Voxel-Space Shape Grammars

by

Zacharia Crumley

Supervisors:

Patrick Marais
James Gain

A thesis submitted to

THE DEPARTMENT OF COMPUTER SCIENCE,
FACULTY OF SCIENCE,
UNIVERSITY OF CAPE TOWN

in fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

August 22, 2012

Plagiarism Declaration

I know the meaning of Plagiarism and declare that all of the work in this document, save for that which is properly acknowledged, is my own.

Abstract

The field of *Procedural Generation* is being increasingly used in modern content generation for its ability to significantly decrease the cost and time involved. One such area of Procedural Generation is *Shape Grammars*, a type of formal grammar that operates on geometric shapes instead of symbols.

Conventional shape grammar implementations use mesh representations of shapes, but this has two significant drawbacks. Firstly, mesh representations make Boolean geometry operations on shapes difficult to accomplish. Boolean geometry operations allow us to combine shapes using Boolean operators (and, or, not), producing complex, composite shapes. A second drawback is that sub-, or trans-shape detailing is challenging to achieve. To address these two problems with conventional mesh-based shape grammars, we present a novel extension to shape grammars, in which a *voxel* representation of the generated shapes is used.

We outline a five stage algorithm for using these extensions and discuss a number of optional enhancements and optimizations. The final output of the algorithm is a detailed mesh model, suitable for use in real-time or offline graphics applications. We also test our extension's performance and range of output with three categories of testing: performance testing, output range testing, and variation testing.

The results of the testing with our proof-of-concept implementation show that our unoptimized algorithm is slower than conventional shape grammar implementations, with a running time that is $O(N^3)$ for an N^3 voxel grid. However, there is scope for further optimization to our algorithm, which would significantly reduce running times and memory consumption. We outline and discuss several such avenues for performance enhancement.

Additionally, testing reveals that our algorithm is able to successfully produce a broad range of detailed outputs, exhibiting many features that would be very difficult to accomplish using mesh-based shape grammar implementations. This range of 3D models includes fractals, skyscraper buildings, space ships, castles, and more. Further, stochastic rules can be used to produce a variety of models that share a basic archetype, but differ noticeably in their details.

Keywords

- Procedural Generation
- Shape Grammar
- Voxel
- Boolean geometry

Acknowledgements

The thankfulness I have towards my family for all their support and assistance along the way can't really be adequately conveyed in words, so I won't try. There were bumps in the road, but with family at your back, it's a much easier journey.

James and Patrick: Thank you for the guidance, wisdom, and occasional whip cracking when my chronic laziness infection acted up.

A big thank you to my classmates for the camaraderie, technical assistance, help with my algorithms, pub lunches, and providing opportunities to avoid doing work.

I would like to thank the anonymous examiners for their suggestions and feedback on my thesis.

The financial assistance of the National Research Foundation (NRF) towards this research is hereby acknowledged. Opinions expressed and conclusions arrived at, are those of the authors and are not necessarily to be attributed to the NRF. Funding assistance for this research was also provided by the University of Cape Town.

Some of the diagrams and images in this thesis were created by others, licensed under the creative commons licenses, or released into the public domain. Full attribution for these images is included in a section at the end of this thesis, on page 110. The original creators do not necessarily endorse me, this thesis, or my use of their work.

I thank the original creators for releasing their images under these licenses, which allow me to make use of their work, saving considerable time that would otherwise have been spent wrestling with Inkscape to make diagrams.

Thanks to the developers of the many open source applications that were used along the way in this masters. Your contributions to the Open Source movement are appreciated.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

For video games, virtual environments, and cinema special effects, cost-effective *content creation* is an increasing concern. In this context, *content* refers to models, animations, textures, sounds, and the like, – essentially, all the components that go into a video game, animated film, or virtual environment, which are not part of the underlying game engine or renderer.

The amount of content needed for these computer-graphics-related applications has been steadily growing with increases in computational power and the quality of graphics. We have now reached a point where hundreds of modellers, animators, and artists can work for months or years to create the content necessary for a single mainstream video game or blockbuster film [1, 52].

The large size of these content-development teams means the costs involved are significant. Salaries, equipment, office space and a plethora of other costs all make industry-grade content creation an expensive proposition. In spite of the number of people working on the project, long development times are still the norm, because authoring content is inherently a lengthy process.

For these reasons, content creators have begun turning to *procedural generation*, as a way of decreasing costs and shortening development times.

1.1 Procedural Generation

Procedural generation refers to the collection of methods designed to algorithmically generate content [38, 44]. That is, instead of having a human manually specify the content (often using specialized purpose-built software), a software program will be run that produces content, often adhering to some set of input constraints. Minimal human interaction is required in procedural generation – generally limited to setting the initial parameters of the algorithm, or providing example inputs.

Hence, procedural generation can significantly reduce content creation time and costs. The algorithms will run and produce content much more quickly than a human designer can create it, and because large amounts of content can be created automatically, the required number of content creators can be reduced, saving on employment costs.

Of course, procedural generation does have disadvantages, and is not a panacea for all the challenges of content creation. It cannot be used to create

every type of content (for example, no algorithms exist for mood-appropriate music, storylines, or voice acting), and fine-grain control of the output is often lost. Nonetheless, even with these constraints, procedural generation is an extremely useful tool that can save significant resources.

Procedural generation is a relatively mature technique, with one of its best-known earliest appearances being the 1984 video game Elite[1]. In that game, the eight different galaxies that the player could explore were procedurally generated.

The use of procedural generation in the early days of computer graphics was mostly driven by necessity, since the available memory was limited. Generating content on the fly reduced the program's memory footprint, because pre-created content did not need to be stored in memory.

A more detailed overview of procedural generation is presented in Section 2.1.

1.2 Shape Grammars

This dissertation focuses specifically on *shape grammars* [49]. These are a type of formal grammar that can be used to procedurally generate shapes and 3D models. In this section we present a brief overview of shape grammars, to clarify how we view them in the context of this thesis, and to provide background knowledge for readers who may be unfamiliar with them.

As with other types of formal grammars, a shape grammar consists of three components:

- A set of items, $\{S_i\}$, which forms the alphabet of the shape grammar.
- The axiom, A, an element of $\{S_i\}$, which is the initial item that the grammar begins from.
- A set of production rules, $\{R_i\}$ which take an element of $\{S_i\}$ as their input (potentially with additional metadata), and output zero or more elements of $\{S_i\}$ (potentially including additional metadata and performing arbitrary computation to determine the output).

The grammar begins with the axiom, A, which forms the initial *current shape set*, $\{C_i\}$. The grammar then operates over multiple iterations. In each iteration, an item C_i from the current shape set, which is the input to a production R_i, is selected and operated on by R_i to produce a new set of items, C_{new}. C_{new} then replaces C_i in the current shape set.

It is also possible that multiple items from the current shape set will be operated on in one iteration, in which case the grammar is said to operate *in parallel*. If only one item is operated on in each iteration, the grammar is said to operate *serially*.

In this way, the current shape set will be transformed (and generally expanded) across multiple iterations. Starting as only the axiom, A, the current shape set will eventually be transformed into the final shape set, $\{F_i\}$, which is the output of the grammar.

[1] Game website: `www.iancgbell.clara.net/elite/`

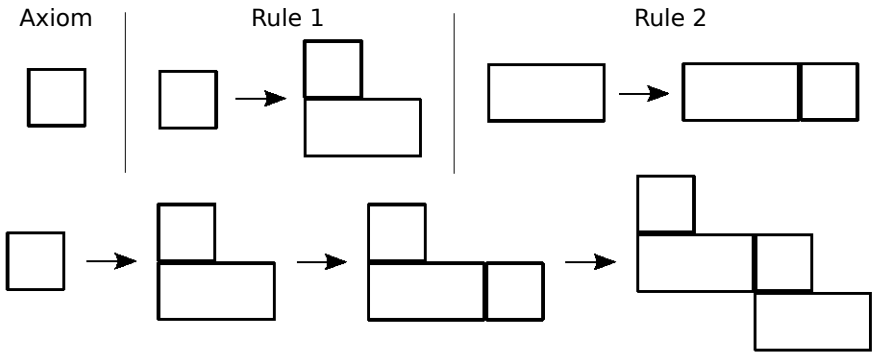


Figure 1.1: A simple shape grammar that produces an infinite series of 2D stairs, made up of squares and rectangle. The grammar's first three iterations are shown.

Shape grammars are distinguished from conventional grammars, in that their rules operate directly on geometric shapes instead of symbols from an alphabet. That is, instead of $\{S_i\}$ consisting of arbitrary symbols, all S_i are geometric objects (points, lines, cubes, octagons, etc.). These geometric objects can be of any dimension, but, in practice, are almost universally two- or three-dimensional.

The production rules, $\{R_i\}$, of a shape grammar are also different. Instead of performing simple string operations such as replacement and addition of symbols, the rules can perform geometric operations on $\{S_i\}$, such as rotation and scaling, in addition to the normal formal grammar operations.

This means that the final output, $\{F_i\}$, of a shape grammar will not be a string of symbols. Instead, it will be a collection of geometric shapes which, if the grammar was suitably designed, will collectively form the geometric model intended by the designer.

An example of a basic two-dimensional shape grammar is shown in figure 1.1. It consists of a square for the axiom, and two rules, one of which takes a square as its input and the other, a rectangle. This grammar is intended to create an infinite set of stairs, and operates serially, with the most recently added shape being the shape that is operated on in each iteration. The first three iterations are shown, and illustrate how the current shape set is transformed by the rules.

Shape grammars were originally developed in architecture as a tool for formalizing architectural design. Although still used for that purpose, they are better known in computer science as a method for procedurally generating models of buildings and other structures. This research focuses on shape grammars in that context: their use in procedural generation.

1.3 Shape Grammar Deficiencies

Conventional shape grammars operate on mesh-based representations of shapes. Shapes are recorded as collections of vertices (points) and edges (line segments between two of the points), which collectively form the mesh. Multiple edges form closed polygons, and those polygons form the surface of the shape – an example of how edges and vertices combine to form a mesh is shown in Figure 1.2.

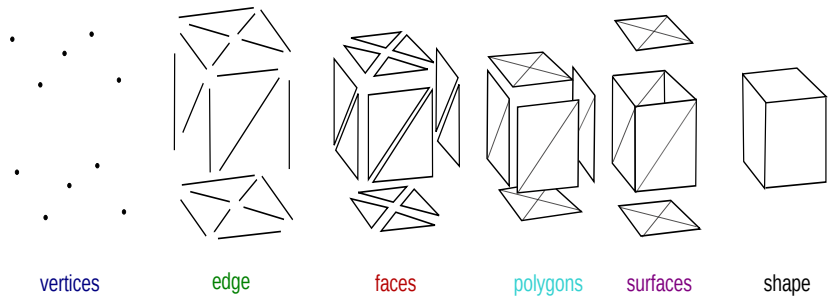


Figure 1.2: This figure shows how vertices and connective edges between them are combined to form mesh models of shapes. In this case 10 vertices and 24 edges are used to represent a 3D rectangle.

Surface detail is added by texturing the closed polygons (either the faces or polygons in Figure 1.2). This is done by projecting a 2D image (the texture) onto the flat surface. Each vertex is assigned coordinates from the texture, and the image is then stretched/scaled to fit onto the face or polygon. In this way, all flat surfaces in the final mesh will have visual detail displayed on them, in the form of a 2D texture.

Geometric operations on meshes are accomplished by acting on the vertices, with the edges being moved as a consequence. Rotation, scaling, shearing, and translation of the overall shape can be accomplished by moving the vertices in certain ways.

Conventional shape grammars operate on shape meshes. The rules create, and operate on, mesh representations of shapes, modifying them until a final collection of shape meshes is produced: the output of the shape grammar. Textures are often assigned to faces during shape grammar interpretation, based on the shape grammar rules.

However, there are two major problems with conventional mesh-based shape grammars, as used in procedural generation. The first deficiency arises from the difficulty of robustly applying constructive solid geometry (CSG) or Boolean geometric operations to meshes [8]. This is because overlapping edges and vertices must be identified and trimmed or removed, as appropriate. Then, new vertices and edges must be created to join the shapes in a watertight manner.

Boolean geometry operations are a powerful tool in geometric modelling and can be used to easily create a variety of shapes that are difficult to generate without them. They are used in engineering design and game engines for their ability to easily create such structures and meta-shapes, when other methods are more time-consuming and less elegant. For this reason, being able to perform Boolean geometry operations in shape grammars, and exploit their generative power, is desirable.

Although algorithms exist for performing Boolean operations on meshes [24, 45], these processes are complex; have edge cases that remain problematic; can suffer from numerical inaccuracies; and can create non-manifold, or badly sampled, meshes. Alternatively, this redundant geometry can be left hidden, but this is inefficient and can lead to visual problems, such as Z-fighting (where two polygons are coplanar and hence interfere with the correct display-

ing of the other). However, even with such an algorithm for handling redundant geometry, the overlap between different shapes can lead to texture seams.

The second limitation of mesh shape grammars is that texturing is done at a per-polygon level. Each shape, or pre-made piece of geometry, has fixed texture coordinates for each vertex and fixed associated textures (or in some cases, a small set to choose from). Hence the surface detail is often fixed, and the possibility for dynamic detailing is limited since, at best, different textures can be selected for each face, which is likely to lead to texture seam issues.

As an additional consequence of working with meshes, it is difficult to incorporate detailing that spans multiple shapes. Because textures are restricted to single surfaces, it is difficult to have detail that crosses between shapes (for example a series of racing stripes running the length of a car model produced by a shape grammar). Additionally, it can be hard to avoid texture seams, especially when the parameters of the shapes can vary.

Related to the above problem, 2D surface detail cannot easily be manipulated at a sub-face or sub-polygon level. Because a texture is a static image projected onto a flat surface, we cannot control details at a level lower than a single face.

Of course, one could theoretically edit textures dynamically to create details at a sub-shape level, but this rapidly becomes difficult, with many problematic edge cases (such as rotations, variations in the shape grammar output, and scaling of the model). Repeated loading and saving of image files is also too inefficient to be done regularly, and the modifications to the texture may interfere with special effects that rely on additional texture maps, such as bump mapping.

It is thus difficult to have surface details on a mesh model that span multiple shapes, and to control the textures at a sub-shape level. These limitations of being constrained to working with an entire face or polygon of a model restrict freedom in detailing.

1.4 Aims

This thesis aims to solve the above-mentioned problems of Boolean geometry operations and surface detailing in shape grammars, by modifying how the grammars are interpreted.

The identified problems both arise because conventional shape grammar implementations work with mesh representation of shapes. If we were to operate on non-mesh representations, these problems could be avoided.

We propose the use of voxel representations of shapes as a solution, since they allow us to avoid the above limitations, and provide some interesting new capabilities for shape grammars.

1.5 Voxels

A *voxel* is the 3D analogue of a pixel, that is, an element in a discrete, regular 3D grid. It can be a binary value (true or false), or contain additional information (such as velocities for a grid-based water simulation). A diagram showing a simple voxel data set is shown in Figure 1.3.

