, 312, 331, 333, 332, 323, 322, 233, 232, 223, 222, 220, 202, 200, 201, 022, 020, 002, 003, and 013. The merge operation should happen after the node with index 113 is inserted. Figure 3.12 shows the key steps of the whole process of building the tree structure. Figure 3.12a shows the state of the tree before inserting node 112, and Figure 3.12b shows the state of the tree after the insertion of node 112 and the merging operation. Figure 3.12c shows the state of the tree after all the insertion and merging operations, and Figure 3.12d shows the results after the trimming operation.

![Figure 3.12 Key steps in the process of building the quadtree](image)
At this stage, it can be seen that the approach developed for this research builds the tree structure using a series of operations and all nodes in tree will contain some material and will not be null, which means the whole tree is filled with stone and air nodes. For the 2D quadtree case, when comparing Figure 3.14a and Figure 3.4d, one can see that the stones floating in the air that are not connected to the walls of the cave have been completely removed.

### 3.3 Determining the Boundary Surfaces

The next stage deals with the removal of small ‘air pockets’ from the stone. Examples of this can be seen in Figure 3.1d and Figure 3.12d (node 002). At the same time, the boundary surfaces that separate stone from air are also determined. In the case of a 3D octree, the boundary surfaces are the internal cave walls.

The approach used for this is based on the flood-fill algorithm (also known as seed-fill), which is traditionally used to fill connected neighbouring pixels with the same colour. The initial seed node must be a node containing air that is located somewhere inside the cave. The algorithm then recursively searches for neighbouring air nodes until it discovers a stone node. Specifically, a point in the cave is provided as the seed, and the flood fill algorithm is used to fill all the space connected to the seed. This means that upon completion of the flood-fill process, any ‘air pockets’ in the stone will have been removed.

Applying the flood fill algorithm can be done in several steps. First, for a given point, the octree leaf node which contains this point must be found. Second, the neighbourhood relationship in octree structure needs to be redefined. The first step can be done by checking whether or not a specific node contains this point. If this node contains the point and has children, recursively check its children until a leaf node is reached. The implementation of this approach is similarly to get octree index function, previously discussed in section 3.2. However, the search process should stop if there is no child node in the search path. The pseudo code for this is shown in Figure 3.13.

```cpp
//This method is used to check whether a specific octree node contains a given point P or not
CONTAINS(POINT_TO_OCTREE, point P)
    SET POINT_TO_PCTREE->length/2 to OFFSET
    IF P.x <= POINT_TO_PCTREE->center.x - OFFSET
    OR P.x > POINT_TO_PCTREE->center.x + OFFSET
    THEN
```

RETURN FALSE
ENDIF
IF P.y <= POINT_TO_PCTREE->center.y - OFFSET
OR P.y >  POINT_TO_PCTREE->center.y + OFFSET
THEN
  RETURN FALSE
ENDIF
IF P.z <= POINT_TO_PCTREE->center.z - OFFSET
OR P.z >  POINT_TO_PCTREE->center.z + OFFSET
THEN
  RETURN FALSE
ENDIF
RETURN TRUE

Figure 3.13 Pseudo code for the CONTAINS and FIND_CONTAINS_NODE functions

The next step is to determine the neighbourhood relationship between octree nodes, which is different from 3D arrays, as the size of the neighbour nodes can be different since they can belong to different levels. Therefore, the algorithm for finding neighbouring nodes in octree can be done by in two steps: First, all neighbour nodes with same size or larger size are found, which means that the level of these nodes is either smaller or equal to the seed node. Second, check whether or not these nodes can be separated. If these nodes are not leaf nodes and can be separated, divided them until the leaf nodes are reached, and use these child nodes instead.

The first step in this approach can be done by generating 6 new points in space by offsetting the centre of node which contains original seed point with the length of this node. Then for each new seed, find the containing nodes with smaller or equal level. Due to the nature of the octree structure, if some of them are out of range, which means the sampling points is not even in the root node range, this means that
the node which contained original seed point is on the boundary of octree. The process of finding the container nodes with limited level is similar to finding container nodes discussed previously, and the pseudo code for this is given in Figure 3.14.

//This method is used to find the leaf octree node contains give point P, and the octree is with root node ROOT, the node’s depth is not greater than LIMITDEPTH
FIND_CONTAINS_NODEWITHDEPTH(point P, LIMITDEPTH)
  IF the result of CONTAINS(ROOT, P)is FALSE THEN
    RETURN NULL
  ENDIF
  SET ROOT to POINT_TO_OCTREE
  WHILE POINT_TO_OCTREE-›material equals junction
    FOR CHILDINDEX = 0 to 7 step 1
      IF the result of CONTAINS(POINT_TO_OCTREE, P) is TRUE THEN
        SET POINT_TO_OCTREE-›children[CHILDINDEX] to POINT_TO_OCTREE
        IF POINT_TO_OCTREE-›depth equals LIMITDEPTH THEN
          RETURN POINT_TO_OCTREE
          BREAK
        ENDIF
      ENDIF
    ENDFOR
  ENDMETHOD
  RETURN POINT_TO_OCTREE

Figure 3.14 Pseudo code for the FIND_CONTAINS_NODEWITHDEPTH function

The second step of dividing nodes is more complicated than the first step. The octree nodes must be divided into different child nodes with different child indices. Figure 3.15a shows the relative positions of the children to each other, and Figure 3.15b shows the six direction surface of node. Table 3.2 in turn, gives the child indices for each surface, which can be used as a lookup table.
Table 3.2 Child indices for the six facing directions

<table>
<thead>
<tr>
<th>Face direction</th>
<th>Child indices</th>
</tr>
</thead>
<tbody>
<tr>
<td>UP</td>
<td>0, 1, 2, 3</td>
</tr>
<tr>
<td>DOWN</td>
<td>4, 5, 6, 7</td>
</tr>
<tr>
<td>LEFT</td>
<td>0, 3, 4, 7</td>
</tr>
<tr>
<td>RIGHT</td>
<td>1, 2, 5, 6</td>
</tr>
<tr>
<td>FRONT</td>
<td>2, 3, 6, 7</td>
</tr>
<tr>
<td>BACK</td>
<td>0, 1, 4, 5</td>
</tr>
</tbody>
</table>

Based on the lookup table, an octree node can be separated into four child nodes with different child indices. For example, if an octree node, \( n \), is the right neighbour of the seed node, then \( n \) should be divided into children with indices of 0, 3, 4 and 7, which are on the left half of node \( n \). Figure 3.16 below depicts how this was done in a quadtree scenario. Figure 3.16a shows the initial seed node, ‘S’. In Figure 3.16b, neighbours ‘N1’ to ‘N4’, of the same depth are found. If the node is not leaf node which has children, this node should be separated into its children nodes which belong to one side according to the relationship with the seed node recursively (Figure 3.16c). For example, ‘N3’ node in Figure 3.16b should be separated into node ‘N3’ and ‘N4’ in Figure 3.16d because ‘N3’ is the down neighbour node of the seed node ‘S’. Figure 3.16d shows all the connected neighbours that were found.
The next task involves generating a boundary surface which splits stone and air based on the constructed octree structure. The boundary nodes were already obtained during the flood fill algorithm on the octree. For instance, if we searched in the up direction when doing flood fill and found that the next node’s material was not air, then the surface between this node and the next node, or more precisely, the node’s top surface or the next node’s bottom face forms part of the boundary surface.

It is easy to find the boundary nodes as they always occurs in pairs, with the material of one node being air, and the material of the other will be stone. Therefore, it seems that we can choose either one’s face as the boundary surface. However, if the depth of this node and the next node is not the same, the smaller node has to be selected to ensure that the boundary surface is correct without being overlapped or
blocked. In order to store boundary information, two new properties had to be added to the octree node structure, as shown in Table 3.3.

<table>
<thead>
<tr>
<th>Name</th>
<th>Data type</th>
<th>Meaning</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>insideBoundary</td>
<td>BOOLEAN</td>
<td>All the surfaces are inside of this node</td>
<td>False</td>
</tr>
<tr>
<td>boundaryDir</td>
<td>INTEGER</td>
<td>Which face(s) is boundary</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.3 Boundary properties of octree node

For boundaryDir property, an enumeration type was needed to indicate the direction of the face (see table 3.4 below). So if one node’s boundaryDir is 7 (i.e. 1 + 2 + 4), this means that the up, down, and left faces are part of final boundary surface.