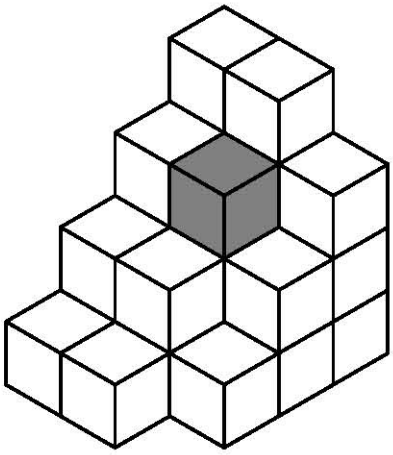


Figure 1.3: A simple example of a small *voxel* data set. One of the voxels has been set to grey.

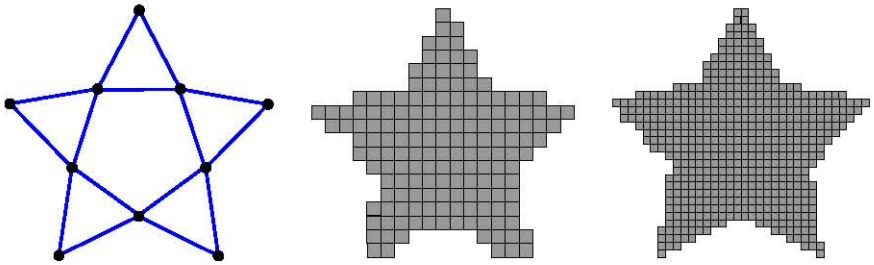


Figure 1.4: This figure shows a *mesh* representation of a 2D star on the left. The blue lines are edges and the black circles are the vertices of the model. The two pictures on the right show voxel representations of the same shape, taken at different resolutions. Note how increasing the voxel resolution decreases aliasing artifacts and provides a better approximation of the shape.

Voxel data is used in many domains, such as: the output of MRI scans, where the final data set is constructed by stacking up slices of data [6]; for representing landscapes in video games [10], where it allows the representation of caves and overhangs; and a host of scientific data applications, ranging from astronomy to geology [6].

For our purposes, we are interested in voxel grids where each voxel stores a binary value indicating whether it is solid or empty. In this way, shapes can be represented by groups of voxels. Instead of having a collection of polygons that make up the shape, we have a collection of adjacent voxels, which collectively form the desired shape. These voxels will be set to solid, while their surrounding neighbours will be set to empty, meaning that the solid voxels form a 3D representation of the shape, surrounded by empty space. Additional metadata (such as detailing information) can also be associated with each voxel. A simple 2D example, that compares mesh and voxel representations is found in Figure 1.4. Further discussion on voxels, and the challenges they present, is deferred to Section 2.3.

1.6 Voxel-Space Shape Grammars

In this work we present an algorithm for interpreting shape grammars in voxel-space, instead of using mesh representations of shapes. Modifying shape grammars to operate in a voxel-space solves the limitations mentioned earlier. Boolean geometry operations on voxels are trivial [21] and hence CSG operations are easily applied [8]. It is a simple matter of iterating over all voxels in the affected area and making them either solid or empty, based on simple logic. For example, to create the geometric union between an overlapping cube and sphere, simply loop over each shape in turn, and copy all 'solid' voxels into the output grid. The resulting output will be a voxel grid that represents the union of the two shapes. In a shape grammar context, this can be achieved by having each shape generated by the grammar have an associated flag indicating whether it is additive, or subtractive, meaning that the voxels within that shape will either be set to solid or empty, respectively. Figure 1.5 shows a comparison between mesh-based and voxel-based Boolean geometry operations.

Accomplishing this with meshes would require intersection detection between polygons, followed by clipping of edges and vertices, and then the creation of new vertices around the clipped areas. Even then, there would still be texture seam issues.

One issue with voxel representations of shape is that the precision of the initial geometry can be lost. When converting a shape defined as a mesh or mathematical equation into a voxel grid, and then back into a mesh, the fine details of the original shape can be lost. In many cases we will want a mesh of our shape for display purposes, so this is a concern, since produced meshes may lack the fine detail desired by the user. However, there is an easy solution to this problem: increase the resolution of the voxel grid, as this will decrease the aliasing issues that lead to precision and fine details being lost. How much the resolution is increased by should be decided on a case-by-case basis, balancing the need for fine detailing against the larger storage requirements that increasing the grid resolution brings. Figure 1.4 shows an example of how increasing the resolution achieves a more accurate representation of the shape.

The second deficiency that we identified in mesh-based shape grammar interpreters, the limitations of detailing mesh models, can also be addressed by using a voxel representation of shapes. Textures can be assigned on a per-voxel basis, which removes their dependence on the polygons and faces of the mesh.

In this manner, each voxel will have associated detailing information, that is used when displaying the model. This detailing information is not limited to textures either, since there are methods to render voxel grids that do not use textures, such as ray tracing [27].

Because the detailing information is assigned on a per-voxel basis, and not a per-polygon basis, we can control the detailing at a much finer level and we are not restricted by polygon boundaries. When the shapes are represented as voxels, there is no concept of boundaries between shapes (the voxels are all just values in a 3D array), and hence the detailing is not limited by those boundaries. This means that we can create consistent global detailing patterns that would be difficult to do procedurally in a purely mesh-based situation. For cases, where we want knowledge of the shape boundaries, metadata can be stored to indicate which voxels belong to which shapes, and the boundaries can then be derived.

Additionally, since the voxel grid resolution can be adjusted, we can ensure

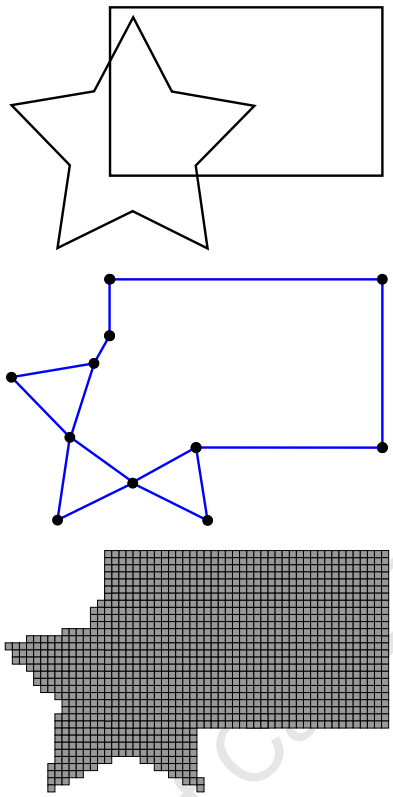


Figure 1.5: This figure demonstrates the 2D geometric union between a star and a rectangle (top). With mesh-representations (middle) there are vertices and edges that overlap and must be clipped, or removed. Additionally, new vertices will need to be created where edges from the different shapes meet. This is complex, and there are a number of edge cases that are problematic. With a voxel representation of the shapes (bottom), finding the union is simple. All voxels that form part of the shapes are simply set to 'solid', and the resultant voxel data set is the union of the two shapes.

that a voxel will be smaller than a polygon face (by increasing the resolution to make it so if it is not). Hence, we are able to easily detail at a sub-polygon level, which is significantly more complex to do with meshes.

As a side-point, it should be noted that this method of per-voxel detailing is not specific to our proposed algorithm. It could, in fact, be used independently for procedurally detailing any voxel-based model or data-set. In this thesis we restrict ourselves to considering use of the technique in the context of shape grammars, but almost all of our observations and suggestions would be applicable for use in other contexts.

Hence, interpreting the shape grammar in a voxel-space, instead of a mesh space, solves the two deficiencies of conventional shape grammar implementations. A more detailed look at our algorithm for accomplishing the above mentioned operations appears in Chapter 3.

1.7 Research Questions and Contributions

This thesis explores the interpretation of shape grammars in a voxel space, in contrast to the traditional mesh geometry approach. Our major, and novel, contribution is an algorithm for this process. The algorithm consists of four principal stages, and produces a mesh model, suitable for use in real-time, or offline, 3D graphics. We also discuss optimizations and optional steps for the algorithm, as well as testing its performance and range of output.

To determine the utility of our new system, we explore and evaluate the following research questions:

- Does using a voxel representation of shapes, instead of a mesh, solve the limitations of shape grammar implementations as identified? Can our voxel-space shape grammar extensions produce models that are difficult or impossible to produce with meshes?
- What limitations, if any, does working in a voxel-space bring? Further, are any of these limitations severe enough to make our algorithm nonviable?
- Are our extensions computationally efficient enough for real-world use?

We have already provided a theoretical answer to the first question, but in the rest of this thesis we assess the practicality of such a system. Likewise, we have covered some obvious areas for the second question, but in later chapters we give a more rigorous analysis of the limitations and whether we can overcome them.

These three research questions represent the major objectives of this work. The goal is to develop a system that expands the generative range of shape grammars without losing existing functionality, and while remaining computationally viable. The research questions examine these three components of our goal and thus their answers form the major success metrics for our work. Determining these answers is the focus for evaluation of our research.

1.8 Overview of Thesis

The remainder of this thesis is presented as follows:

Chapter 2 covers background and related work relevant to this thesis. It gives a brief overview of the wider field of procedural generation before delving into shape grammars and voxels specifically.

In Chapter 3 we present a framework for utilizing the extensions developed in this research. Each of the five algorithm stages is covered in detail, and we discuss implementation details and optimizations.

We performed three types of testing and evaluation of our algorithm, and these are described in Chapter 4. Each type of testing and its results are covered, and then limitations and conclusions are discussed.

Finally, in Chapter 5, we summarize the rest of this thesis, draw conclusions from our findings, and present some suggestions for future work.

Chapter 2

Background and Related Work

In this chapter we examine the existing academic work and related background knowledge relevant to our project. This includes procedural generation as a whole, shape grammars in specific, and previous use of voxels in procedural generation.

Section 2.1 focuses on the wider field of procedural generation. It looks at its origins, its advantages and disadvantages, discusses a few of its key elements, and finally delves into four of the best known types of procedural generation. Although our research is focused on a specific area of procedural generation, knowledge of the wider field is important for understanding our research, which is why this section is included.

In Section 2.2, we cover previous work in the area most relevant to this research: shape grammars. This section is focused on specific pieces of important research in the field, from the inception of shape grammars, to recent developments. Our work incorporates ideas from a number of the papers mentioned in this section.

The use of voxels in procedural generation is discussed in section 2.3. Although voxels are not widely used in mainstream procedural generation, they have seen use in research, and some video games. The section also discusses data structures for efficiently managing voxels.

2.1 Procedural Generation

In the context of this dissertation, *procedural generation* refers to the collection of methods designed to algorithmically generate content for games, film, and virtual environments.

The content produced by these methods is primarily generated by an algorithm, with minimal human interaction – generally limited to setting the initial parameters of the algorithm. Examples of such parameters include: a small production rule-set that can be used to make an entire forest, or a 128-bit random seed to an algorithm that creates an entire landscape.

Contemporary use of procedural generation is driven by budgetary and time constraints. The continuing increase in computer processing power means that

Figure 2.1: This image shows a nature scene, where the foliage and plants were procedurally generated by the SpeedTree software package. Copyright ©2011 IDV, Inc. All rights reserved.

the amount of content in graphics applications, and its complexity, has been steadily increasing. As a result, the resources required to create all necessary content has skyrocketed, resulting in larger costs and development times.

For example, the multitude of special effects in the blockbuster film Avatar[1] took three years to create, with a peak of 900 people working on them [52]. The film cost approximately $310 million to produce [1], a large percentage of which was used to create the computer generated visuals.

Procedural generation provides a way to quickly generate large amounts of content with minimal human interaction, reducing these increasing development costs. One of the better-known commercial procedural products is SpeedTree[2], a middleware product for procedurally creating trees and plants. Other examples include CityEngine[3], used for generating buildings and whole cities, and Terragen[4], which creates landscapes. Images of the content these products can produce are shown in figures 2.1 to 2.3.

Procedural generation also appears in ad hoc applications outside industry and academia. The Demoscene[5] is the most prominent example of this. In this context a 'demo' is an executable file (constrained to be less than a certain size) which generates a visual and auditory presentation. Because the executable size is so constrained, procedural generation is the only viable option for producing content. The Demoscene subculture has formed around this, and produces some

[1]Film website: `http://www.avatarmovie.com/`
[2]Product website: `www.speedtree.com`
[3]Product website: `http://www.esri.com/software/cityengine`
[4]Product website: `www.planetside.co.uk`
[5]More information about the Demoscene can be found at: `www.demoscene.info`

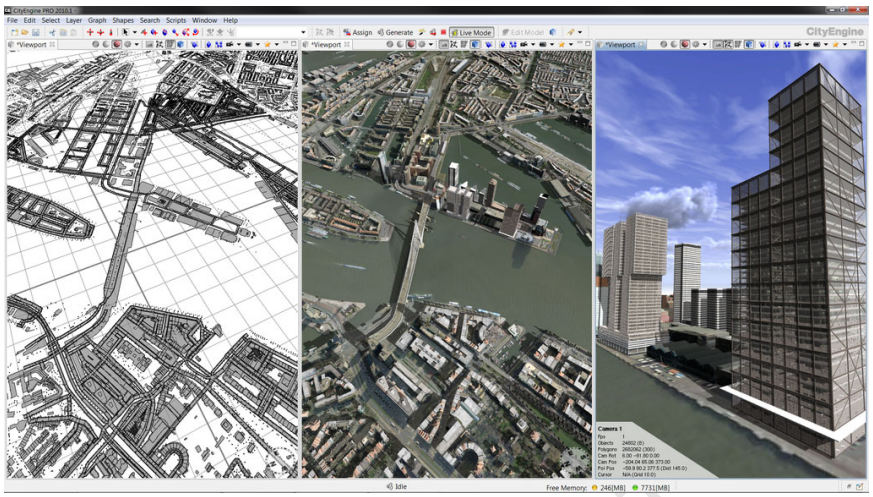


Figure 2.2: An image of the CityEngine suite in the process of procedurally creating a city. Copyright ©1995 – 2011 Esri.

Figure 2.3: This Martian terrain was created procedurally, using the Terragen software package. ©2005 www.planetside.co.uk

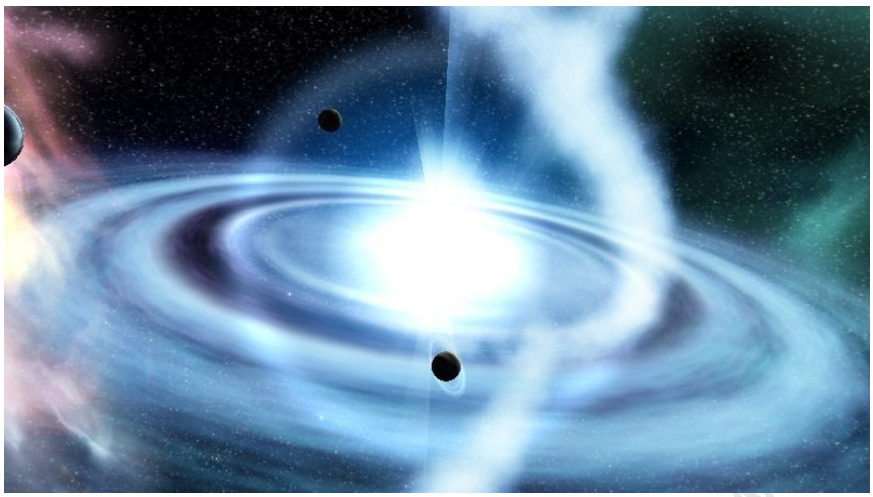


Figure 2.4: This is a screenshot from the demo 'Beyond' created by Demoscene group 'Conspiracy'. The executable is only 54 kilobytes in size, but produces 4 minutes of visuals through procedural generation.

extremely impressive work. See figures 2.4 and 2.5 for sample images. In some cases the Demoscene has even inspired new techniques that are used elsewhere. For example, in the video game Spore[6], user-created creatures are procedurally animated and detailed [15]. Figure 2.6 shows some user-generated creatures from the game.

It is important to note that procedural generation cannot generate every form of content, and is limited to content that has certain properties, such as a fractal description, elements of randomness, clear mathematical definitions, or exploitable commonalities across all instances of the content. Properties like these provide an underlying structure that can be evaluated and produced algorithmically.

Unfortunately, procedural generation usually results in less control over the produced content. High-level constraints can be supplied but, in most cases, there is little that can be done to change specific low-level or localised details. One way to counter this problem is manual editing of the generated content. With this, a user will take the procedurally generated output, and modify it by hand, using specialized modelling software, such as Blender[7] or 3ds Max[8].

Alternatively, there are other methods proposed in research, such as adjectival interfaces which use machine learning to associate words with characteristics in produced content [16]. After the system has been trained on which adjectives should produce which features in the output, the user can then select a number of adjectives and the system will output content that has features matching those adjectives.

Visual grammar editing [29] is another option, where a user can change generating rules for a grammar to produce specific results in the output, using

[6]Game website: `www.spore.com`
[7]Project website: `www.blender.org`
[8]Product website: `www.autodesk.com/3dsmax`

Figure 2.5: This is a screenshot from 'Interceptor', a demo made by Demoscene group 'Black Maiden'. The executable is only 157kb in size, but it features a full 3D model and a number of graphical effects.

a graphical interface, which is easier to work with than text-based rules.

The most common elements of procedural generation are self-similarity; randomness; and elementary mathematical functions such as sin, cos, and the equations of shapes.

Self-similarity refers to structures that exhibit features that are similar (or identical) to other parts of the same structure. In the context of procedural generation, self-similarity usually means that features are repeated at multiple scales. For example, the branch of a tree often looks like a smaller version of the same type of tree, which has undergone an affine transformation to shrink it.

The main purpose of randomness is to produce a variety of outputs from a single algorithm. A deterministic algorithm will produce the same output given the same input, but an algorithm that uses a different random seed as input for each run can produce multiple different outputs.

The use of randomness also allows a type of compression. Instead of storing the final content, only the algorithm to create the content and the random seed need to be stored, since the content can be perfectly reproduced from these. In almost all practical cases the content is much larger than its generating algorithm, netting significant gains in storage space [38]. This method was used to save space in the early days of computer games, and is known as Kolmogorov compression, from the idea of Kolmogorov complexity [28].

Mathematical functions such as trigonometric functions, equations of shapes and image filters provide the primitives for procedural generation. These can be modified or used with other functions to produce specific effects or shapes, such as distorting a sphere mesh into an ellipsoid and then randomly displacing the vertices to create surface detail.

There are many other, less common, sub-components that can go into a

Figure 2.6: This image shows four creatures created in the video game Spore, by Maxis. These creatures are designed by a user, but their detailing and animations are procedurally created. These images were taken from [15] and are the property of their original creators.

procedural generation algorithm. For example, the video game Borderlands[9] procedurally generates weapons for the player. Specifically, each weapon has basic characteristics and special enhancements that are procedurally decided upon, giving the game a vast range of player-usable weapons that would have taken years to be hand-made by a human. A unique aspect with this approach was that the procedural algorithm had to consider game mechanics balance, and ensure that none of the weapons it created were underpowered, or overpowered, by balancing the good and bad characteristics out.

Another example is the AI 'Director' found in the Left 4 Dead games[10]. The game's levels are mostly static, but the 'Director' procedurally generates narrative elements of the game, with the goal of creating tension in the players. The locations where zombie enemies spawn from, and their number and frequency, are dynamically chosen to keep things unpredictable for the players.

A large amount of work has been done on procedural generation, both in academia and in industry, and a full survey of the many types is beyond the scope of this work. We point readers to surveys of the field [22, 38, 48] for further information.

We now discuss some of the best-known and most well-established methods in procedural generation. Although the following methods are not directly relevant to our research, they do suggest ideas and techniques that could be helpful in our work. Additionally, they provide examples, and a more concrete understanding, of the hallmarks of procedural generation.

2.1.1 L-systems

L-systems are a class of formal grammar that execute in parallel, rather than serially. This means that L-systems often produce content with a high degree of self-similarity, since the same rules that created the symbols in the previous iteration will operate on those symbols again, producing the same style of output. When the scale is decreased across successive iterations, the L-system will produce fractal output.

This makes L-systems well-suited to generating content which exhibits fractal behaviour or has similar structures across many scales, such as road networks [42] and plants [44] (for which L-systems are considered the de facto method).

An L-system consists of an alphabet of symbols, covering all possible symbols that will appear during interpretation and in the final output; a set of productions (known as the rule set) which replace symbols with zero or more other symbols; and an axiom symbol, which is what the interpretation starts with.

At each iteration, every symbol in the output string (initially only the axiom) for which a production exists in the rule set (making it a non-terminal symbol) is replaced by the right hand side of that production. Symbols which cannot be used as the input to any of the rules are known as 'terminal symbols' and are not operated on. See Figure 2.7 for an example of a simple L-system being interpreted.

This process is repeated on the new output string for each iteration until either: there are only terminal symbols left in the output string (symbols which

[9] Game website: `www.borderlandsthegame.com`
[10] Game website: `www.l4d.com`

Alphabet: A, B
Axiom: A
Rules: (A -> AB), (B -> A)

Output

```
n = 0 : A
n = 1 : AB
n = 2 : ABA
n = 3 : ABAAB
n = 4 : ABAABABA
n = 5 : ABAABABAABAAB
n = 6 : ABAABABAABAABABAABABA
n = 7 : ABAABABAABAABABAABABAABAABABAABAAB
```

Diagram

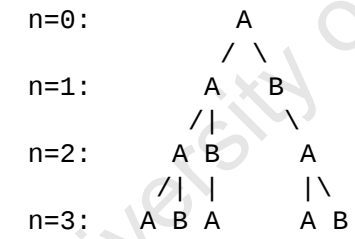


Figure 2.7: This figure demonstrates a simple L-system and how it is interpreted over a number of iterations. This L-system is actually the original L-system that Lindenmayer proposed to model the growth of algae. The alphabet consists of two symbols, one of which is the axiom, and two rules. Seven iterations of the L-system are shown, and a visual diagram of the first three iterations is provided for clarity. This figure was derived from the Wikipedia article on L-systems. Please see the Acknowledgements section on page 110 for full attribution and licensing of this derivative work.

do not match any productions), or until some pre-selected iteration limit is reached.

The string of symbols that is produced by the L-system is then interpreted by some method to produce the actual content. The best known interpretation method is turtle interpretation, used for trees and plants [44].

Turtle interpretation is most widely recognized as a feature of the Logo programming language. The turtle (or cursor) has a position, orientation, and is either down (creates visible lines when it moves) or is up (does not create visible lines when it moves). It follows a list of instructions than can change the above attributes and move the turtle. This means that arbitrary vector images can be produced by a series of turtle instructions.

In L-systems, a similar idea is used. The output string of the L-system is interpreted left-to-right as a series of instructions for the turtle to follow, which produce a 2D, or 3D, collection of lines that form the branches of the tree. For actual use in games and virtual environments the lines will be converted into textured cylinders or surfaces, but the same basic idea is still used at the core.

The rule set and axiom of the L-system control the characteristics of the produced content. For example, when building a grid-like road network from an L-system, the rules would ensure that roads are roughly straight and either parallel or perpendicular to each other.

There have been many extensions and enhancements developed for L-systems, the foremost of which we mention here.

Stochastic L-systems [7] introduce randomness in rules, so that the rules can randomly choose outputs from a given selection, instead of choosing deterministically. This increases the range of outputs possible from a single L-system.

With *environmental awareness* [39], L-system rules can query the virtual environment around the generated content and adjust outputs according to that information. For example, a plant could change it's growth direction over multiple iterations to maximize sun exposure.

Finally, having parameters associated with symbols [44] allows a whole new level of metadata to be used by the rules. Parameters can be arbitrary information not easily controlled via grammar symbols that is defined or used in the generation process. For example, the thickness and bark colour of branches in a tree.

Enhancements such as these, combined with a long established history, mean that L-systems are one of the fundamental methods of procedural generation.

2.1.2 Terrain Generation

Terrain generation is aimed at producing models of landscapes, usually encoded as 2.5D height maps, that appear realistic. A 2.5D (two and a half dimensional) height map is a two-dimensional grid, where each element in the grid stores a height. In this way, the grid can represent terrain by indicating the heights of the ground at each point in a rectangular area. Height maps are the most common method of representing terrain, since they are simple and can be stored efficiently using image compression techniques. However, they are unable to represent fully 3D terrain, such as caves and overhangs.

Terrain generation was pioneered by Ken Musgrave [36], but has since been further expanded by others [40]. Musgrave's approach made use of fractals,

where a polygon is recursively subdivided and the vertices of the new polygons are randomly displaced. These displacements at multiple scales create a plausibly realistic landscape.

Since then, many newer algorithms for producing terrain have been developed: for example, Perlin noise [43], a randomly generated type of gradient noise where features are of similar size; genetic algorithms [41], a heuristic based search technique, which is used to find terrain with features similar to those in a given database of samples; erosion-based methods [37], which aim to simulate the natural processes that create real landscapes; and patch-based methods, which stitch together small pieces of real-world terrain data to produce larger landscapes [57].

The different methods of terrain generation each have their own advantages and disadvantages, but these tend to reduce down to a realism versus complexity/performance trade-off. Hence, the choice of algorithm for a specific implementation usually ends up being a balancing act between the quality of landscape required, and the computational resources available. For further information on terrain generation, we refer readers to surveys of the field [48].

2.1.3 Texture Creation

Textures are 2D (and sometimes 3D) images that are projected onto the surfaces of polygon models to efficiently simulate visual detail. A variety of methods have been developed for procedurally creating textures.

There are methods that create completely new textures from random seeds. The foremost of these is Perlin noise [43], which has the advantage that all visual features are of the same size. This affords the user a great deal of control, because multiple items of Perlin noise can be easily scaled and combined to produce a range of different visual effects. The downside to these is that they tend to be specific to a certain class of texture. For example, Perlin noise is well-suited for effects such as smoke or flame, but would not be able to create a grained wood texture. An example of Perlin noise is shown in Figure 2.8.

Another approach is to use physics simulations, which have visual representations, such as reaction-diffusion systems. Reaction-diffusion systems are methods of modelling how substances will tend to be distributed after interacting with each other and diffusing into the local area, in the form of differential equations.

Turk's work [51] on creating textures for arbitrary polyhedral surfaces pioneered the approach of using reaction-diffusion systems. It operates by simulating reaction and diffusion directly on the mesh, and using the results to assign colour values to positions. See Figure 2.9 for some examples of textures produced by his work.

Although these simulation-based methods can produce good quality textures for certain situations, they are often more computationally intensive than alternatives, and have a generative range limited to a certain class of texture.

Finally, there is the cell-based approach, where the texture space is subdivided into cells based on randomly placed points. These tend to produce textures with an 'organic' appearance.

Voronoi diagrams are commonly used in this approach. A Voronoi diagram occurs when dividing an area of space up around a set of generator points in the

Figure 2.8: A sample of randomly generated Perlin noise.

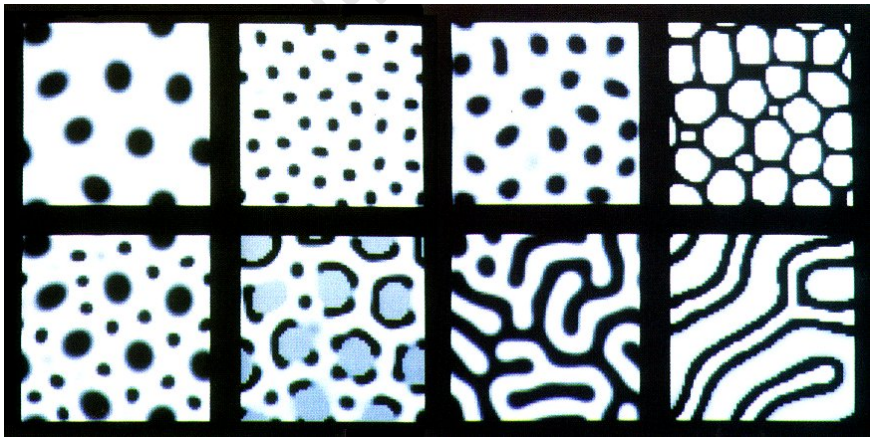


Figure 2.9: This figure shows several textures generated by reaction-diffusion processes. This image was taken from [51] and is reproduced by permission of the Association for Computing Machinery. ©ACM 1991.

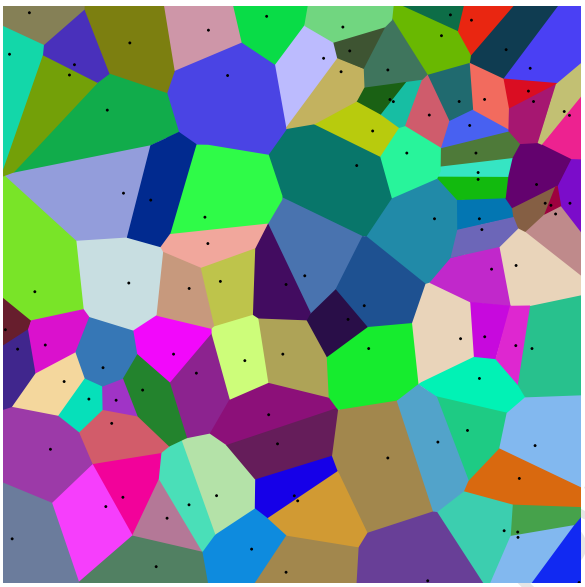


Figure 2.10: An example of a colorized Voronoi diagram showing how space is divided up around the black dots so that every point in the image is associated with its closest dot.

space, such that every point in the space is associated with its closest generator point (see figure 2.10). The divided space is known as a Voronoi diagram.

Itoh et al. demonstrate this cell-based technique in their work [18] where a region is subdivided into pseudo-Voronoi tessellations that resemble reptile skin.

Similar to simulation-based methods, cell-based approaches can produce high quality content within a certain scope, but they are generally limited to that scope.

There are also algorithms that take an example texture and create new textures similar to the example. The first efficient method of this type was developed by Wei and Levoy [53]. It operates on a per-pixel level by selecting the pixel colour that best matches the surrounding neighbourhood of pixels. See figure 2.11 for an example of this process.

This process is accelerated by using tree-structured vector quantization. This is a technique for converting a large multi-dimensional search space into a tree data structure which greatly reduces the time to search for items ($O(log(N))$ instead of $O(N)$). This requires that the search space can be ordered in some consistent method, but drastically reduces running times.

It should be noted that acceptable textures can only be generated from samples that meet certain requirements. Specifically, the texture should consist of only features that are similar across the entire texture in terms of size and appearance (Wei et al. describe as the property of 'stationarity'), and each pixel's colour must be associated only with the surrounding nearby pixels (described as 'locality' in the original paper). This is because the algorithms are only able to match and imitate existing patterns with relatively small features, and cannot extrapolate larger complex features.