<table>
<thead>
<tr>
<th>Boundary direction</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOBOUNDARY</td>
<td>0</td>
</tr>
<tr>
<td>UP</td>
<td>1</td>
</tr>
<tr>
<td>DOWN</td>
<td>2</td>
</tr>
<tr>
<td>LEFT</td>
<td>4</td>
</tr>
<tr>
<td>RIGHT</td>
<td>8</td>
</tr>
<tr>
<td>FRONT</td>
<td>16</td>
</tr>
<tr>
<td>BACK</td>
<td>32</td>
</tr>
</tbody>
</table>

Table 3.4 Enumeration values for the boundary properties

Figure 3.17 shows the entire flood fill process for a 2D quadtree case. Figure 3.17a illustrates the quadtree after the bottom-up building process, and Figure 3.17b shows the status after the flood fill algorithm, and indicates all the boundary nodes in air. From this, the cave boundary surfaces can be determined by selecting the smaller of adjacent stone-air nodes (with larger depth), as depicted in Figure 3.17c. Then the polygonal surfaces of the cave wall can be computed, as shown in Figure 3.17d.
The process of flood fill and fetching boundary surface of cave model are done at the same time, and the pseudo code for this is shown in Figure 3.18.

```c
//This method is used to do flood fill algorithm on octree structure
//for given seed point P, and fetch boundary surface at the same
//time, the boundary surface information is stored separated in
//boundary nodes, for cave model, the target material is air
FLOOD_FILL(POINT P, MATERIAL TARGET, MATERIAL REPLACE)
    SET FIND_CONTIANS_NODE(P) to OCTREE_ORIG
    IF OCTREE_ORIG equals NULL OR OCTREE_ORIG->material is not equal
to TARGET THEN
        RETURN
    ENDIF
    Create empty QUEUE Q
    INSERT OCTREE_ORIG into Q
    WHILE Q is not empty
        POP Q into OCTREE_ORIG
        IF OCTREE.ORIG->material equals TARGET
```
THEN SET REPLACE to OCTREE_ORIG->material

//Up direction
SET (0, -OCTREE_ORIG->length, 0) to OFFSET
SET FIND_CONTIANS_NODEWITHDEPTH(OCTREE_ORIG->centre,
    OCTREE_ORIG->centre +
    OFFSET, OCTREE_ORIG->depth)
to OCTREE_NEXT
IF OCTREE_NEXT is not NULL
THEN
    Create empty QUEUE TEMPQ
    INSERT OCTREE_NEXT into TEMPQ
    WHILE TEMPQ is not empty
    POP TEMPQ into TEMPQ
    IF OCTREE_NEXT->material equals junction
    THEN
        INSERT child 4, 5, 6, 7 of OCTREE_NEXT into TEMPQ
        CONTINUE
    ENDIF
    //replace material
    IF OCTREE_NEXT->material equals TARGET
    THEN
        SET REPLACE to OCTREE_NEXT->material
        INSERT OCTREE_NEXT into Q
        CONTINUE
    ENDIF
    //assign boundary information
    IF OCTREE_NEXT->material equals stone AND
        OCTREE_ORIG->material equals REPLACE
    THEN
        //choose smaller nodes as the boundary nodes
        IF OCTREE_NEXT->depth < OCTREE_ORIG->depth
        THEN
            Add UP boundary information into
            OCTREE_ORIG node
            SET FLASE to OCTREE_ORIG->insideBoundary
        ELSE
            Add DOWN boundary information into
            OCTREE_NEXT node
            SET FLASE to OCTREE_NEXT->insideBoundary
        ENDIF
    ENDIF
ENDIF
ENDWHILE
ENDIF
ENDWHILE
//The other search direction
ENDWHILE

Figure 3.18 Pseudo code for the FLOOD_FILL algorithm

Since every vertex’s relative position in the octree node is fixed, a lookup table can be used for the boundary surface mesh triangles. Table 3.5 is made up of child indices, and separated to inside and outside, which means the boundary surface is inside or outside of the specific node.
<table>
<thead>
<tr>
<th>Face</th>
<th>Triangle indices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Up face outside indices</td>
<td>0, 1, 2, 2, 3, 0</td>
</tr>
<tr>
<td>Down face outside indices</td>
<td>4, 7, 6, 6, 5, 4</td>
</tr>
<tr>
<td>Left face outside indices</td>
<td>4, 0, 3, 3, 7, 4</td>
</tr>
<tr>
<td>Right face outside indices</td>
<td>2, 1, 5, 5, 6, 2</td>
</tr>
<tr>
<td>Front face outside indices</td>
<td>6, 7, 3, 6, 3, 2</td>
</tr>
<tr>
<td>Back face outside indices</td>
<td>0, 4, 5, 5, 1, 0</td>
</tr>
<tr>
<td>Up face inside indices</td>
<td>1, 0, 2, 2, 0, 3</td>
</tr>
<tr>
<td>Down face inside indices</td>
<td>6, 4, 5, 7, 4, 6</td>
</tr>
<tr>
<td>Left face inside indices</td>
<td>4, 3, 0, 7, 3, 4</td>
</tr>
<tr>
<td>Right face inside indices</td>
<td>1, 2, 5, 5, 2, 6</td>
</tr>
<tr>
<td>Front face inside indices</td>
<td>3, 7, 6, 3, 6, 2</td>
</tr>
<tr>
<td>Back face inside indices</td>
<td>4, 0, 5, 5, 0, 1</td>
</tr>
</tbody>
</table>

Table 3.5 Face triangles indices of the octree nodes

At this stage it can be seen that boundary surface can be generated using this approach without any holes. The boundary surface is stored separately in each node whose boundaryDir is not zero. According to mesh triangle indices lookup table, boundary surfaces can be transferred into a triangle mesh. Figure 3.19 below shows a 2D case scenario.
3.4 Experiment Results

All tests were done on a Core 2 duo E8600 at 3.33 GHz, and an ATI Radeon HD 3450 graphics card. The program is written in C# and using XNA 3.1 game studio, and the Microsoft Visual C# 2008 IDE.

3.4.1 Model complexity

Table 3.6 and Figure 3.20 show a comparison of the cave model’s octree complexity for octrees constructed with different depths. The final result is the rough inside surfaces of a 3D cave model based on a sphere, which was constructed using the proposed approach. The turbulence was done using the Perlin noise function and a bias function with the bias value set to 0.05. The inside and outside nodes are the boundary leaf nodes of the final octree.
Figure 3.20 Model complexity comparison of octree structure

<table>
<thead>
<tr>
<th>Octree Node Count</th>
<th>Octree Maximum Depth</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inside Boundary Node Count</td>
<td></td>
<td>172</td>
<td>606</td>
<td>2284</td>
<td>9501</td>
<td>43766</td>
</tr>
<tr>
<td>Outside Boundary Node Count</td>
<td></td>
<td>250</td>
<td>1012</td>
<td>4235</td>
<td>17858</td>
<td>73760</td>
</tr>
</tbody>
</table>

Table 3.6 Model complexity comparison of octree structure

Since the final model is a mesh made up of triangles, analysis was also performed to obtain the vertex count and triangle count of the final model. These are shown in Figure 3.21, with the details in Table 3.7. From the figure, one can easily see that the final model’s vertex and triangle count increases exponentially with the maximum depth of the octree.
Figure 3.21 Model complexity comparison of triangle mesh

<table>
<thead>
<tr>
<th>Octree Maximum Depth</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mesh Vertex/Triangle Count</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mesh Vertex Count</td>
<td>654</td>
<td>2093</td>
<td>11135</td>
<td>47756</td>
<td>207836</td>
</tr>
<tr>
<td>Mesh Triangle Count</td>
<td>1296</td>
<td>5196</td>
<td>21376</td>
<td>90760</td>
<td>388084</td>
</tr>
</tbody>
</table>

Table 3.7 Model complexity comparison of triangle mesh

3.4.2 Material sampling count in space comparison

A comparison between three different approaches of procedurally generating the octree structure was conducted. The three approaches generate the exact same 3D cave model. The first two approaches are the standard, well recognised, octree construction approaches, namely, the “top-down” and “bottom-up” approaches, whereas the third approach is the method developed in this research.

The first approach is to build the octree structure using a top-down sequence [Samet 1990]. In this approach, all the points in a node are sampled, and the node is separated into eight child node if there is more than one type of material. Then the
flood fill algorithm is performed on the octree using a given stone seed (replace all connected nodes to stone). Finally, the flood fill algorithm was done on the octree using a given air seed (replace all nodes which are not stone with air) and to get the boundary surface at the same time.