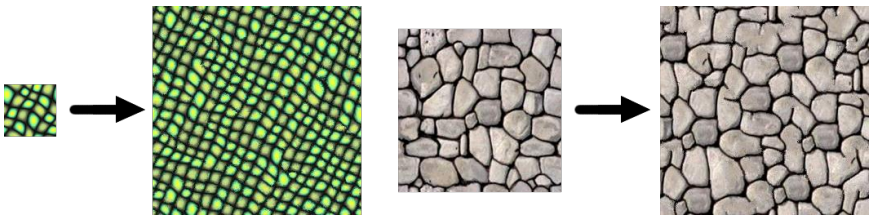


Figure 2.11: An example of texture synthesis by Wei et. al.'s texture synthesis algorithm. The larger images are produced by the running the algorithm on the smaller images.

For example, an image of tree bark would work as an input, since the pieces of bark are all very similar to each other, and the image consists of nothing but these features. A picture of a face would fail, however, because there is no regular repeating pattern that spans the entire image, and the algorithm cannot extrapolate that the face should be connected to a neck and body.

These sample-based methods tend to produce higher quality output than the methods that create entirely new textures, but are significantly slower. Additionally, they require example inputs to operate, and cannot create textures from scratch.

2.1.4 Random Level Generation

The need to randomly create game levels, or stages, arose out of the early ASCII graphics Roguelike games[11], as a way of varying subsequent playthroughs. These early algorithms were simple. For example, NetHack[12] simply creates a set of non-overlapping rectangles, randomly generates content in them, and then connects them with passages (see Figure 2.12 for an example of a generated level).

As time passed and technology improved, randomly created levels and their generation algorithms became increasingly complex. For example: the game Diablo 2[13], where pre-built elements are incorporated into the levels, and Infinity: The Quest for Earth[14], which procedurally generates an entire galaxy, down to the level of terrain on planets. Infinity's generating algorithm is proprietary, but is known to be based around a hierarchical octree and aims to produce content that is relatively consistent with the known make-up of our galaxy.

Some level generation algorithms even incorporate complex realistic elements, such as the geological simulations and erosion in Dwarf Fortress[15]. The basic terrain is generated using fractal-based algorithms, but computationally heavy simulation processes are done to produce a very detailed world map.

The majority of the work in this field comes from the gaming industry, and not academic research. Unfortunately, this means that details of the algorithms are not always available.

[11] More information on Roguelike games available at: `en.wikipedia.org/wiki/Roguelike`
[12] Game website: `www.nethack.org`
[13] Game website: `www.blizzard.com/games/d2/`
[14] Game website: `www.infinity-universe.com/Infinity/`
[15] Game website: `www.bay12games.com/dwarves/`

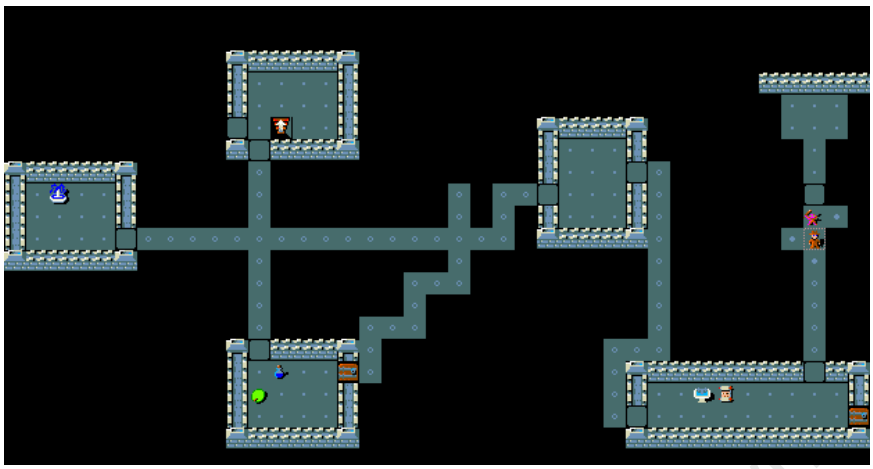


Figure 2.12: A random dungeon generated by the Roguelike game, NetHack.

2.2 Shape Grammars

Stiny and Gips [49] first developed shape grammars in an attempt to formalize architectural design. Although still used in architecture, they have also been adapted for use in the procedural generation of structures for other applications, such as virtual environments, video games, and historical reconstructions [34, 56]. These computer graphics applications of shape grammars are the focus of this section.

The original definition of shape grammars proposed by Stiny and Gips is very broad, and allows scope for a large amount of complexity in shape grammars. In particular, it allows for emergent shapes, where one or more shapes could combine to form a new shape, which would then be treated as the new meta-shape, and not as the individual shapes that make it up. For example, two identical right angle triangles could be placed with their hypotenuses adjacent to form a rectangle. In the original definition of shape grammars, this would then be recognized and operated on as a rectangle, and not as two triangles.

However, writing a shape grammar interpreter that can correctly handle emergent shapes is challenging, due to issues of recognizing new shapes formed from collections of other shapes. This means that, in practice, implemented shape grammar interpreters do not support the full complexity of shape grammars as proposed by Stiny and Gips. Instead they generally operate on symbolic evaluation, with a symbol for each shape, similar to how formal grammars work. In the rest of this thesis, we limit ourselves to dealing with this restricted, but implementable group of symbolic shape grammars.

Additionally, most shape grammar implementations have terminal symbols included in their definitions, which are treated as placeholders for actual pieces of geometry. Non-terminal symbols represent intermediate shapes, which are operated on by the rules until only terminal shape remain. This means that most shape grammar implementations use rule sets intended to run until all symbols are terminal, rather than running for a fixed number of iterations.

Early implementations of shape grammars were simplistic, with a limited

range of output. They operated deterministically on simple collections of shapes and produced straightforward models, with minimal variation.

However, over time, several ideas from procedural generation, principally from L-systems, were incorporated into shape grammar implementations. These enhancements expanded the generative power of shape grammars significantly.

Most notable among these extensions were environmental sensitivity and stochastic rules.

With environmental sensitivity [39], rules can query the environment and the current set of shapes, and then adjust their output based on the information they retrieve. For example, a rule producing doors in the wall of a house might query its neighbourhood to ensure that it will not place a door overlapping with a perpendicular wall, and a rule creating solar-powered geysers on the roofs of buildings would query the sun's direction and rotate the panels appropriately.

Stochastic rules [7] introduce randomness, by randomly choosing different outputs and parameters to achieve variation in the shapes generated. For example, when generating a ship model with a shape grammar, the dimensions of the ship could be randomized within certain ranges, and the height and location of the mast could be randomly selected.

The major benefit of stochastic rules is that multiple models conforming to a coherent style and theme can be produced by one shape grammar. When creating a city of buildings, one well-designed stochastic shape grammar could generate all buildings of a certain class (skyscrapers, warehouses, residential buildings, etc.), instead of each building requiring its own shape grammar.

Shape grammars were extended explicitly for building generation using *split grammar* rules [56], which focus on the subdivision of shapes. For example, a building's wall is divided first into floors, then the floors into different rooms, and finally the walls of the rooms into different windows.

Split grammars are particularly well suited to façade generation [56] (generating the side faces of buildings) due to the regular, grid-like layout of building windows. They work by recursive subdivision of a shape, guided by following productions from a rule set. The final set of shapes that arise from the repeated subdivision form a model of the desired building wall.

However, split grammars are less successful at creating the internal structure of buildings, due to the internals of buildings being much less regular and not as grid-like as the exteriors of buildings. However, other techniques have been developed for generating building interiors [14, 32].

Another problem with split grammars is that they are not the best tool for creating the basic shape of the building, since buildings are better represented as the combination of shapes, instead of subdivisions of them. Early approaches to this problem of creating the building's basic shape tended to be simplistic, such as using n-sided prisms as the split grammar axioms and then running a shape grammar on each of the prism's sides, in isolation from the others [11].

Subsequently, better solutions for this problem were developed. For example, starting with aerial imagery, and using image recognition to determine the footprints of buildings in the imagery, and reverse projection operations to create a simple 3D geometric representation of the building [26]. Alternatively, the building's basic structure can be created by shape grammars themselves, by combining geometric primitives, often with stochastic rules [34].

Along with the other extensions mentioned, the features of split grammars have since been incorporated into current grammar implementations, unifying

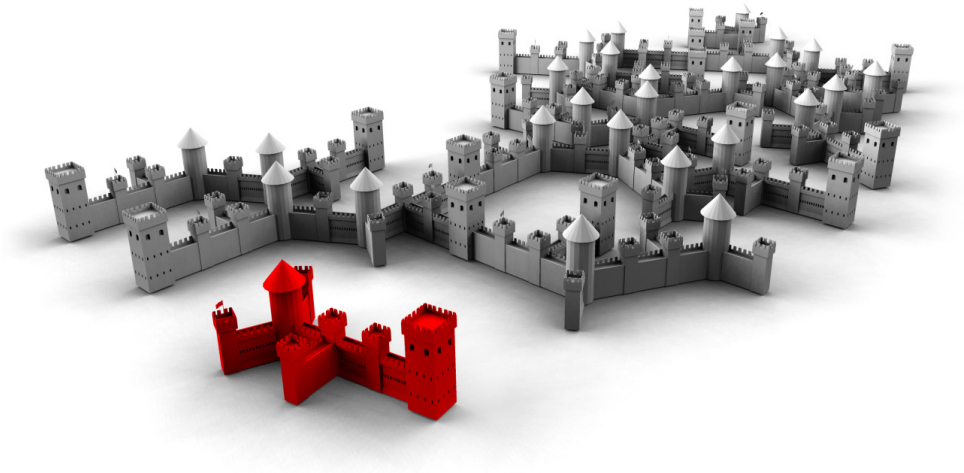


Figure 2.13: This image, taken from the work of Bokeloh et al., shows how an input to their algorithm (the red model) can be analyzed and used to produce output models in the same style (the grey models).

all features under one grammar, allowing for greater ease-of-use [34].

The range of output possible from modern shape grammar implementations is extremely broad [34] and as a result, shape grammars are considered the industry standard for procedurally generating architecture. They are able to create whole cities with realistic buildings (using additional techniques [42, 50] to create the road networks and city block layout). The best example of this in practice is the CGA grammar of CityEngine[16], a commercial product for procedurally creating cities and buildings.

Example-based shape grammars can be used to generate different models in the same style as the given-example. Input can be an image of a building's façade, as in the work by Müller et al. [35], where image analysis techniques are used to identify the features of a façade, from which a shape grammar to generate that particular façade is derived. Additionally, texturing information is extracted from the picture, and used to detail the models produced from the derived shape grammar.

Alternatively, the work of Bokeloh et al. [3] takes an existing model as its input example. The model is analyzed and subdivided into component parts, and a shape grammar is derived that re-uses the components of the example to produce new models in the same style as the example. An example of an input and associated outputs from their work is shown in Figure 2.13.

The problem of easily, and visually, editing shape grammars for the layman has also been addressed by Lipp et al. [29] in their work on interactively editing shape grammars for architecture. Their approach is primarily concerned with operating on the grid-like façades of buildings, but does feature methods for shaping the overall building structure. It works with the derivation tree of

[16]Product website: `http://www.esri.com/software/cityengine`

shapes that the shape grammar produces, and provides a GUI for shape grammar generation for users not familiar with the intricacies of formal grammars.

Recent work [2, 17] has also developed methods for automatically creating a structural skeleton for models generated by shape grammars. These methods are integrated into the shape grammar process, creating skeleton bones for the different parts of the model and linking them together with appropriate joints and constraints as the generative process occurs. The grammar itself can be extended to let the rules tweak these skeletons, allowing the user to control the structural skeleton's creation in parallel with the model creation itself. This means that structural skeletons do not need to be manually created by hand, saving time and effort. The created skeletons can then be used to create animations or run structural simulations on the generated models (by passing the bones, joints, and constraints to a physics engine which will simulate their interactions).

However, in spite of the large body of work described above, shape grammars do have limitations. In split grammars, shapes that span multiple subdivisions, and shape intersections, are difficult to handle gracefully. In addition, particularly unusual building designs with complex elements, such as tunnels and interior hollows, are very hard to generate.

2.3 Voxels

Voxel data is used in many different areas of computer science, such as computational astronomy, medical imaging, and fluid simulation. The range of uses are far too numerous to cover in this section, so we focus on their use in the area relevant to our work: procedural generation. The following subsections cover the issue of efficiently storing voxel data, and the two major areas in procedural generation where voxels are used.

2.3.1 Voxel Storage

Managing voxel data presents some challenges [21]. In spite of the widespread use of voxel data sets, managing, storing, and processing voxels can be challenging. For example, a relatively small 256 x 256 x 256 grid of integers requires 64 megabytes of storage, and a single integer per element cannot store much information. This illustrates one of the major problems with voxel data sets: storage. They tend to be quite large when stored naïvely, requiring approximately $O(N^3)$ for storage, where N is the average dimension size.

There are a number of options for storing voxel data sets, the most obvious being a 3D array. Although conceptually pleasing and simple, this is unsuitable for medium sized or larger data sets, due to the amount of space required.

Compression can be used to cut down on the space requirements, but this is only useful for secondary storage, since the data needs to be uncompressed in RAM for actual manipulation. Additionally, if the data set has a high entropy, then compression is ineffective at reducing the storage size.

Hierarchical tree data structures [47] are a much better choice for voxel data storage. In the context of voxel data sets, tree data structures can reduce the data size by recursively creating nodes down to only the level of granularity needed to accurately represent the data set. They also allow the aggregation

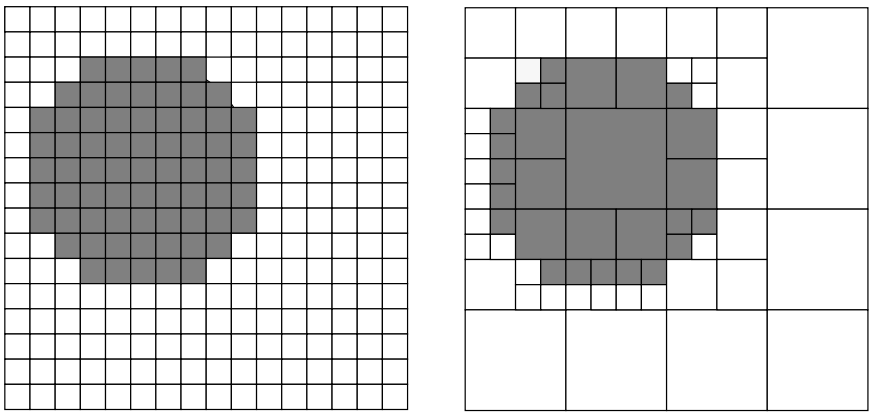


Figure 2.14: A 2D illustration of how hierarchical tree data structures can reduce the memory requirements for storing voxels. On the left, the voxel data set is stored as a 2D array, requiring that each of the 256 elements of the grid are explicitly stored. On the right, the same data set is stored in a quadtree (the 2D equivalent of an octree), where groups of voxels can be aggregated together if all of them are either solid, or empty. With the tree, only 67 elements need be stored, plus the tree overhead – 26.1% of the space required for the array. This generalizes to three dimensions, and in most cases storing the data set in a tree data structure will require much less space than using an array.

of adjacent elements that are the same, since these elements can all be stored as one aggregate parent node, rather than multiple individual nodes. This is illustrated in Figure 2.14.