The second approach is to construct the octree using a bottom-up algorithm [Samet 1990]. First, assign materials to the nodes by visiting each node in Morton order [Morton 1966], before performing the flood fill algorithm (the flood fill approach is the same as in the first approach).

However, these two classic approaches have their own disadvantages due to the specific requirements of our approach. Therefore, third approach is the approach developed in this research, which is to construct the octree using the insert node algorithm which was previously discussed by giving a stone seed, then replace all null nodes’ material in the octree with air. Then the flood fill algorithm was performed on the octree by giving an air seed and to get the boundary surfaces at the same time.

Two aspects were chosen for comparison. The first property was to count the number of times that the material for that space was checked (given a point in space, return whether it is part of stone or not) in the whole generating process. The second property was to count the number of times the material of the octree was checked (given a tree index, return whether it is part of stone or not).

Table 3.8 and Figure 3.22 below shows the sampling count in the different approaches described before. One sampling means given a point’s coordinates in space, return whether it belongs to stone or air according to the distribution function. It can be seen from Table 3.8 and Figure 3.22 that the checking times for approach 3 have been greatly reduced, and the computation is reduced even more when the depth is larger.
Figure 3.22 Comparison of sampling times in space

<table>
<thead>
<tr>
<th>Octree Maximum Depth</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sampling Times Count</td>
<td>Approach 1</td>
<td>Approach 2</td>
<td>Approach 3</td>
<td>Approach 1</td>
<td>Approach 2</td>
</tr>
<tr>
<td>6211</td>
<td>106212</td>
<td>414810</td>
<td>3329425</td>
<td>26616879</td>
<td>4096</td>
</tr>
<tr>
<td>Approach 3</td>
<td>3992</td>
<td>29642</td>
<td>225313</td>
<td>1799992</td>
<td>14187327</td>
</tr>
</tbody>
</table>

Table 3.8 Comparison of sampling times in space

For approach 1, there are many duplicate samplings since the building sequence is top-down, and the sampling count is steady for approach 2 (just related to the maximum depth of octree) since it just samples all the points in Morton order. The approach 3 has the smallest sampling count because it only checks points when it is necessary when doing flood fill algorithm. However, there are still many duplicate calculations on the same point in space during approach 3 even when the scan line flood fill algorithm was used. The performance should be even better if the flood fill algorithm can be improved.
3.4.3 Material sampling count in octree comparison

Table 3.9 Comparison of sampling times in octree structure. One sampling time means that given a point’s coordinate in space, find the related octree leaf node, and check the material of this node to see whether this point is a stone or air node.

<table>
<thead>
<tr>
<th>Octree Maximum Depth</th>
<th>Sampling Times Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>7706</td>
</tr>
<tr>
<td>6</td>
<td>39154</td>
</tr>
<tr>
<td>7</td>
<td>236444</td>
</tr>
<tr>
<td>8</td>
<td>1588910</td>
</tr>
<tr>
<td>9</td>
<td>10026594</td>
</tr>
</tbody>
</table>

![Comparison of sampling times in octree structure](image)

Figure 3.23 Comparison of sampling times in octree structure

<table>
<thead>
<tr>
<th>Octree Maximum Depth</th>
<th>Sampling Times Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Approach 1</td>
<td>7706</td>
</tr>
<tr>
<td>Approach 2</td>
<td>39154</td>
</tr>
<tr>
<td>Approach 3</td>
<td>236444</td>
</tr>
<tr>
<td>Approach 3</td>
<td>1588910</td>
</tr>
<tr>
<td>Approach 3</td>
<td>10026594</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Octree Maximum Depth</th>
<th>Sampling Times Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Approach 1</td>
<td>3357</td>
</tr>
<tr>
<td>Approach 2</td>
<td>17109</td>
</tr>
<tr>
<td>Approach 3</td>
<td>90318</td>
</tr>
<tr>
<td>Approach 3</td>
<td>678910</td>
</tr>
<tr>
<td>Approach 3</td>
<td>4860953</td>
</tr>
</tbody>
</table>

Table 3.9 Comparison of sampling times in octree structure

The table indicates the calculation count of checked materials in octree comparison for three different approaches. From the table, it can be seen that the count of approach 1 and approach 2 are the same, and the calculation has been reduced by more than half in approach 3. This is because the first flood fill process has been merged into the construction of octree process.
3.4.4 Intersecting surface comparison

Figure 3.24 shows the 2D intersecting surface of sphere based cave models with different bias values and Perlin noise frequencies, where black stands for stone and white stands for air. Note and Perlin noise used here is a kind of $1/f$ noise, which means the result of noise value is summaries with several Perlin noise value with double frequencies and half amplitudes until reach the resolution.

Figure 3.25 shows the intersecting surface of cubic based cave models with different bias values and Perlin noise frequencies.

<table>
<thead>
<tr>
<th>Frequency</th>
<th>Bias Value</th>
<th>Perlin noise</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>0.05</td>
<td>0.005</td>
</tr>
<tr>
<td>256</td>
<td><img src="image1" alt="Images" /></td>
<td><img src="image2" alt="Images" /></td>
</tr>
<tr>
<td>64</td>
<td><img src="image4" alt="Images" /></td>
<td><img src="image5" alt="Images" /></td>
</tr>
<tr>
<td>16</td>
<td><img src="image7" alt="Images" /></td>
<td><img src="image8" alt="Images" /></td>
</tr>
<tr>
<td>4</td>
<td><img src="image10" alt="Images" /></td>
<td><img src="image11" alt="Images" /></td>
</tr>
<tr>
<td>1</td>
<td><img src="image13" alt="Images" /></td>
<td><img src="image14" alt="Images" /></td>
</tr>
</tbody>
</table>

Figure 3.24 Intersection surface of Perlin noise sphere based
<table>
<thead>
<tr>
<th>Frequency</th>
<th>Bias Value</th>
<th>Perlin noise</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.5</td>
<td>0.05</td>
</tr>
<tr>
<td>256</td>
<td>![Image]</td>
<td>![Image]</td>
</tr>
<tr>
<td>64</td>
<td>![Image]</td>
<td>![Image]</td>
</tr>
<tr>
<td>16</td>
<td>![Image]</td>
<td>![Image]</td>
</tr>
<tr>
<td>4</td>
<td>![Image]</td>
<td>![Image]</td>
</tr>
<tr>
<td>1</td>
<td>![Image]</td>
<td>![Image]</td>
</tr>
</tbody>
</table>

Figure 3.25 Intersection surface of Perlin noise cubic based

From Figure 3.24 and Figure 3.25, the difference can easily be seen. The bias value controls the final shape of the cave model. In particular, larger bias values give rise to cave models with rough walls, and makes the final cave smaller, while smaller bias values result in cave models that closely follow the cave foundation shape.

The different Perlin noise sampling frequencies control the roughness of the final shape. In particular, larger sampling frequencies lead to smoother surfaces, while smaller sampling frequencies lead to rougher surfaces.

Tests were also performed to construct the distribution function of cave using two other types of procedural noise (i.e. wavelet noise and simplex noise were used). Figure 3.26 and Figure 3.27 show the intersecting surface of sphere and cubic based cave models with different bias values and wavelet noise frequencies. Note that the noise is not 1/f noise, and just use wavelet noise values, which is not summarised by several wavelet noise values with doubled frequency and halved amplitude.
Figure 3.28 and Figure 3.29 show the intersecting surface of sphere and cubic based cave models with different bias values and simplex noise frequencies. The result simplex noise value is $1/f$ noise, which means summarised by several wavelet noise values with doubled frequency and halved amplitude.

<table>
<thead>
<tr>
<th>Frequency</th>
<th>Bias Value</th>
<th>Wavelet Noise</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>0.05</td>
<td></td>
</tr>
<tr>
<td>0.005</td>
<td>0.0005</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.26 Intersection surface of wavelet noise sphere based
<table>
<thead>
<tr>
<th>Frequency</th>
<th>Bias Value</th>
<th>Wavelet Noise</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>0.5</td>
<td>![Image]</td>
</tr>
<tr>
<td></td>
<td>0.05</td>
<td>![Image]</td>
</tr>
<tr>
<td></td>
<td>0.005</td>
<td>![Image]</td>
</tr>
<tr>
<td></td>
<td>0.0005</td>
<td>![Image]</td>
</tr>
<tr>
<td>64</td>
<td>0.5</td>
<td>![Image]</td>
</tr>
<tr>
<td></td>
<td>0.05</td>
<td>![Image]</td>
</tr>
<tr>
<td></td>
<td>0.005</td>
<td>![Image]</td>
</tr>
<tr>
<td></td>
<td>0.0005</td>
<td>![Image]</td>
</tr>
<tr>
<td>32</td>
<td>0.5</td>
<td>![Image]</td>
</tr>
<tr>
<td></td>
<td>0.05</td>
<td>![Image]</td>
</tr>
<tr>
<td></td>
<td>0.005</td>
<td>![Image]</td>
</tr>
<tr>
<td></td>
<td>0.0005</td>
<td>![Image]</td>
</tr>
<tr>
<td>16</td>
<td>0.5</td>
<td>![Image]</td>
</tr>
<tr>
<td></td>
<td>0.05</td>
<td>![Image]</td>
</tr>
<tr>
<td></td>
<td>0.005</td>
<td>![Image]</td>
</tr>
<tr>
<td></td>
<td>0.0005</td>
<td>![Image]</td>
</tr>
</tbody>
</table>

Figure 3.27 Intersection surface of wavelet noise cubic based

<table>
<thead>
<tr>
<th>Frequency</th>
<th>Bias Value</th>
<th>Simplex Noise</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.5</td>
<td>![Image]</td>
</tr>
<tr>
<td></td>
<td>0.05</td>
<td>![Image]</td>
</tr>
<tr>
<td></td>
<td>0.005</td>
<td>![Image]</td>
</tr>
<tr>
<td></td>
<td>0.0005</td>
<td>![Image]</td>
</tr>
<tr>
<td>4</td>
<td>0.5</td>
<td>![Image]</td>
</tr>
<tr>
<td></td>
<td>0.05</td>
<td>![Image]</td>
</tr>
<tr>
<td></td>
<td>0.005</td>
<td>![Image]</td>
</tr>
<tr>
<td></td>
<td>0.0005</td>
<td>![Image]</td>
</tr>
<tr>
<td>16</td>
<td>0.5</td>
<td>![Image]</td>
</tr>
<tr>
<td></td>
<td>0.05</td>
<td>![Image]</td>
</tr>
<tr>
<td></td>
<td>0.005</td>
<td>![Image]</td>
</tr>
<tr>
<td></td>
<td>0.0005</td>
<td>![Image]</td>
</tr>
<tr>
<td>64</td>
<td>0.5</td>
<td>![Image]</td>
</tr>
<tr>
<td></td>
<td>0.05</td>
<td>![Image]</td>
</tr>
<tr>
<td></td>
<td>0.005</td>
<td>![Image]</td>
</tr>
<tr>
<td></td>
<td>0.0005</td>
<td>![Image]</td>
</tr>
</tbody>
</table>

Figure 3.28 Intersection surface of simplex noise sphere based
<table>
<thead>
<tr>
<th>Frequency</th>
<th>Bias Value</th>
<th>Simplex Noise</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.5</td>
<td>0.05</td>
</tr>
<tr>
<td>1</td>
<td>![Image]</td>
<td>![Image]</td>
</tr>
<tr>
<td>4</td>
<td>![Image]</td>
<td>![Image]</td>
</tr>
<tr>
<td>16</td>
<td>![Image]</td>
<td>![Image]</td>
</tr>
<tr>
<td>64</td>
<td>![Image]</td>
<td>![Image]</td>
</tr>
</tbody>
</table>

**Figure 3.29 Intersection surface of simplex noise cubic based**

From the figures (Figure 3.26, Figure 3.27, Figure 3.28 and Figure 3.29), we can see that the final cave models based on different kinds of procedural noises (Perlin noise, wavelet noise and simplex noise) have similar results with regard to the effect of the bias value and the frequency of noise. However, wavelet noise and simplex noise also have their own advantages. For wavelet noise, the result would be much steadier since wavelet noise is a kind of perfect band-limited noise, and for simplex noise, it can reach the similar results to Perlin noise with less computational task.

The figures below show the different intersecting surface of Perlin noise with a ‘fractal’ power spectrum based on a Perlin noise frequency of 64. The pseudo code is shown in Figure 3.30, and Figure 3.31 shows the different noise and final cave model with different $N$ values.
//Assume the frequency is FREQUENCY and the amplitude is AMPLITUDE
//at beginning
FRACTAL_NOISE(POINT P, INTEGER N)
  SET 0 to VALUE
  SET FREQUENCY to F
  SET AMPLITUDE to A
  FOR i = 0 to N-1 STEP 1
    VALUE += PERLINNOISE(P * F)
    SET F * 2 to F
    SET A / 2 to A
  ENDFOR
  RETURN VALUE

Figure 3.30 Pseudo code of FRACTAL_NOISE function

(a) N = 1
(b) N = 4
(c) N = 2
(d) N = 5
(e) N = 3
(f) N = 6

Figure 3.31 Fractal comparison with different parameter N

From Figure 3.31, all the final cave models are similar with different levels of detail, and the effect of parameter N is also obvious: For larger values, the result contains more detailed information, and the resulting surface is rougher, while smaller N values lead to smoother surfaces with less detail. So different N values can be used to meet different requirements.
Figure 3.32 below are screen shots for intersection surface of 3D case and wire frame of it with octree maximum depth 8 and quadtree maximum depth 10.
As Figure 3.32 shows, the boundary surface and wire frame are more complex and closer to the basic shapes, and the model complexity rapidly increases with the octree maximum depth’s growth, which had been discussed in Figure 3.21.

### 3.5 Comparison with Related Work

This chapter presents an approach to generating a triangle-based mesh representing the boundary surfaces of a cave foundation. The motivation for developing this proposed approach lies in the fact that most current state-of-the-art methods of terrain generation only deal with generating the surface of a terrain. Hence, if merely generating the surface of a terrain, standard approaches like the heightmap method [Musgrave 1993] or the layered data representation [Benes and Forsbach 2001] method can be adopted. In addition, although much research on procedural terrain generation in areas like simulating terrain erosion and a variety of approaches have been developed to achieve physically based erosion techniques [Olsen 2004; Zhou et al. 2007; Smelik et al. 2009], they are all focused on generating the surfaces of terrains. As such, none of these approaches can handle the case of overhangs, like caves or arches. Therefore, a suitable approach has to be developed in order to cater for the specific needs of procedurally generating 3D cave structures, which is the focus of this research.

In comparison with existing approaches of obtaining cave structures, am Ende [2001] used a physical scanning approach that requires the use of 3D scanning hardware to scan and record accurate spatial data about actual real-life caves. This is an extremely tedious process that involves countless hours and a lot of manual labour.
The approach presented in this chapter avoids this manual labour by getting a computer to auto-generate the cave structure. However, since random noise functions are used in the proposed approach, if compared to the work by am Ende [2001], the procedurally generated cave structure is not accurately based on any real-life caves. Thus, the approach proposed in this thesis cannot be used for modelling any caves that exist in the real world.

Another current approach to generating caves was proposed by Johnson et al. [2010]. However, their approach was focused on generating infinite 2D cave maps using a cellular automata based algorithm. In comparison with their approach, the purpose of the approach presented in this chapter is to generate 3D cave structures as opposed to 2D cave maps. For the experimental results shown in the previous section, it can be seen that 3D noise functions can successfully be used to generate 3D cave foundations, unlike the case where complicated cellular automata based algorithms have to be used as proposed by Johnson et al. [2010]. Moreover, they merely show that their method produces 2D images of cave passages, rather than full 3D models of caves as can be obtained from results of the proposed approach.

Unlike the simple generation of 2D cave maps, Boggus and Crawfis [2009a, 2009b and 2010] recently proposed the procedural creation of synthetic 3D cave models. To represent the cave foundation in their approach, they use a pair of heightmaps; one to represent the cave floor, and the other to represent the cave ceiling. While this is a fast method that can be used to represent the cave structure, it has severe limitations as it cannot be used to represent any form of arches or irregular rock formations in the resulting cave structure. In some of their work, the cave ceiling is a perfect mirror image of the cave floor which is unrealistic. In addition, they model their caves around Bezier or B-Spline curves which makes the resulting cave surface appear unrealistically smooth. To alleviate this, they suggest using computer graphics techniques like bump mapping and displacement mapping to add detail to the smooth surfaces.