An alternative to regular tree data structures are *pointerless trees*. This is a way of storing a tree without the use of pointers, that operates on the same principles as a 1D array being used to encode a binary heap. This has the benefit of avoiding pointer overhead, but these techniques are more complex to implement than pointer-based data structures, and for most voxel-storage situations, the small performance benefit they bring is not worth the added implementation complexity.

There are four main types of trees which can be used for storing voxel data sets: octrees, point region octrees (PR-octrees), kd-trees, and R-trees. Here we will examine how they work, their strengths, and their weaknesses. Further details on these, and other methods of voxel data set storage, can be found in the literature [10, 47].

The best-known example of a hierarchical tree data structure for space subdivision is the *octree* [47]. Octrees work by recursively subdividing 3D space into 8 equal size octants, with each octant becoming a child node of the parent node that spatially contains all the octants. Figure 2.15 shows how an octree operates.

Octrees have many applications, from collision detection [13] to compression [4], and are a widely used data structure in 3D graphics applications. They also have the benefit of allowing for spatial indexing.

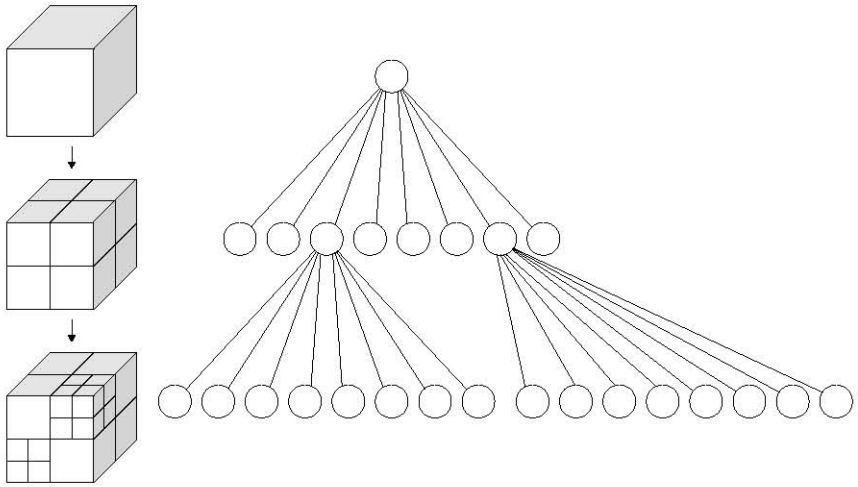


Figure 2.15: An example of an octree being used to divide up a cuboid space. The root node is partitioned into eight octants, but then only the two octants that require further subdivision are subdivided.

Spatial indexing is a feature of some data structures that store spatial information. These data structures can be used to efficiently query what objects are in a region of space and the distances between objects. Spatial indexing is often accomplished by subdividing the space into a tree structure and then walking the tree in some fashion to find the desired information.

Spatial indexing is important because it allows the efficient use of common operations such as nearest neighbour finding, collision detection, and the like, on spatial data sets (maps, virtual environments, simulations, etc.). Without it, these operations would be significantly more computationally intense. In the context of storing voxel data though, spatial indexing is of little use, since the benefits it brings are typically not needed.

A *point-region octree* (PR-octree) is a modification to the standard octree, where the splitting point is not always at the centre of the space contained by the parent node. Instead, it can be located anywhere in the parent's space, and its location is stored by the node. See Figure 2.16 for a PR-octree example.

The benefit of the PR-octree is that it allows the tree to be better balanced. During the creation stage, the splitting points can be chosen to evenly divide the contents of the tree at each level.

Kd-trees can be considered a special case of a binary space partitioning tree (BSP) tree. A BSP tree is a type of binary tree used to hierarchically divide up a space. It operates by recursively splitting the space on either side of arbitrarily aligned planes.

BSP trees are used to accelerate rendering of scenes, because the tree of a scene can be precomputed and then that tree used to accelerate the painter's algorithm. BSP trees are not suitable for storing voxel data, however, since non-axis-aligned splits cause aliasing issues.

Kd-trees differ from BSP trees in that their spatial subdivisions are all axis-

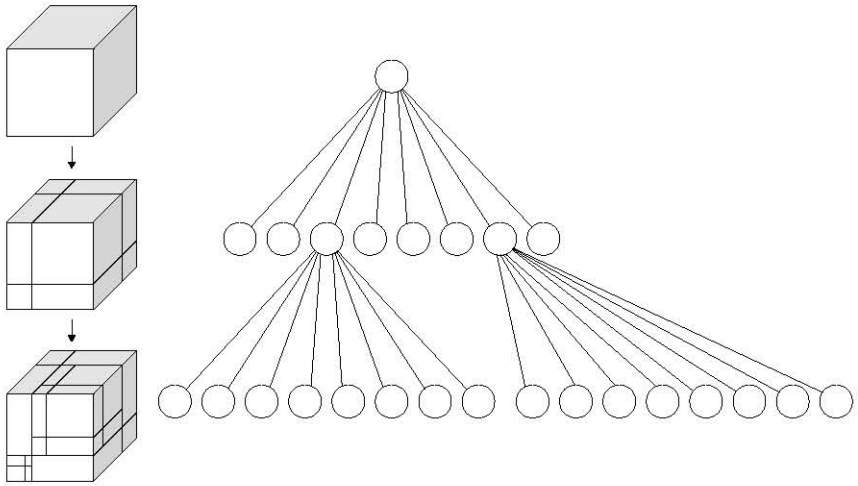


Figure 2.16: An example of a point-region octree. Note that the splitting point for each subdivision is not centered in the space to be subdivided, as it is in a normal octree. Instead it is stored in the relevant node of the tree.

aligned. The two subdivisions are not necessarily of equal size though. The axis to which the split is aligned to can either be a function of node depth (for example, alternating X, Y and Z) or stored explicitly (which is useful for keeping the tree balanced during creation). Unlike BSP trees, kd-trees are typically used for storing data, and not for rendering it. A simple kd-tree example is shown in Figure 2.17.

Because they are binary, less information needs to be stored at each kd-tree node than in trees with more children per node, but traversals will be longer, since the tree is deeper.

Finally, *R-trees* are the spatial equivalent of B-trees. Each node is the minimum bounding rectangle that contains N children rectangles, up to a maximum of M. The leaves of the tree may contain a list of the points and shapes that the tree actually stores, assuming the tree does not just store rectangular regions. A diagram of a R-tree is presented in Figure 2.18.

Bounding regions at the same level of the tree may overlap, but each child node only ever has one parent node. R-tree nodes can also be split when they have too many children. This is done in a manner similar to B-trees, and allows the tree to be balanced.

The major benefit that R-trees bring is that they are very efficient at spatial queries, such as finding the ten closest items of a certain type near an arbitrary point (when all the items are stored in the tree).

Any of the above tree types can be used to store voxel data sets, and each type brings different benefits, which may be useful for application-specific uses. Discussion of the best type of tree for our purposes is deferred to Section 3.3.1.

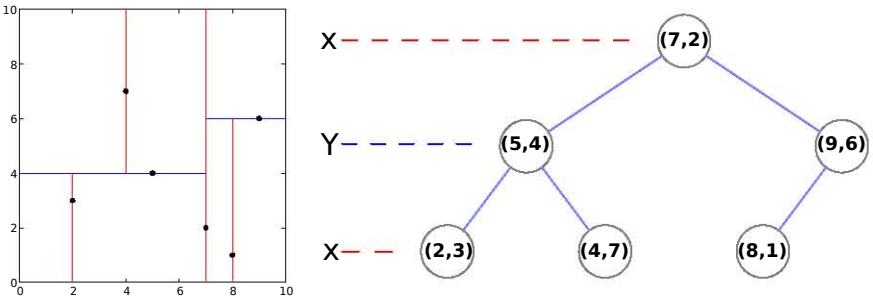


Figure 2.17: An example of a kd-tree, used to hierarchically store a collection of points in a square 2D space. Every node of the tree contains the coordinates of one point (not just the leaf nodes), and during creation, the points are chosen and inserted into the tree so as to keep it balanced. This kd-tree alternates between splitting along the X and Y axes.

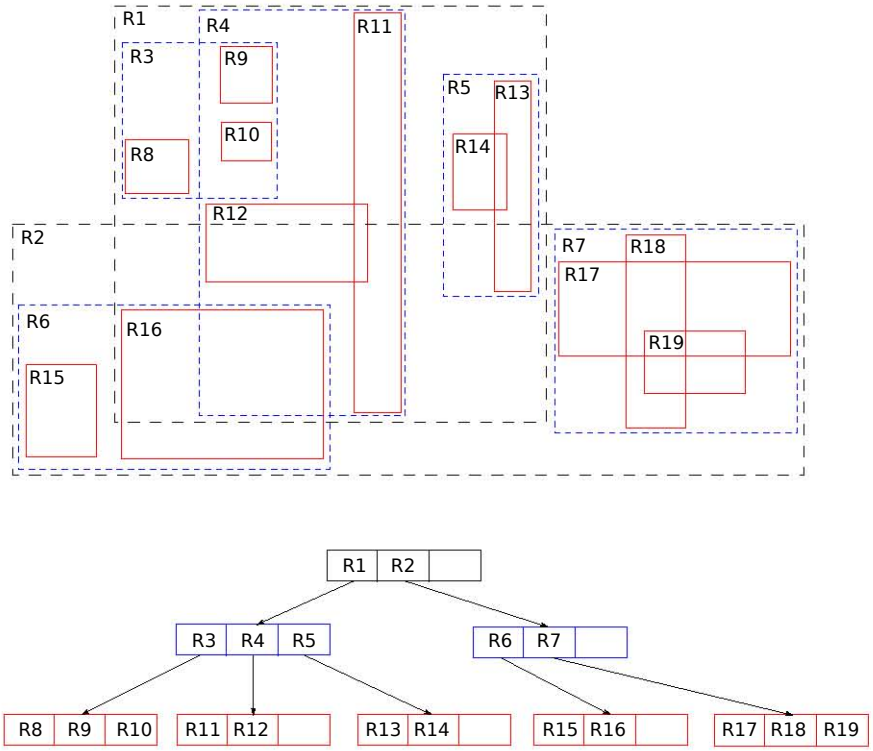


Figure 2.18: An example of a R-tree, in a continuous space. Each node can have a maximum of three rectangles within it, and will be split into two nodes should more be added. Each node's bounding rectangle completely encompasses the bounding rectangles of its children, for example, R2 encompasses R6 and R7. Bounding rectangles can also overlap, such as R1 and R2.

Figure 2.19: A screenshot showing fully 3D voxel terrain created by the independent video game, Minecraft. ©2011 Mojang AB.

2.3.2 Terrain Representation

Voxels have seen sporadic use in procedural generation. Most commonly, they are used to represent terrain (often procedurally generated) [10]. In this regard they are superior to 2.5D heightmaps because they allow the specification of fully 3D terrain, with elements such as overhangs and tunnels that are impossible to represent with a heightmap. One of the best-known games to use a voxel terrain representation is Crysis[17].

However, voxel terrain does have higher performance requirements and increased complexity, since the change from 2D to 3D significantly increases the number of elements to be stored and processed ($O(N^3)$ instead of $O(N^2)$). For this reason, 2.5D heightmaps are used far more commonly than voxel terrain in games.

One of the foremost examples of voxels used for procedural generation is the independent video game Minecraft[18]. The entire world is represented with voxels and the landscape, rivers, caves, and buildings are all procedurally generated by the game, using a multi-pass algorithm that makes heavy use of 3D Perlin noise. An example of terrain from the game is shown in Figure 2.19.

Unfortunately, due to the proprietary nature of the game, the full details of the algorithm used are not in the public domain. What is known is that the game is 'chunk', or block based, with the world being generated in chunks that are seamlessly joined together. Only the chunks near a player are generated and loaded into memory at any point. The world is made up of different adjacent 'biomes' (plains, forest, mountain, etc.) which are created using different generation algorithms, but blend into each other. The currently-loaded part of the world is also stored in some space efficient data structure, most likely an Octree.

[17]Game website: `www.ea.com/crysis/`
[18]Game website: `www.minecraft.net`

When voxel data is used for applications such as landscapes, it is often necessary that it be converted to a format suitable for rendering. Although 3D grids can be rendered to an image using methods such as ray casting [27], these are often too computationally intensive for real-time use. A more common option is to convert the voxels into a mesh, using either the Marching Cubes algorithm [30], or by simply converting each voxel into a mesh cube (as is done in Minecraft).

2.3.3 3D Texture Generation

Since a 3D texture is effectively a voxel grid, voxels methods are also used in procedurally generating 3D textures. These are generally extensions of 2D texture generation methods converted to work in 3D [23].

These texture synthesis methods have also been extended to create 3D models. Merell's algorithm [33] works by assigning a cuboid section of geometry to each voxel type, and then keeping a record of how these cubes of geometry can be placed adjacent to each other while keeping the resulting model consistent. The sections of geometry must be created such that each piece can seamlessly blend into a few of the other sections. If a pre-created section cannot join to at least one other section, it can lead to problems such as visual inconsistencies.

Texture synthesis methods are then employed to create a, potentially infinite, 3D grid of cuboid sections of geometry that are joined so as to be consistent (the pieces of geometry join seamlessly so that the edges of the sections are not apparent). A downside to this approach is that it requires the manually created cuboid sections of geometry.

This approach differs from our voxel-space shape grammar algorithm in several respects. Firstly, there is much less control over the output in Merell's algorithm. General guidelines and high level meta-shapes can be defined, but medium to low-level control of the output is not possible. Secondly, it requires pre-created pieces of geometry to make up the created model, whereas shape grammars do not. Thirdly, Merell's method is aimed at a different scale to ours. His is intended to produce large regions uniformly covered in similar range of pieces of geometry. In contrast, our algorithm is aimed at creating a single standalone model.

2.4 Summary

In this chapter we have examined and discussed the background information relevant to our research. This includes procedural generation as a wider field, shape grammars and their myriad of extensions, and the use of voxels in procedural generation.

Although there are large bodies of previous work on shape grammars and voxels that are relevant to us, there is little crossover between shape grammars and voxels in the literature, and certainly nothing similar to the algorithm that we are proposing. This shows that our work is, to the best of our knowledge, novel.

There are however, many aspects from the previous work that we shall build upon in the remainder of this work, and incorporate into our design for voxel-space shape grammars. For example: Kolmogorov compression, storing voxels

in space efficient data structures, and many of the shape grammar extensions created by others.

Chapter 3

A Framework for Voxel-Space Shape Grammars

The core novel contribution of this thesis is an extension to shape grammars, where they are interpreted in a voxel-space. We have developed a framework for using this extension, and the features it provides. In this chapter we discuss the design and implementation of the framework, and suggest avenues for improvement and optimization.

The framework that we developed in this work was intended to be a proof-of-concept implementation of the shape grammar extensions that we developed. These extensions are aimed at allowing the easy use of Boolean geometry operations and sub-shape rule-based detailing in shape grammars, as outlined in chapter 1.

Our algorithm builds on, and includes, some of the techniques discussed in the previous chapter, but also features a number of novel ideas that we have developed. It operates in multiple stages, and makes use of an additional input that conventional shape grammars do not have, but this is necessary to control the added complexity our extensions introduce. The rest of this chapter covers the different stages of the algorithm, from both design and implementation perspectives, and shows how our extensions are able to increase the generative range of shape grammars.