The experimental results of the approach proposed in this research show that it can overcome these problems. In particular, because random noise functions are used instead of smooth curves, the walls of the 3D cave structure are not unrealistically smooth surfaces that have to be alleviated using computer graphics techniques. Instead, rough surfaces which are characteristic of internal cave surfaces are modelled directly into the resulting 3D cave model. Furthermore, by using a voxel-
based octree approach instead of a pair of heightmaps, the 3D caves resulting from the proposed approach can handle irregular formations like arches in the cave, which cannot be represented using a heightmap approach. In addition, the experimental results have shown that it is extremely convenient to procedurally generate cave structures with different characteristics by simply changing the noise function parameters.

3.6 Summary

In this chapter, a novel procedural-based approach to generating a triangle-based cave foundation using noise functions was introduced. The noise function was used to separate between air and stone sections of the 3D cave model. In addition, experimental results have shown that cave structures with different characteristics can be generated by simply changing the noise function and bias value parameters. The advantage of this is that different procedurally generated caves can easily be created by simply adjusting the noise function parameters.

Compared with previous approaches in procedural terrain generation, the approach presented in this chapter can overcome the limitation of previous terrain generation techniques which are only able to produce terrain surfaces. Furthermore, unlike previous methods of 3D cave modelling which involves the laborious task of scanning physical caves in order to obtain accurate 3D information required in the modelling of the cave, the present approach can obtain this information using a computer algorithm.

Also, experimental results have shown that the proposed approach overcomes limitations observed in current procedural cave generation methods, which produce cave structures that are unrealistically smooth and cannot represent irregular cave structures like arches in the cave formation. The novel approach presented in this chapter allows for the modelling of rough surfaces which are characteristic of the surfaces found in real caves, without having to rely on additional computer graphics techniques to simulate these effects.
CHAPTER 4: PROCESSING THE BOUNDARY SURFACE

The boundary surface generated using the process described in chapter 3 is not realistic since it consists of cube surfaces. This chapter first describes a smoothing function to change vertex positions according to all adjacent vertices based on the Laplacian smoothing function, and then discusses how to calculate the normal vectors for each vertex, which can also be used to increase the realistic effects on the final cave model. However, cracks may occur due to the change in the positions of the vertices. Therefore, this chapter also analyses the reason of cracks, and introduces an algorithm for patching all potential cracks.

4.1 Surface Smoothing

Although the polygonal cave wall surface has been generated, it is directly made up of small cube surfaces, which makes the boundary surfaces of cave like walls when the cubes are not small enough, as can be seen in Figure 3.32. So it is necessary to convert the boundary surface polygons obtained from data stored in a voxel-based octree, into a smoother looking polygonal mesh. This resulting mesh can then be rendered in real-time using standard triangle rasterisation techniques. A method of smoothing the resulting polygonal mesh is introduced based on a standard Laplacian smoothing function (Equation 4.1) [Vollmer et al. 1999], which can improve realistic effects without rebuild the octree structure with a larger maximum depth.

\[
v' = \lambda v + \frac{1 - \lambda}{n} \sum_{i=1}^{n} v_i, \quad (0 \leq \lambda \leq 1)
\]  

Equation 4.1 shows the formula to change vertex coordinates from their original positions towards to all adjacent vertexes. In this formula, \(v\) represents the original vertex location, and \(v_i\) stands for all the neighboring vertexes surrounding vertex \(v\). Thus, \(v'\) represents the adjusted vertex position. Parameter \(\lambda\) is the weighting factor and typically is assigned to 0.5, which stands for the contribution proportion to the adjusted vertex of original vertex, and the other contribution of adjusted vertex is averagely distributed to all adjacent vertexes. The implementation of this function effectively changes the original location of the triangle vertices and moves them closer toward their adjacent vertices, and hence removing sharp corners and creating a smoother polygonal mesh. Figure 4.1 shows the result after smoothing operation, and the black cracks can be found on the boundary surface.
The whole process of smoothing the boundary surfaces can be done in two steps. First, we need to find all adjacent relationship according to the triangles mesh, then for each vertex, apply the formula in Equation 4.1 to change its position. The pseudo code of this is shown in Figure 4.2 below.

```plaintext
// Begin work
// Assuming all the vertexes of final mesh are stored in array
// VertexList, and there are no duplicate vertexes in this array.
// Assuming all the triangles' vertexes' indexes are stored in array
// IndexList. For example, if IndexList is like (0,1,2,2,1,0,...) that
// means the first triangle is consisted of vertexes in VertexList
// with index 0, 1 and 2.

// This approach is used to find all adjacent vertexes for each
// vertex on the mesh
// Define a data structure to store the adjacent relationship, the
// first part is the original vertex index, and the second part is an
// adjacent list to store all adjacent vertexes' indexes to original
// vertex.
FOR each triangle in IndexList
  SET the first vertex index to A
  SET the second vertex index to B
  SET the third vertex index to C
  Check whether A is existed in the adjacent list of B, if not
```
exist, insert A into adjacent list of B.

Check whether A is existed in the adjacent list of C, if not exist, insert A into adjacent list of C.
Check whether B is existed in the adjacent list of A, if not exist, insert B into adjacent list of A.
Check whether B is existed in the adjacent list of C, if not exist, insert B into adjacent list of C.
Check whether C is existed in the adjacent list of A, if not exist, insert C into adjacent list of A.
Check whether C is existed in the adjacent list of B, if not exist, insert C into adjacent list of B.

END FOR

//This approach is used to applying smoothing function on each vertex
//Assume $\lambda$ stands for the percentage of original vertex to the adjusted vertex.
//Create a new vertex list (labelled with VertexList).
FOR each vertex v in VertexList
    SET adjustVertex to $\lambda \times v$
    SET n to the count of adjacent list of v
    FOR each vertex $v_i$ in adjacent list of v
        ADD $((1 - \lambda) \times v_i) / n$ to adjustVertex itself
    END FOR
    ADD adjustVertex to the new vertex list
END FOR
Replace the vertex list with the new one

Figure 4.2 Pseudo code of smoothing function

4.2 Patching Cracks

It can be seen from Figure 4.1 that the smoothing operation results in many cracks appearing on the boundary surface. The reason for the cracks is shown in the Figure 4.3 below.
Based on Figure 4.3a, at beginning there are 3 triangles $ABC$, $AED$ and $ECD$, and point $E$ is on the segment $AC$. However, after changing vertex positions (Figure 4.3b), there will be a crack along $ACE$, because point $E'$ is no longer on the segment $A'C'$. This will also happen on the boundary surface after smoothing, and the essential reason for the cracks is because the boundary surfaces of octree nodes next to each other may have different depths, and what is worse, in some special case, the displacement would happen more than one times on one segment, which means that the shapes of cracks can be other polygon shapes rather than triangles, such as rectangle and pentagon, etc. There are many different solutions to fix this crack issue, such as adding a new triangle $ACE$ into the triangle list. However, the shape of the cracks are hard to predict, which brings complexity to keep consist, seamless and coplanar property.

The solution developed by this research is to patch these cracks by separating the original triangles, depending on the intersection points of the side, into several new triangles instead. Therefore, it is easier than to generate new triangles just fit the cracks, and it is also convenient to keep a right-handed polygon winding sequence. For the example above in Figure 4.3a, triangles $ABE$ and $BCE$ can be used instead of triangle $ABC$.

In order to do this process, the side of triangles which could cause potential cracks have to be found. As mentioned before, the intersection points are the only reason, and all intersection groups based on all the vertexes and connections of the
final mesh model must be found. However, this would be a time-consuming if all the vertices have to be directly searched. Instead, after some research, it was found that in the octree structure, if there is any intersection point in segment $AB$, the middle position (labelled $C$) of $AB$ should be the first one of the intersection points, and this feature is asymptotical correct, so depending on this, all the intersections of the given points can be efficiently found.

1. For each segment named $AB$ in the final triangle mesh, define a new array named $ARRAY$, and insert point $A$ and point $B$ into this array

2. For each pair with indexes $i$ and $i + 1$ in the array, calculate the middle point $C$ of this pair, and check whether this middle point $C$ is one of the vertexes of final triangle mesh. If yes, add $C$ into this array at position $i + 1$. Then repeat step 2 until all the middle points are not on the list of final triangle mesh.

After this approach, we should get an array such as $ACDEB$, and point $C$, $D$ and $E$ are intersect points of side $AB$. The pseudo code of this progress is shown below in Figure 4.4.

```plaintext
//This process is used to find all intersection points of mesh, and
//this step should be done before smoothing.
FOR each edges in IndexList
  SET vertex vecBegin to the begin vertex of edge by index
  SET vertex vecEnd to the end vertex of edge by index
  Create an index list IntersectList to store begin, end vertex
  index and intersection points index (if any)
  Insert vecBegin’s index, vecEnd’s index into IntersectList
  SET TRUE to CONTINUEFLAG
  WHILE CONTINUEFLAG equals TRUE
    SET FLASE to CONTINUEFLAG
    SET loopCount to the size of IntersectList - 1
    FOR i = 0 to loopCount step 1
      SET vertex midVertex to the middle point of vertexes with
      indexes IntersectList[i], and IntersectList[i + 1]
      Check whether midVertex exists in vertexList
      IF exists,
      THEN
        Insert the index of midVertex into IntersectList at
        position i + 1
        INCREMENT i
        SET CONTINUEFLAG to true
      ENDIF
    END FOR
  ENDWHILE
END FOR
IF the count of IntersectList is greater than 2, save it into
structure IntersectArray
END FOR
```

Figure 4.4 Pseudo code of finding intersection points
Then next step is to split the original triangle. The figure below shows all the scenarios for splitting triangle $ABC$ (note the order of $ABC$ is fixed, that means, triangle $ABC$ and triangle $BCA$ stand for different triangles), the $BC$ side has the highest splitting priority, then the $AB$ side, and $CA$ side has the lowest split priority. Figure 4.5 shows all scenarios of splitting original triangles. In general, there are eight different scenarios of triangles in our case including the no-splitting scenario (shown as Figure 4.5a). Figure 4.5b, Figure 4.5c and Figure 4.5d show the cases where there is only one edge of triangles has intersection points; Figure 4.5e, Figure 4.5f and Figure 4.5g show the case where there are two edges of triangles with intersection points, and Figure 4.5h show all edges of triangles with intersection points.

Figure 4.5 The way of splitting a triangle based on different scenarios

Figure 4.6 shows the possible splitting cases for a 2D quadtree case. In Figure 4.6a, there are three triangles need to be split, labelled with 1, 2 and 3, and point $A$, $B$, $C$, $D$, $E$, $F$ and $G$, are intersection points. Figure 4.6b shows the result after separating the triangles.
FOR each triangle in IndexList
Check whether there is at least one edge has intersection points according to IntersectArray
IF there is no intersection points in either edge
THEN
CONTINUE
ENDIF
//remove the original triangle firstly
Remove the triangle vertex index from IndexList
SET the first vertex index to A
SET the second vertex index to B
SET the third vertex index to C
IF there is intersection points in edge BC
THEN
SET IntersectBC with element in IntersectArray, whose begin vertex’s index is B and end vertex’s index is C
ENDIF
IF there is intersection points in edge AB
THEN
SET IntersectAB with element in IntersectArray, whose begin vertex’s index is A and end vertex’s index is B
ENDIF
IF there is intersection points in edge CA
THEN
SET IntersectCA with element in IntersectArray, whose begin vertex’s index is C and end vertex’s index is A
ENDIF
//patch cracks by splitting triangles
IF there is intersection points in edge BC
THEN
//check whether the left part should be separated
IF there is intersection points in edge AB
THEN
FOR i = 0 to count of intersectionAB-1 step 1
    INSERT
ELSE
    //splitting occurs
ENDIF
ENDIF
IntersectionBC[1],
IntersectionAB[i] and
IntersectionAB[i + 1] into IndexList
END FOR
ELSE
INSERT
A,
B and 
IntersectionBC[1] into IndexList
ENDIF
//add the middle part
FOR i = 1 to count of IntersectionBC - 2 step 1
INSERT
A,
IntersectionBC[i] and 
IntersectionBC[i + 1] into IndexList
END FOR
IF there is intersection points in edge CA
THEN
FOR i = 0 to count of intersectionCA - 1 step 1
INSERT
penultimate element of IntersectionBC,
IntersectionCA[i] and 
IntersectionCA[i + 1] into IndexList
END FOR
ELSE
INSERT
A, 
penultimate element of IntersectionBC and 
C into IndexList
END IF
ELSE //there is no intersection point in edge BC
IF there is intersection point in edge AB
THEN
IF there is intersection point in edge CA
THEN
FOR i = 0 to count of intersectionCA - 1 step 1
INSERT
IntersectionAB[1], 
IntersectionCA[i] and 
IntersectionCA[i + 1] into IndexList
END FOR
ELSE
INSERT
A, 
IntersectionAB[1] and 
C into indexList,
ENDIF
FOR i = 1 to count of intersectionAB - 1 step 1
INSERT
C,
IntersectionAB[i] and 
IntersectionAB[i + 1] into IndexList
END FOR
ELSE
FOR i = 0 to count of intersectionCA - 1 step 1
INSERT
B, 
IntersectionCA[i] and 
IntersectionCA[i + 1] into IndexList
END FOR
Figure 4.7 Pseudo code of splitting triangles

4. 3 Calculating Mesh Normal Vectors

As discussed before, calculating normal vectors for each vertex in mesh is also an effective way to smooth the polygon surface based on the lighting computations. In order to achieve this, all the triangles should be fetched which share the specific vertex first, and then calculate the normal vectors for each triangle. Although using the average value as the normal vector of specific vertex is the faster way, the normal vector would lean in the direction which has more adjacent vertices. Therefore, the inverse value of related triangle area was used as the weight to the final normal vector. The formula is shown below (Equation 4.2).

\[
n' = \sum_{i=1}^{n} \frac{n_i}{area(\Delta t_i)}
\]  

(4.2)

4. 4 Experiment Results and Analysis

4.4.1 Model complexity comparison

Since the final model is a mesh consisting of triangles, the triangle count is used to show the complexity of the models. Table 4.1 and Figure 4.8 below show the results of the triangle splitting approach used to patch the cracks. The table depicts the total number of triangles of boundary surface before and after smoothing, for increasing levels of voxel resolution used in the generation of the 3D cave structure.
Figure 4.8 Before and after smooth model complexity comparison

<table>
<thead>
<tr>
<th>Octree Maximum Depth</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triangle Count</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Triangle Count Before Smooth</td>
<td>1296</td>
<td>5196</td>
<td>21376</td>
<td>90760</td>
<td>388084</td>
</tr>
<tr>
<td>Triangle Count After Smooth</td>
<td>1344</td>
<td>5388</td>
<td>22328</td>
<td>95888</td>
<td>418864</td>
</tr>
</tbody>
</table>

Table 4.1 Before and after smooth model complexity comparison

From Table 4.1 and Figure 4.8, results show that in general, the number of additional triangles generated to produce crack-free mesh form the smoothing process is less than 10%, and this increase is a small price to pay when it comes to generating a crack-free mesh.

Figure 4.9 shows a comparison of the interior cave walls before and after the smoothing process. The screen shots show the wireframe and solid polygonal mesh representations before and after smoothing.
Figure 4.9 Comparison of 3D surface before and after smoothing and patching cracks

Figure 4.10 shows screen shots with the same cave model and the same camera position with different parameter $\lambda$ values.
Figure 4.10 Comparison of smooth results with different $\lambda$ values
From Figure 4.10, we can see the comparison of wire frames and boundary surfaces with different $\lambda$ values. In general, small $\lambda$ values lead to smoother boundary surfaces and large $\lambda$ values would cause rough boundary surfaces. Specially, $\lambda$ equals 1 means the adjacent vertices contribute nothing to the final adjusted vertices, and $\lambda$ equals 0 means the adjusted vertices do not consider about original vertex’s position.

### 4.5 Comparison with Related Work

One of the most relevant and well established techniques of addressing the generation of seamless mesh based on volume data is marching cube algorithm [Lorensen and Cline 1987; Wilhelm and Gelder 1992; Shekhar et al. 1996]. However, the purpose of the marching cubes algorithm produces smooth iso-surfaces from 3D voxels. The purpose of the proposed approach is not to obtain smooth iso-surfaces, but rather to produce slightly rough surfaces which are characteristic of realistic surfaces found in real caves. Hence, the octree structure used in this research requires the use of a different algorithm.

Instead of adopting iso-surface generation algorithms, the approach employed uses a Laplacian smoothing function [Vollmer et al. 1999]. The result was a polygonal mesh that is smoother than having cube like shapes, yet at the same time not completely smooth iso-surfaces. This gives rise to the effect of having slightly rough cave walls which are modelled into the 3D geometry rather than having to resort to computer graphics techniques like bump mapping or displacement mapping as suggested by Boggus and Crawfis [2010].

The problem with using the Laplacian smoothing function by itself was that cracks in the form of missing triangles appeared at certain locations in the surface of the cave walls. A number of techniques have previously been proposed to fix cracks in meshes, for example work by Fakir and Greg [2003] and Wu et al. [2010]. This section discusses the limitations of their methods and describes why the approach presented in this chapter manages to overcome these limitations.