The layout of this chapter is as follows: Section 3.1 provides a high-level outline of the process and Sections 3.2 to 3.6 discuss each of the stages of the algorithm in detail. Section 3.7 discusses implementation details and potential optimizations, and finally, Section 3.8 summarizes the chapter.

3.1 Framework Overview

The process of interpreting a shape grammar to produce a model in our framework consists of five stages (one of which is optional), and requires two inputs from the user. The final output is a textured mesh, suitable for use in modern graphics applications. See figure 3.1 for a diagrammatic overview of this

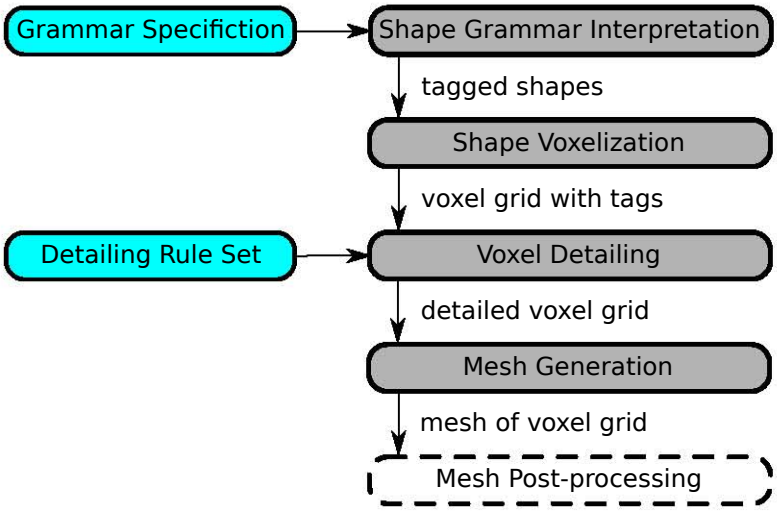


Figure 3.1: A flow chart of our algorithm. The cyan boxes are the user inputs, and the grey boxes are the four stages of the algorithm. Arrow labels indicate the output of each stage. Mesh post-processing is an optional final step.

process.

The two user inputs to the algorithm are:

- **Shape Grammar Specification:** The shape grammar specification consists of three items: the axiom, which is the symbol that the grammar begins with and any metadata associated with it; the set of productions, or rules, of the grammar which transform the shapes; and other parameters for the shape grammar itself, such as the maximum number of iterations allowed. This collection of items is what is needed to run the shape grammar to completion and produce a model. These items are discussed further in Section 3.2.1.
- **Detailing Rule Set:** The collection of rules that are used to assign visual detailing information to the voxel grid, and any global parameters that go with them. The information generated from this rule set is used when displaying the created model. The specification of this rule set is similar to the shape grammar specification, but does not have an axiom. Further discussion on this input is found in Section 3.4.

The five stages of the algorithm are:

1. **Shape Grammar Interpretation:** The input grammar is run to produce a collection of shapes, with associated metadata tags. This step is very similar to how a conventional shape grammar would run, and produces an exact specification of shapes that is passed onto the next stage. Shape grammar enhancements (such as stochastic rules and context sensitivity) can be used during this interpretation process.
2. **Shape Voxelization:** The shapes from the previous step are voxelized, and any metadata tags from the shapes are inherited by the voxels. Be-

cause voxel representations of the shapes are used, Boolean geometry operations can easily be applied during this step. The final product of this stage is a voxel grid representing the shape grammar's output from stage 1.

3. **Voxel Detailing:** Surface voxels in the grid are assigned visual detailing information by the detailing rule set. The rules make use of the metadata tags assigned by the shape grammar to get contextual information about the shapes, and give each voxel a set of detailing tags which can be used to provide visual detail when displaying the model.

4. **Mesh Generation:** A mesh representation of the voxel shape is produced, using the marching cubes algorithm. Only the surface voxels need to be operated on to produce the mesh, so other voxels are ignored. Detailing information from the previous step is assigned to the generated mesh.

5. **Mesh Post-processing:** The generated mesh is smoothed and refined. This step is optional, but improves the mesh's visual quality if performed. Any of a large number of mesh smoothing algorithms can be used.

We cover each of the five stages in detail, explain their operation, what inputs they use, and what outputs they generate.

3.2 Shape Grammar Interpretation

The first stage of the algorithm requires that we specify a shape grammar and then run it to produce the output set of shapes.

This step is very similar to running a conventional shape grammar, with some minor differences. A set of shape grammar rules, and an axiom shape, are provided by the user. This rule set is then run on the axiom, producing new shapes and modifying existing shapes on each iteration of the rules. This continually-updated set of shapes (the current shape set) converges to the final output of the shape grammar. A simple example of this stage is shown in figure 3.2, and pseudocode for this stage of the generative process is found in algorithm 3.1.

Figure 3.2 shows a simplified shape grammar example, which builds a simple two-dimensional house. The four rules, and an image showing the derivation process are included. The grammar is interpreted in parallel (as many shapes as possible are operated on in each iteration), and the symmetric copies of shapes are added after interpretation is complete. A symbolic representation of the grammar's output after each iteration (with coordinates and other shape data excluded for ease of understanding) is shown in Figure 3.3.

3.2.1 Grammar Interpretation Details

A full specification of the shape grammar requires three components: the grammar's axiom, the grammar's rules/productions, and the parameters necessary to run the grammar. These three items are specified by the user, and passed to the shape grammar interpreter.

Axiom: building_base

rule 0: building_base **->** roof (shape: triangle) **|** tower (shape: rectangle; symmetry: reflective) **|** window (shape: square; symmetry: reflective) **|** door (shape: square)

rule 1: roof **->** roof | chimney (shape: rectangle)

rule 2: tower **->** tower | tower_peak (shape: triangle) **|** tower_window (shape: rectangle)

rule 3: window **->** window | window_arch (shape: triangle)

rule 4: door **->** door | door_arch (shape: circle)

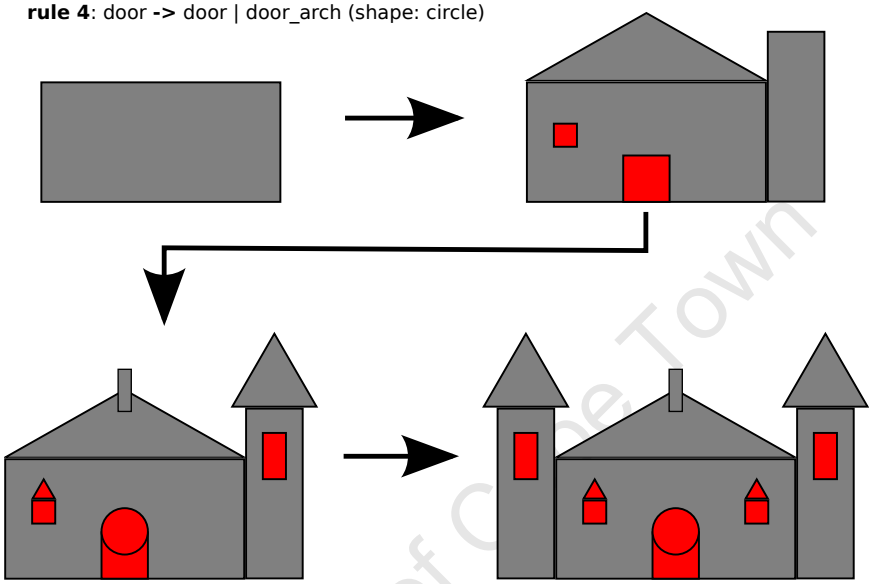


Figure 3.2: A simple shape grammar being interpreted in parallel to form a basic building. Grey shapes are additive, and red ones are subtractive. Position and size information in the example grammar are not shown, for the sake of simplicity. Note that the final step involves the symmetric copies being created, as the grammar only requires two iterations to complete (after two iterations, all shapes in the current shape set are terminal).

The axiom is simple enough to specify, but for certain grammars, associated metadata tags might also need to be explicitly stated, for example a castle generating grammar that begins with the keep as a rectangle would need tags for "material:stone" to be specified by the user.

The exact specification of the grammar rules depends on what shape grammar extensions and operations are supported by the implementation. They should follow a well-defined syntax that can be parsed to ensure validity and their order of priority should be indicated to avoid a situation where one of two rules could be used, and the interpreter is unable to choose which. In the literature, the most common method of indicating rule priorities is to have the user explicitly assign each rule an integer priority [34].

The parameters for running the grammar are global pieces of data that the interpreter requires to correctly interpret the grammar. The most obvious of these is the maximum number of iterations the grammar is allowed to run for. Other items are likely to be implementation specific and related to shape

Algorithm 3.1 Shape Grammar Interpretation

```
currShapeSet ← {Axiom}
iterations ← 0
MaxIterations ← getMaxIterations()
while iterations < MaxIterations and currShapeSet.hasNonTerminals()
do
  if parallelExecution = TRUE then
    for all i ∈ currShapeSet do
      i ← doRuleDerivation(i)
    end for
  else
    a ← getFirstNonTerminal(currShapeSet)
    a ← doRuleDerivation(a)
  end if
  iterations ← iterations + 1
end while
for all i ∈ currShapeSet do
  if i.hasSymmetry() then
    s ← i.createSymmetricCopies()
    currShapeSet.insert(s, i − 1)
  end if
end for
return currShapeSet
```



```
Iteration 0: { building_base }

Iteration 1: { roof | tower (right) | window (left) | door }

Iteration 2: { roof | chimney | tower (right) | tower_peak
    (right) | tower_window (right) | window (left) | window_arch
    (left) | door | door_arch }

After Symmetric Copy Creation: { roof | chimney | tower (right)
    | tower_peak (right) | tower_window (right) | tower (left) |
    tower_peak (left) | tower_window (left) | window (left) |
    window_arch (left) | window (right) | window_arch (right) |
    door | door_arch }
```

Figure 3.3: A symbolic representation of the output from the shape grammar in Figure 3.2 across three iterations of interpretation. The final output, after symmetric copy creation is also shown. Associated coordinates and other metadata is excluded for the sake of simplicity.

grammar extensions. For example, in a context-sensitive shape grammar that generates houses, the average sun direction would be included here, so that the rules could query that information and change aspects of the house based on it.

There are no special format requirements for how these three items are specified. In our implementation they are specified in a text file, using a custom syntax, but other implementations could use their own formats.

The process of running the grammar will terminate under one of two conditions: Either after a user-defined maximum number of iterations have been run, or once all shapes in the current shape set are terminal and hence none of the rules in the rule set can be applied to any of the shapes.

Fractal models, and other shape grammars which will not terminate naturally, such as spirals, require that a maximum number of iterations be specified or they will run forever. However, most well-designed shape grammars to create buildings and architecture should be created such that they will eventually consist of nothing but terminal shapes, and hence finish running naturally.

Rules can be run in parallel (as done in L-systems), or serially (as is done in traditional formal grammars). These are appropriate in different situations, depending on the type of model being generated by the shape grammar. Serial rule derivation is suitable for most situations, such as buildings, vehicles, and other structures. Parallel rule derivation is needed for models that have fractal qualities, such as trees and 3D fractal objects, because parallel derivation at decreasing scales is perfect for creating fractal structures [44]. In our implementation, the user can choose between parallel and serial derivation for each different shape grammar individually.

3.2.2 Supported Shapes

The selection of shapes available to the grammar is dependent on the implementation. The possibilities range from only simple shapes, such as rectangles, cylinder, etc. to complex parametric shapes that can be defined during interpretation, such as volumes or rotation, user-defined splines, and arbitrary meshes loaded from file.

Although the more complex shapes can increase generative range, they are more difficult to handle, and users may struggle to work with them through the medium of shape grammar rules, due to their inherent complexity, and the difficulty of visualizing the results of adjusting their parameters.

At a minimum, the interpreter should be able to handle simple shapes such as cubes, rectangles, prisms, ellipsoids, and cylinders. These are much easier for users to visualize and work with when designing shape grammars, and are still sufficient to create reasonably complex models, especially when combined using Boolean geometry operations.

3.2.3 Grammar Enhancements and Extensions

Any of the many enhancements and extensions to grammar generation methods can be used here: environmental sensitivity [39], stochastic rules [7], a derivation tree for querying earlier states during the rules' execution [34], and more. Our implementation includes these three extensions, as well as split grammar operators [56].

Environmental sensitivity allows a rule to query the environment around the shape it is operating on and use that information to modify its output. This is achieved in our implementation by making the full current shape set accessible to the rules (but read-only). Because the rules are written in Python, arbitrary queries about the current shape set can easily be specified, and the rules written to change their outputs based on the results of the queries.

In addition, the use of a full programming language to specify rules allows a great deal of expressive power. Complex environmental queries can be constructed more easily than with a constrained rule grammar. For example, should a user wish a rule to query whether a shape lies in the shadow of another shape cast by a light source that changes position on different iterations, this can be coded using the expressive power of Python, which is a full programming language. However, constructing such a complex query in a limited grammar syntax would be problematic, perhaps even impossible. For this reason, we opted to have our shape grammar rules be specified in the Python programming language.

Stochastic rules mean that the rules in the shape grammar have access to randomness, and can use it to make random decisions about rule outputs and shape metadata. Our implementation achieves this by making the built-in 'random' Python class available to the rules directly. Rules can call the class's methods to easily obtain numbers from a wide range of floating point and integer random distributions.

Stochastic rules are particularly important for their ability to create multiple models from a single shape grammar. Instead of hard coding a parameter, the user can indicate that a random value from a range of numbers should be chosen each time the rule is run. From this, a single shape grammar can produce different models across multiple runs, where all the generated models will share a consistent theme and style (assuming the grammar is reasonably well-designed).

For example, when creating a large number of cars to populate a virtual city, the dimensions of the cars could be randomly decided by choosing values for their length, height and width from a uniform distribution. This is an extremely useful extension that allows the creation of many models with minimal work. For actual examples of stochastic shape grammars, refer to appendix B, where three such grammars from our implementation are included.

Our implementation also stores a derivation tree of the grammar's interpretation. This is a tree data structure, with depth equal to the current number of iterations, where each layer of the tree represents the current shape set at the iteration equal to the depth of that layer. For example, the root of the tree is the axiom, the root's children are the shapes derived from the axiom on the first iteration, those node's children are the shapes derived from them on the second iteration of interpretation, and so on.

The benefit of storing this tree is that it can be used to query previous states of the current shape set. This is useful in certain situations. For example, when creating the façade of a building, a rule may wish to query the original dimensions of one side of the building, but that side of the building has been split up into multiple façade elements. It would be wasteful and inefficient to calculate the original dimensions from those façade elements, so instead the rule can use the derivation tree to query an earlier version of the current shape set, where the building's façade was still one rectangle.

Splitting operators allow rules to access the power of split grammars. They are extremely useful and well-suited to creating buildings. Our interpreter has a parametrized split operator that can be used to split a shape into a number of smaller shapes, along any of the three Cartesian axes.

Our implementation also includes a collection of standard utility functions for common operations, such as scaling, translation, rotation, and hollowing shapes. All that is required from this stage of the generation process is a specification of the final set of shapes, so virtually any extension should be usable here.

There are two other grammar extensions that we found most useful for generating models during our experimentation, which we discuss below.