The approach of splitting triangles in order to patch the cracks as used in this research is relatively similar to that recently proposed by Wu et al. [2010]. However, the approach presented in Wu et al. [2010] was specifically for the case of quadtree based terrain. Thus, their approach only works for patching cracks for a 2D quadtree data structure. Compared to their approach, the method shown in this thesis works on
a 3D octree data structure and hence is much more flexible and extensible. Moreover, in Wu et al.’s [2010] approach, they only handle the case where the quadtree is reduced by one level of detail. The approach presented in this chapter can handle cases where adjacent nodes in the octree are split by more than once.

The work by Fakir and Greg [2003] showed a method of repairing the triangle meshes obtained from CAD models. The focus of their approach was to repair the meshes by patching the cracks so that there are no gaps or holes in the final representation of the 3D model. However, while their approach successfully manages to repair a model to produce a mesh without any holes, their approach involves first having to convert their 3D model into a voxelised representation before their approach can be applied. In addition, their approach involves ray stabbing to handle intersecting surfaces. Hence, this is substantially different from the proposed approach where information is stored in an octree data structure. The proposed approach allows for smoothing the sharp edges of the octree, whilst at the same time patching the cracks in a single integrated algorithm, instead of having to split it into different stages.

4.6 Summary
The chapter described how to do smoothing on the boundary surfaces generated using the approach from the previous chapter, to improve the realistic effects on the final result. First, a Laplacian based smoothing function was described to smooth the surface by changing vertex positions from original position towards their adjacent vertices. Cracks or holes might occur because the boundary nodes connected to each other may belong to different depth of octree structure, so an effective algorithm was developed to remove these cracks by separating the original triangles into several small triangles according to the different scenarios. For most of the cases, the increase in the degree of model complexity is less than 10% after smoothing operation.
CHAPTER 5: SIMULATION OF CAVE CHARACTERISTICS

Stalactites and stalagmites are two typical speleothems that occur in natural caves. This chapter proposes a procedural approach to simulate stalactites and stalagmites, which are controlled by parameters and merged into the cave model generation process described in chapters 3 and 4 of this thesis. First, a formula is introduced that indicates the relationship between the radius and the length of stalactites. Based on this formula, a group of stalactites model can be generated with different parameters. Then the relationship between stalactite and stalagmite is shown, and the stalagmites which are related to stalactites can be simulated in a similar way. Finally, this chapter describes how to merge the stalactites and stalagmites into the existing cave model.

5.1 Stalactite Simulation

Stalactites are the most typical phenomena in natural caves. The research on stalactite shapes is not as popular as the research of the chemical mechanisms underlying its growth. Short et al. [2005] introduced a formula of the length and the radius of the stalactite to accurately simulate its shape. Equation 5.1 shows the formula implemented in this research which is used to describe the shape of stalactites in this chapter. This equation is based on the equation proposed by Short et al. [2005].

\[ l(r) \equiv \text{scale} \left( \frac{3}{4} r^3 - \frac{2}{3} \ln r \right) \]  

(5.1)

In this formula, parameter \( r \) stands for the radius of specific stalactite, and the result is the length of stalactite. Parameter \( \text{scale} \) is used to control the final shape of stalactites; larger \( \text{scale} \) values can be used to generate slender stalactites and smaller \( \text{scale} \) can lead to stocky ones. Therefore, different stalactites with different shapes can be generated by assigning different parameters to them.

The next question is how to generate multi-stalactites at one time, since all the stalactites should occur in the same cave model, and they may intersect or overlapped with each other. The solution to this problem is to construct a virtual height map as the ceiling of the cave, and then to merge this into the distribution function of stone and air described in Section 3.1. The virtual height map can be built in several steps. First, a set of parameters needs to be decided before the approach, which includes the total number of stalactites, the radius of stalactites range, and the
related scales. Note that several different types of stalactites can be merged together with different radius ranges and scales to get better results. Then randomly locate the centres of stalactites on the height map, or locate the stalactites in certain range to meet specific requirements. The radii of related stalactites also need to be randomly assigned in a certain range. Next, go through all the stalactites, and add the shapes of them into the height map one by one. The pseudo code of this process is shown in Figure 5.1 below.

```
//This method is used to get the length of stalactite according to the radius of it.

GET_LENGTH(DOUBLE RADIUS)
    SET 0.75 * pow (RADIUS, 4/3) to TEMP1
    SET pow(RADIUS, 2/3) to TEMP2
    SET 1/3 * LOG(RADIUS) to TEMP3
    RETURN TEMP1
- TEMP2 - TEMP3

//This method is used to construct the virtual height map of stalactites, before this process, a set of parameter should be given, including the total number of stalactites, the radius range, the scale value and the map size

STRUCTURE STALACTITE()
    DOUBLE radius //stalactite radius
    POINT  centre //stalactite centre
    DOUBLE scale  //stalactite scale

BUILD_VIRTUAL_HEIGHT(INTEGER MAPSIZE, INTEGER TOTALNUMBER, INTEGER MINRANGE, INTEGER MAXRANGE, DOUBLE SCALE)
    DEFINE a new STALACTITE array STALACTITES with size TOTALNUMBER
    DEFINE a 2D array HEIGHTMAP with size MAPSIZE * MAPSIZE
    //initialize stalactite information
    FOR i = 0 to TOTALNUMBER -1 step 1
        SET a random value with range 0 to MAPSIZE to STALACTITES[i].centre.x
        SET a random value with range 0 to MAPSIZE to STALACTITES[i].centre.y
        SET a random value with range MINRANGE to MAXRANGE to STALACTITES[i].radius
    ENDFOR
    //add stalactites into virtual height map one by one
    FOR i = 0 to TOTALNUMBER -1
        SET LENGTH(STALACTITES[i].radius) to TOTALHEIGHT
        FOR each element P in array HEIGHTMAP
            SET STALACTITES[i].centre.x - P.x to TX
            SET STALACTITES[i].centre.y - P.y to TY
            SET the distance between P and STALACTITES[i].centre to DISTANCE
            IF DISTANCE is greater than STALACTITES[i].radius then
                CONTINUE
            ELSE
                IF DISTANCE equals to 0 then
```

Add TOTALHEIGHT to STALACTITES[P.x][P.y]
ELSE
SET GET_LENGTH(DISTANCE) to TEMPHEIGHT
Add TOTALHEIGHT - TEMPHEIGHT to STALACTITES[P.x][P.y]
ENDIF
ENDIF //DISTANCE is greater than STALACTITES[i].radius
END FOR
ENDFOR //FOR i = 0 to TOTALNUMBER -1

Figure 5.1 Pseudo code of simulation stalactite shape

Figure 5.2 shows the result after this process, and the intersected, overlapped stalactites can also be simulated.

Figure 5.2 Stalactite shape simulation

5.2 Stalagmite Simulation
For stalagmites, they always occur in pairs with stalactites, and usually grows faster than stalactites. By using a parameter ratio to control this ratio, ratio should be set to a value larger than 1. Based on this parameter, the virtual height map of stalactites can be reused to store stalagmites information. Figure 5.3 below is a screen shot of stalactites and stalagmites, where parameter ratio was set to 1.5.
5.3 Merging Stalactites and Stalagmites into the Cave Model

This section is on how to merge stalactites and stalagmites together into the cave model. In order to reduce the additional calculation task, the better way to do this is rewrite the distribution function of stone and air. So the new distribution function of stone and air should follow this sequence to check whether a specific point in space belongs to stone or air. If a point is judged to belong to air based on the original distribution method, it also needs to be checked additionally whether this point belongs to part of the stalactites or stalagmites, so that the stalactites and stalagmites can be added into the cave model. The process of checking whether a spatial point belongs to stalactites or stalagmites can be implemented by applying the upper ceiling and lower floor value and the virtual height map described in the previous section. The pseudo code of this process is shown in Figure 5.4 below.

```plaintext
// This method is used to check whether a spatial point belongs to
// stalactites or stalagmites according to the virtual height map,
// the upper ceiling and floor value

CHECK_STALACTITE_MATERIAL(POINT P,
    DOUBLE CEIL,
    DOUBLE FLOOR,
    DOUBLE RATIO,
    HEIGHTMAP)

SET HEIGHTMAP[P.x][P.z] to STALACTITEHEIGHT
SET RATIO * STALACTITEHEIGHT to STALAGMITEHEIGHT
// check whether this point belongs to stalactites
IF P.y > CEIL - STALACTITEHEIGHT
    THEN
```
RETURN STONE
ENDIF
//check whether this point belongs to stalagmites
IF P.y < FLOOR + STALAGMITEHEIGHT
    RETURN STONE
ENDIF
RETURN AIR