Shape Tagging

The first of these extensions is the tagging of shapes with metadata. One of the operations that the shape grammar rules can perform is to add metadata tags to shapes in the current shape set. We implement these as arbitrary strings. These serve to preserve additional information about the shapes that can be used in later stages of the generation algorithm. For example, when generating a castle, a cylinder (and its children if recursive tagging is used) could be tagged with "type:tower" and "material:stone".

These metadata tags indicate additional properties of the shape that significantly enhance later detailing and texturing. This makes detailing the voxels significantly easier, because there is no extra overhead needed to determine what the properties of a voxel are; we can just examine the tags to know what properties the voxel should have.

Additionally tagging is important for distinguishing which voxels came from which shapes, since shape boundaries are lost in the conversion to voxels. By giving a unique identifier tag to each shape output from the shape grammar, we can examine a voxel in the grid and know exactly which shapes from the original grammar it is derived from. This means we do not lose knowledge of the original shapes that created a voxel grid. Additionally, this is useful for visual debugging of a shape grammar, since each shape can be assigned a unique colour. This lets the user examine how the shapes are interacting with each other, making it easier to see problems in the grammar.

Tagging of shapes is not new, and has been used in other shape grammar implementations [34]. However, previous uses have used tags as part of the shape grammar derivation process, indicating whether shapes were active or inactive, their overlap with other shapes, and the like. Our usage of tags to store metadata information is novel in that they are used outside the shape grammar derivation process, in our voxel detailing scheme (see Section 3.4).

Symmetry Specification

The second shape grammar extension is our support for the specification of symmetry in the grammar rules. Our grammar implementation has special operators for indicating that a shape, or group of shapes, (and any child shapes that derive from them) should be cloned to create symmetrical versions.

Symmetry operators are an important element in shape grammars, because symmetry is very common, with cars, buildings, ships, spacecraft, and many

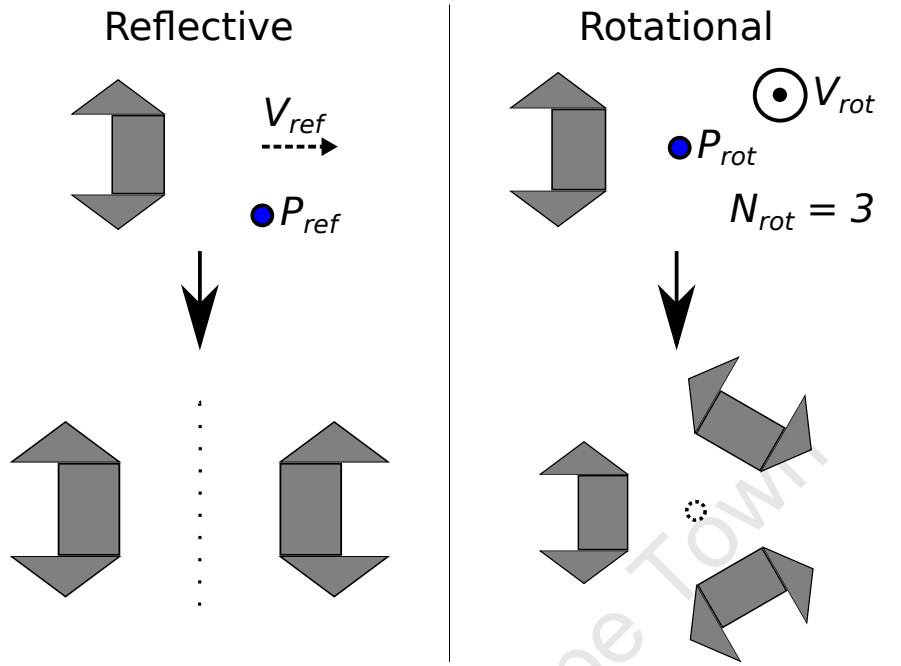


Figure 3.4: An example of the two types of symmetry, used in our shape grammar. Note that in the rotational example, V_{rot} is pointing out of the page.

other structures featuring it. Manually creating symmetric copies of objects is inefficient and tedious, which is why symmetry operators are so useful. They allow the shape grammar designer to easily create symmetry, saving large amounts of design time.

The topic of symmetry and shape grammars has seen much discussion in the literature [3, 35]. However, this discussion has focused on analyzing symmetry in existing models or image inputs, in order to create new models based off them. There has been very little discussion on how to implement the specification of symmetry in shape grammar rules.

As a result, there is no universally accepted method of specifying symmetry information in shape grammars, and we have designed our own approach to declaring symmetry for our implementation of our algorithm. Using our own approach also allowed us to ensure that our handling of symmetry would not cause problems with the rest of our algorithm.

We support two types of symmetry: rotational and reflective. Figure 3.4 shows an example of how our symmetry operators work. In rotational symmetry, three arguments are provided to the operator, from which positioning information for the symmetrical copies can be derived:

P_{rot} The center point around which the symmetric branches are rotated.

V_{rot} A vector normal to the plane of rotation.

N_{rot} The number of rotational copies to create.

For reflective symmetry, only two arguments are required to fully construct the mirror copies of the shape, or group of shapes, to be reflectively copied:

P_{ref} A point on the plane of reflection.

V_{ref} A normal to the plane of reflection.

These symmetry-specifying points and vectors can either be specified in global coordinates, or relative to the local coordinate frame of the symmetric object, whichever the user prefers.

The symmetry information is set by shape grammar rules while the grammar is run, but the actual symmetric copies are only added to the current shape set once the grammar rules are finished running.

This is done as a post-process because further shapes could be added to the set of shapes undergoing symmetry, in iterations after the symmetry is specified. Rather than tracking the symmetric copies and updating each of them for every change to the original set of shapes, we simply flag the set of shapes for symmetry and wait until the rule derivation is finished before creating the symmetric copies, which is more efficient.

One potential issue with this approach of adding symmetric copies after the grammar has run is that it can break context sensitivity. If a shape grammar rule adds a shape in a spot that will have a symmetric copy created in it, and that rule makes queries about its neighbourhood, the information it received will no longer be accurate once those symmetric copies are created.

For example, consider a spaceship's wing being created using a reflective symmetry rule, and subsequently, a series of windows, which make context sensitivity queries, are placed on the other side of the spaceship where the wing's reflective copy will be placed. The queries will indicate that the space is not occupied, and the windows will be created accordingly. However, later, when the reflective wing copy is created, that previously empty space will be occupied, and the windows will overlap with the wing.

This problem can be solved by creating symmetric copies in synchronization with the original, but this raises the efficiency problems described above, meaning that having context-sensitivity queries that are guaranteed to be correct comes with a small performance penalty.

Our implementation uses the first approach, and creates the symmetric copies of the original shape after the shape grammar has finished being derived. This is simpler and more efficient, and although context-sensitivity queries could be incorrect, we did not experience any issues during our testing.

3.2.4 Output

Once the shape grammar interpretation has finished, a full specification of the final output set of shapes is passed to the next stage of the algorithm. This includes positions, dimensions, orientations, tags, and any other relevant information. The exact format is obviously implementation dependent, but what is most important is that it unambiguously describes the output in a format suitable for further use.

An example output from our implementation, of our spiral model (see Figure 4.14), is shown in Figure 3.5.

```
name: rectangle active: True additive: True position: 0.0 0.0
    0.0 extents: 5.0 10.0 5.0 orientation: -1.0 0 0.0 0.0 -1.0
    0.0 0.0 0.0 1.0 priority: 0 tags_begin: spiral tags_end #
name: rectangle active: True additive: True position: -6.75
    -17.5 5.0 extents: 5.0 10.0 5.0 orientation: -0.734 -0.678
    0.0 0.678 -0.734 0.0 0.0 0.0 1.0 priority: 0 tags_begin:
    spiral tags_end #
name: rectangle active: True additive: True position: -23.585
    -25.768 10.0 extents: 5.0 10.0 5.0 orientation: -0.115
    -0.993 0.0 0.993 -0.11 0.0 0.0 0.0 1.0 priority: 0
    tags_begin: spiral tags_end #
name: rectangle active: True additive: True position: -41.748
    -21.086 15.0 extents: 5.0 10.0 5.0 orientation: 0.529 -0.848
    0.0 0.848 0.529 0.0 0.0 0.0 1.0 priority: 0 tags_begin:
    spiral tags_end #
name: spiral_segment active: True additive: True position:
    -53.027 -6.099 20.0 extents: 5.0 10.0 5.0 orientation: 0.931
    -0.364 0.0 0.364 0.931 0.0 0.0 0.0 1.0 priority: 0
    tags_begin: spiral tags_end
```

Figure 3.5: An example of a shape grammar output from our implementation. This output corresponds to the 'spiral' model shown in Figure 4.14.

The format is keyword variable names, followed by their value. For example, 'position:' indicates that the following three values are the x, y, and z coordinates of the shape's position, and 'tags_begin:' indicates that all whitespace separated strings that follow are metadata tags associated with the shape, until the 'tags_end' string is encountered. A # character indicates the end of an individual shape's specification. This format was designed to be human readable, but also easily parse-able by our implementation.

This final, and unambiguous, shape specification is all that is required from this stage. In fact, with some minor modifications, it should be entirely possible to use any other shape grammar implementation for this stage, and pass its output to the next stage of the algorithm. Certain minor features, such as shape priority (discussed in the next section), would need to be added to the other shape grammar implementation, but no major changes would be required.

3.3 Shape Voxelization

In this phase, the shapes output from the shape grammar are voxelized into a voxel grid. This process is analogous to rasterizing vector graphics into a pixel image format. An example of this voxelization process is shown in Figure 3.6, with pseudocode in Algorithm 3.2.

Hence, we go from working with the collection of shapes from the shape grammar stage to working with a voxel grid containing those shapes.

In many cases it will be necessary to scale the shapes output from the grammar into the voxel grid's space. In our implementation, the grammar could generate shapes at any scale and coordinates, and the shapes were then automatically scaled into the coordinates of the voxel grid (which is a cuboid shape with a size that is a power of 2). This allows grammar designers to not be limited to a certain scale or coordinate range, while still ensuring that all shapes

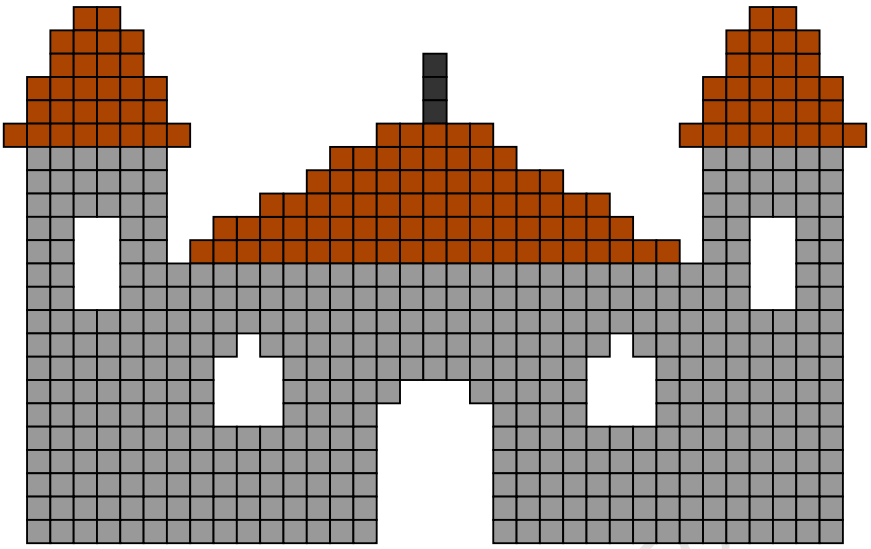


Figure 3.6: The output of the simple building shape grammar from figure 3.2, after being voxelized. The colours of the voxels correspond to the tags they inherited from shapes. Grey indicates 'material:wall', brown indicates 'material:roof', and the dark grey 'material:chimney'.

generated by the grammar are voxelized.

In the following subsections, we look at the practical issues around shape voxelization, and present our solutions to the problems and other recommendations.

3.3.1 Voxel Storage

As mentioned in Section 2.3.1, there are number of tree data structures that can be used to store voxel data sets, each of which has different properties. Because we are expecting to deal with large, complex voxel data sets, it is a requirement that we store the data sets in space-efficient tree data structures suited to that

Algorithm 3.2 Shape Voxelization

```
shapes ← getShapeGrammarOutput()
shapes ← sortByPriority(shapes)
shapes_bounding_box ← getBoundingBoxOfAllShapes(shapes)
gridResolution ← getVoxelGridResolution()
voxelGrid ← initializeEmptyGrid(gridResolution)
for all i ∈ shapes do
  i ← scaleShape(i, gridResolution, shapes_bounding_box)
end for
for all i ∈ shapes do
  voxelGrid.voxelizeShape(i)
end for
return voxelGrid
```

Criteria		Octree	PR-octree	Kd-tree	R-tree
Memory Efficiency	Voxel Aggregation	3	3	3	2
	Balance Guarantees	1	3	3	3
Performance	Data Analysis Requirements	3	2	2	1
	Efficient Lookup/Modification	2	2	2	2
	Re-balancing Requirements	3	1	1	1
Ease of implementation		3	2	2	1
Total		15	13	13	10

Table 3.1: Taxonomy of how the different types of tree meet our requirements for storing voxel data. Each item is rated on a scale of 1 to 3, where 1 is worst and 3 is best. Voxel aggregation indicates how well the tree type supports multiple adjacent voxels being aggregated together to save memory. The second row, Balance Guarantees, indicates the tree's ability to ensure that it is balanced, preventing degenerate cases from consuming more memory than necessary and guaranteeing logarithmic lookup times. In the performance category, Data Analysis Requirements shows indicates to what degree the data set must be analyzed during tree creation. Efficient Lookup/Modification indicates, on average, how fast we can expect lookups and editing of individual voxels to be (this includes consideration of periodic re-balancing where necessary). Re-balancing Requirements shows how much periodic re-balancing of the tree is necessary. The Total row shows the cumulative scores of the different tree types, where higher is better.

purpose. Using a naïve method, such as an array in an implementation of our algorithm would require infeasible amounts of memory.

When evaluating which type of tree is best suited to our purposes, there are three important factors to consider: memory efficiency, performance, and ease of implementation. A taxonomy of the options is shown in Table 3.1.

Memory efficiency is the entire reason for choosing a hierarchical tree data structure, and is the most important factor. Performance is less important, since our algorithm is not required to be real-time, though it must still operate at an acceptable level (a low-power polynomial-time algorithm). Finally, ease of implementation had to be considered due to the limited time frame of this project.

In terms of memory efficiency, all four tree types are at a similar level. They all allow for adjacent voxel elements of the same type to be aggregated into one parent node, which is where the main memory gains are made. R-trees perform worse than the other tree types on this front, since their design is focused more

on balancing their child nodes than efficiently aggregating, which is why they are ranked behind the other tree types.

PR-octrees, kd-trees and R-trees can all adjust the position of their dividing point/plane, which allows more efficient aggregation of voxels, theoretically requiring less memory than an octree. These extra savings are not an order-of-magnitude improvement though, and come at the cost of significant added complexity. Additionally, these extra gains rely on analyzing the data to be stored, which is an expensive operation that would be hugely inefficient for every update to the data set.