//New distribution method, based on original distribution function,
//and stalactite/stalagmite distribution
NEW_DISTRIBUTION_METHOD(POINT P)
    IF DISTRIBUTION_FUNCTION(P) equals to STONE
        THEN
            RETURN STONE
    ELSE
        IF CHECK_STALACTITE_MATERIAL(P) equals to STONE
            THEN
                RETURN STONE
        ELSE
            RETURN AIR
        ENDIF
    ENDIF
ENDIF

Figure 5.4 Pseudo code of stalagmite simulation

5.4 Experiment Results and Analysis

5.4.1 Stalactites and stalagmites final shape comparison

Figure 5.5 shows stalactites and stalagmites simulation results in 2D case with different radius ranges and different scales in stalactite shape formula (Equation 5.1).
From Figure 5.5, it can be seen that the final shapes of stalactites and stalagmites with different combination of radius ranges and scale values. In general, to get appropriate results, smaller radius ranges should combine with large scale values, and bigger radius ranges should combine with smaller scale values, or the result would be too slender or too huge. Naturally, we can recognise that in order to get better effect it is a good way to combining different types together. So some of stocky stalactites and stalagmites can form the ‘foundation’, and some other slender
ones can contribute more detail and fluctuations to the final result. Figure 5.6 shows the combination of radius with range from 5 to 10 and scale 10; radius with range from 10 to 20 and scale 1 and radius with range from 20 to 50, with scale 0.5.

Figure 5.6 Shapes of combination with different parameters

5.4.2 Three-dimensional surface comparison

Figure 5.7 and Figure 5.8 shows the wire frames and boundary surfaces of stalactites models before and after smoothing operation.
(a) Stalactite wire frame before smooth

(b) Stalactite wire frame after smooth

Figure 5.7 Wireframes of stalactites model before and after smoothing

(a) Stalactite boundary surface before smoothing
Figure 5.8 Boundary surfaces of stalactites model before and after smoothing

Figure 5.9 and Figure 5.10 show the wireframes and boundary surfaces of final cave model based on slander and sphere shapes with stalactites and stalagmites models.
Figure 5.9 Wireframes and boundary surfaces of cave model (slender shape based)

(a) Wire frame of cave model (sphere shape based)

(b) Boundary surface of cave model (slender shape based)
From Figures (Figure 5.7, Figure 5.8, Figure 5.9 and Figure 5.10) above, we can see the final cave models’ wire frames and boundary surfaces. As discussed before, the combinations of different parameters and foundation shapes could generate multitudinous cave models to meet different specific requirements.

5.5 Comparison with Related Work

The approach to procedurally generating stalactites and stalagmites used in this chapter is based on the formula proposed by Short et al. [2005]. In their work, they merely proposed a formula to model the chemical growth of stalactites but they did not perform a 3D computer simulation to visualise the results of their formula from a 3D modelling standpoint. This chapter presents the results of not only implementing their formula to generate 3D models of cave stalactites and stalagmites, but also to integrate the generation of these speleothems together with the procedurally generated cave foundation.

To date, there is relatively little research that has been conducted on the specific area of 3D modelling of speleothems. Kim et al. [2006] developed a method of modelling icicles and stalactites based on ice dynamics. However, their proposed approach is a physically based approach that requires a large number of iterations to
run. Results from their experiments indicate that it would take half an hour to generate a 2D picture. Compared to the approach presented in this research, in which the 3D cave model can be rendered in real-time, the proposed approach is more useful for the purpose of this research.

Tortelli and Walter [2009] presented a GPU based method of generating the growth of stalactites, stalagmites and cave columns in real-time. However, results of their approach show that their generated speleothems are smooth mesh models which do not have the complexity and roughness of realistic cave surfaces. This was not the purpose of the proposed approach, which seeks to generate believable cave features complete with rough surface properties.

5.6 Summary
This chapter describes a procedural approach to simulating two typical phenomena in natural caves: stalactites and stalagmites. The formula that described the relationship between the radius and the length of stalactite was given. From this, a temporary virtual height map was constructed to store all stalactites information. The relationship between stalagmites and stalactites was then described and the stalactites and stalagmites were merged into cave model using a redefined distribution function.
CHAPTER 6: CONCLUSION

The automated generation of virtual 3D cave models is an important but largely unexplored area of research. The goal of this research was to develop and examine a method of procedurally generating visually believable 3D cave models and to be able to present the resulting models to the user in real-time.

To meet these objectives, the research presented in this thesis introduced a new approach to procedurally generate 3D cave structures using procedural noise functions. The Perlin noise function was the main function that was used to give rise to smooth noise characteristics. Furthermore, diverse cave structures can be generated by using different noise functions and by adjusting parameters such as the bias and frequency values. An efficient method of storing the resulting 3D spatial cave data using voxel-based octrees was also presented along with a specialised bottom-up octree construction approach.

However, the use of noise functions with an octree data structure alone gave rise to the problem of ‘floating islands’ and ‘air pockets’ in the resulting cave structure. Thus, this research also developed a method of processing and removing these problem areas from the cave structure, while at the same time being able to determine the internal boundary surfaces of the cave. This method was based on the flood-fill technique, used in this research in conjunction with octrees. The boundary surfaces were used to construct a triangle mesh that could be rendered in real-time at interactive rates. Furthermore, unlike smooth iso-surface techniques, the approach conducted in this research produces surfaces with contain a certain degree of roughness which is characteristic of realistic cave surfaces.

This thesis then presented a method of smoothing the resulting polygonal mesh of the cave based on a Laplacian smoothing function. While the smoothing process initially gave rise to the problem of cracks appearing in the cave walls, this research developed an effective solution to this by splitting certain triangles to patch the cracks. The approach introduced in this research can effectively be applied to octree data structures to produce a resulting triangle mesh without any unwanted holes or gaps.

To increase the realism of the cave model, an approach to generating stalactites and stalagmites was developed. This was based on the formula proposed by Short et al. [2005] to model the growth of stalactites. These stalactites and stalagmites were
added seamlessly to the overall procedural cave generation process. This resulted in a full triangle mesh model of a 3D cave structure with stalactites and stalagmites. The mesh model produced can then be displayed in real-time using standard computer graphics rendering techniques.

In summary, this research have investigated and developed a unique approach to automating the process of generating visually believable 3D cave models with different cave characteristics that can be rendered in real-time. This contributes to current research efforts in procedural content generation and can be used in areas like computer graphics, virtual reality, video game development as well as movie production.

6.1 Future Work

For generating the cave foundation, although some scenes can procedurally be generated by combining several basic shapes such as spheres, boxes and cylinders, the parameters to control of final cave foundation are still not very well defined. For example, it is hard to generate specific cave models with rough ceilings and floorings, but with smooth surrounding walls. This can potentially be done by summing different weighted procedural noises with different frequencies. However, this is left as future work.

In addition, natural caves contain many other speleothems besides stalactites and stalagmites, such as columns and scallops. These phenomena can also be simulated using a procedural approach. However, new equations will have to be introduced to describe their shapes, and some algorithms will have to be developed to merge them into the cave model.

At present, this research only dealt with the procedural generation of the polygonal cave model. In order to make the final cave model appear realistic, the polygonal model will have to be textured using realistic natural looking textures. Since caves have complex shapes, traditional 2D texture mapping techniques will be hard to use when rendering cave models, as the 2D textures may be unrealistically stretched over the surface. Therefore, the procedural generation of 3D textures for the cave is a potential solution that can be explored.
REFERENCES


Flores, L & Horsley, D 'Underground cave sequence for land of the lost', in Proceedings of SIGGRAPH: Talks, 2009


Greeeff, G 2009, 'Interactive voxel terrain design using procedural techniques', Master thesis, Department of Mathematical Sciences, Stellenbosch University.


Olsen, J 2004, 'Realtime procedural terrain generation', Technical report, Department of Mathematics and Computer Science (IMADA), University of Southern Denmark.


Samet, H 1990, The design and analysis of spatial data structures, Addison-Wesley, Boston, MA.


Stam, J 2000, 'Interacting with smoke and fire in real time', *Communications of the ACM*, vol. 43, no. 7, pp. 76-83.