Note that in the table, the "Balance Guarantees" row in the memory efficiency section indicates only to what extent the tree is capable of guaranteeing balance. The performance impact of this balancing is compared in the performance section of the table, with "Data Analysis Requirements" indicating how much pre-processing analysis is necessary before creating the tree, and the "Re-balancing Requirements" row indicating how computationally expensive the periodic re-balancing is. Both of these performance factors are discussed below.

In theory, the computational performance of the different tree types is quite similar, with all operations being on the same order of magnitude asymptotically. However, as we can see in the table, there after differentiating factors in practice.

As shown in the "Data Analysis Requirements" row, tree creation is slightly quicker for the octree, because it does not require analysis of the data to ensure balance. The R-tree is ranked worst because it requires more complex pre-processing analysis than a PR-octree and kd-tree do.

Lookup and modification is logarithmic across the different tree types, and this is reflected in the table row labelled "Efficient Lookup/Modification", where all trees have the same rating. No trees have any feature to differentiate each other in this category.

However, the PR-octree, kd-tree and R-tree all require periodic re-balancing if the voxel data set changes significantly. Re-balancing is an expensive operation, and since we are going to be working with data sets that change frequently due to continual editing and addition of new shapes, we consider the performance impact of re-balancing in the table row titled "Re-balancing Requirements". The octree wins in this category, as it does not require any balancing operations. The other tree types all require periodic re-balancing to maintain their efficient lookup guarantees.

The final criterion in the table is ease of implementation. The octree is once again the best in this category. It is extremely simple, and does not require data analysis or re-balancing, which can be complex, especially when heuristics are involved. The R-tree, in comparison, would be very complex to implement for voxel data, since bounding boxes can overlap, which would cause ambiguities without some system for resolving them.

It should be noted that the special situations and qualities, for which the various types of tree are best suited, are not necessarily guaranteed to appear in the data sets that we will be expecting for our work. For example, the spatial adjacency lookups of R-trees, and the nearest-neighbour searches that kd-trees excel at, are unlikely to be particularly useful in our work. This is why special qualities of the tree types are not considered.

Based on the above taxonomy and evaluation, we believe that a standard

octree is the most suitable type of tree for our purposes. It is the best option for performance, the easiest to implement, and the minor advantage the others have in memory savings are not significant enough to offset the octree's other benefits.

Other implementations of our algorithm might use different types of trees and potentially, the specialized features they bring. However, in our implementation we make no assumptions about the data sets and opt for generality, using a basic octree.

3.3.2 Shape Tag Handling

When shapes produced from the shape grammar in the first stage are voxelized, the tags associated with them are inherited by the voxels produced during the shapes' voxelization. Any voxel that falls inside the shape when it is voxelized will inherit any tags that shape had. See Figure 3.6 for an example of this.

In the case of overlapping shapes, it is possible that a voxel may inherit tags from multiple shapes. This is not problematic at this stage, but may have unintended consequences during detailing.

For example, when creating a tunnel through a building using subtractive shapes, shapes to create features on the exterior of the building may have significant overlap into the interior of the building. Hence, the voxels in the interior of the building would have multiple tags, including from shapes intended for the exterior of the building. As a result, if the created tunnel passes through areas with these multiple tags, the walls of the tunnel could have occasional patches detailed as though they were part of the building's exterior.

When designing grammars, users should consider the possibility of voxels having tags from multiple, overlapping shapes, and ensure that their grammars can handle these situation correctly.

3.3.3 Order of Shape Addition

The order in which shapes are added is also important, because shapes may be additive or subtractive (additive for creating solid structures, or subtractive for carving empty spaces out of solids). Adding and subtracting geometry in this manner is not commutative. Hence a grammar may generate unintended results, depending on the order in which the shapes are voxelized. Figure 3.7 shows a visual example of this problem.

To resolve this ambiguity, the shape grammar rules can assign a *priority* to the shapes. This is an integer attribute associated with each shape that determines when the shape will be voxelized. Before voxelization, the shapes are sorted by priority, and are then added in sorted order. Any sorting algorithm can be used, since there are no special requirements or complications when sorting the shapes. This allows a user to explicitly control when shapes are added, and resolve order-dependency issues.

For example, consider Figure 3.7 which attempts to create a rectangle with a circular arch above it (the rightmost model in the figure). The model is constructed from three shapes, a rectangle and two circles, the smaller of which is subtractive. Order is important, because if the subtractive shape is created either first or last, it will produce an incorrect model (the left and middle results in the figure). The shapes must be added in a specific order to produce the

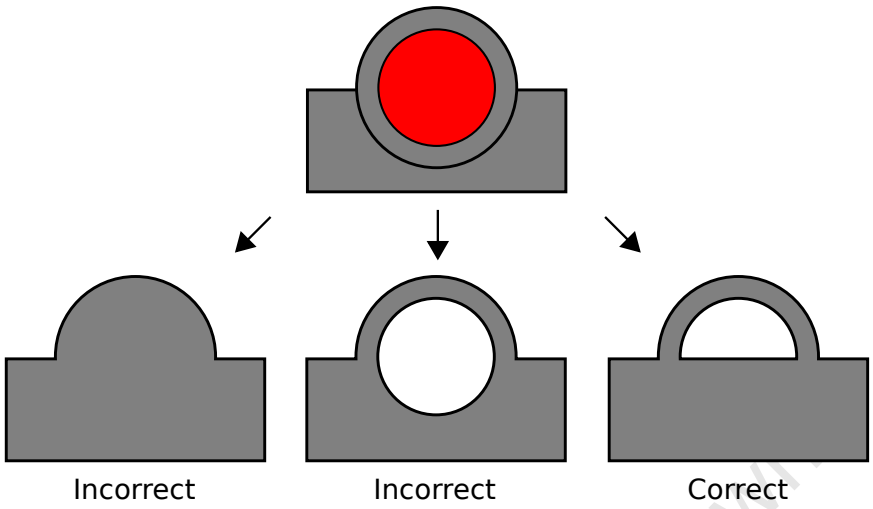


Figure 3.7: An example of how order is important when creating additive and subtractive shapes. The red inner sphere is subtractive, and the grey larger sphere and rectangle are additive. Applying the subtractive sphere first, last, or in between the other two shapes will produce different results, some of which will not be what the user desires.

desired result. First the larger additive circle, then the subtractive circle, which will create a hollow sphere. Finally, the rectangle must be added, which fills the lower half of the circle and produces our desired result. Any other order of shape addition will produce incorrect results.

Figure 3.8 shows a simplified grammar that produces the correct result for the example in figure 3.7, along with the prioritized output. Note how each shape is assigned a priority to ensure they are voxelized in the correct order.

3.3.4 Manual Editing

Finally, it is possible to manually edit the voxel grid once the shapes have been voxelized. This could be done to allow hand-crafted modifications to the output of a grammar, or because the user is dissatisfied with some aspect of the output that is difficult to correct in the grammar.

Manual editing is an important capability for artists and modellers, and our shape grammar extensions do not restrict this at all, although it does require voxel editing software.

Supporting manual editing of the voxel grid was not an explicit design choice. However, a benefit of the way our algorithm operates (in stages) is that manual editing of the grid is possible between the stages.

A significant problem with manual editing is that tagging information would be lost through the use of voxel editing software that is not tag-aware. A custom voxel editing program that could preserve and change tags would solve this problem, but we recommend that if the user wishes to manually edit the shape grammar output, they edit the final mesh produced by our algorithm (see Section 3.6) instead of the voxel data set.

In a similar vein, it would be possible to have grammar-aware manual editing

```
Shape Grammar:

Axiom: outer_circle (additive = true, priority = 1)

Rule 0: outer_circle -> outer_circle | inner_circle(additive =
  false, priority = 2) | rectangle(additive = true, priority = 3)

Output:

1: outer_circle (additive = true)
2: inner_circle (additive = false)
3: rectangle (additive = true)
```

Figure 3.8: A simplified version of the grammar that produces the correct output from Figure 3.7, and the final output in priority order. Note the use of the additive flag, and priorities, to ensure the correct result is produced.

software. Such software would take account of the underlying grammar when editing operations where performed. For example, when editing one symmetric copy of a shape, the software could simultaneously apply the same operation to all other copies of the shape. Should the user wish to perform manual modifications at this stage, it would be beneficial if the software used was grammar-aware.

The final output of this stage is a 3D voxel grid, where each voxel is either solid or empty, and may have metadata tags associated with it.

3.4 Voxel Detailing

In this stage of the algorithm, the voxels are assigned a concrete appearance that will be used when displaying the final model. This can include, but is not limited to, texturing information, bump maps, displacement maps, lighting information, and materials. The goal is to provide the model with visual detail to be used for display purposes. A simple example of a voxel grid after undergoing detailing is shown in Figure 3.9, and pseudocode of our detailing process is found in Algorithm 3.3.

Algorithm 3.3 Voxel Detailing

```
detailingRuleSet ← getDetailingRuleSet()
maxIterations ← getMaxDetailingIteration()
surfaceVoxels ← voxelGrid.getSurfaceVoxels()
for i = 1 to maxIterations do
  for all j ∈ surfaceVoxels do
    neighbours = voxelGrid.getNeighbouringVoxels(j)
    j.detailingTags ← detailingRuleSet.runRules(j, neighbours)
  end for
end for
```

This is done on a per-voxel basis, by a user-created *detailing rule set* which operates on each voxel individually. These rules may iterate over the voxels multiple times, allowing the creation of complex multi-pass detail.

Detailing Rules
rule_0: **if** "material:roof" in tags **then** texture = red_tile
rule_1: **if** "material:chimney" in tags **then** texture = black_stone
rule_2: **if** "material:stone" in tags **then** texture = random_selection(grey_brick, dark_brick)
rule_3: **if** voxel.borders_enclosed_space() == true **then** texture = blue_paint

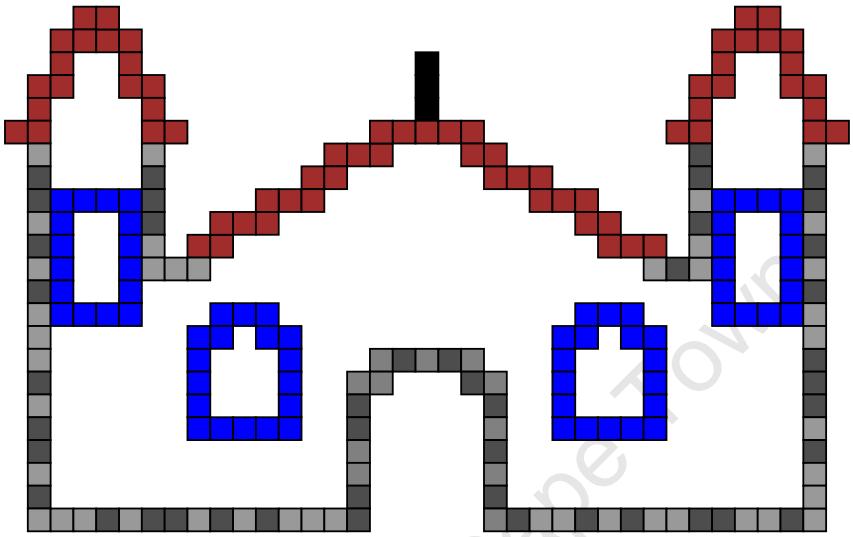


Figure 3.9: The voxel grid from figure 3.6 after detailing. Each voxel has been assigned a texture in accordance with the detailing rule set supplied. Non-surface voxels are ignored, and are not displayed in the diagram. Pseudocode for the detailing rules is shown.

An example of such multi-pass detail would be the plasma-like pattern shown in Figure 3.10. The pattern works by having a base colour and seed points created in the first pass. Then in subsequent passes, the seed points are randomly grown outwards with cells more distant from the seed point being given lighter colours. Pseudocode for this detailing rule set is found in Algorithm 3.5.

The specification of the detailing rule set is broadly similar to the specification of the shape grammar, but features a number of differences. Firstly, there is no axiom. The detailing rules operate independently on whatever information is contained in the voxel passed to them, and do not require any axiom.

As with the shape grammar rules, the exact syntax of the detailing rules depends on what operations and capabilities are made available to the detailing process, but the rules should be specified in a manner that is easily parse-able. Additionally, some method for indicating which rules take priority in ambiguous situations (when either of two rules could be used on a voxel) should be included.

Global parameters for the detailing process, such as the maximum number of detailing passes, must also be included with the detailing rule set. The only mandatory global parameter is the number of passes, but additional extensions used by the detailing rule set may also require global information specified by the user. For example, if detailing a model with a dynamic camouflage pattern that depends on the environment, using context-sensitive rules, those environmental details would be specified as global parameters, so that the rules could access

them.

One particular advantage of operating at a per-voxel level, is that cellular automata (CA) patterns can be created, since they map very well onto the discrete, gridded nature of voxels.

Cellular automata are a class of automatons that operate on discrete grid-like spaces over multiple iterations [55]. Each cell in the space has an associated state ranging from a simple on or off, to complex multi-dimensional information. Cellular automata have a function that is applied to every cell on each iteration, which may update the state. This is a very close mapping to our method of doing surface detailing, which makes cellular automata easy to run under our framework.

However, in our implementation, the detailing rules are focused on 2D surface detail, not the solidity of the voxels themselves. Hence, the rules are limited to operating on the detailing tags of voxels, and cannot modify whether individual voxels are solid or empty. This capability could be added in future work though, allowing automata patterns that operate on the voxels themselves.

In spite of their relatively simplistic nature, cellular automata are capable of generating surprising emergent behaviour across multiple iterations. For example, Langton's Ant [25] has been proven to be Turing complete, and Rule 30 [55] produces aperiodic, chaotic output. This means that cellular automata are able to produce complex detailing on models under our framework.

As mentioned above, detailing is done on a per-voxel level as opposed to the per-shape level of conventional shape grammars. Operating at a per-voxel level means that detailing elements are not constrained by shape boundaries, allowing the creation of detail features which span shapes and work on sub-shape scales.

This is the main justification for doing detailing in this manner. With the voxel grid, there is no distinction between the different shapes produced by the shape grammar, and the rule set is free to detail the voxels without being constrained by shape boundaries. This allows the easy creation of global detailing patterns that consistently cross different shapes in the model. In conventional grammars this would be much more difficult to do as the textures on adjacent polygons faces would have to be aligned, to avoid visible seams in the detailing. This sort of texture alignment is very difficult to do algorithmically (that is, without human guidance) especially in a procedural context.

The disadvantage to this detailing freedom which our per-voxel scheme brings, is more complexity for the user. This complexity could be reduced in two ways. Firstly, by creating a visual rule editor, as opposed to text-based programming. Secondly, by designing an interface that allows rapid prototyping of rules on small examples, to quickly detect problems. However, we did not implement these, and leave them to future work.

The scope of these rules is extremely broad, and features such as context-sensitivity and randomness can easily be included. Everything from assigning a simple texture, based on position, to complex randomized multi-pass procedural methods are possible. In our implementation, the rule set was specified in Python, allowing the vast scope that comes with the expressive power of a programming language.

In fact, this method of detailing voxels has many similarities to pixel shaders. With pixel shaders, each pixel on the screen is operated on independently by a small routine of code. The shader can be fed arbitrary information which can be used in processing that pixel, and the final result is that the pixel has been

modified in some manner so that the collection of modified pixels form a visual effect on the screen.

Our detailing scheme is very similar to this, but differs in that it is three-dimensional, and the results of detailing are passed for use in later stages of our algorithm, instead of being displayed directly.

When detailing the voxel grid, relevant details about the voxel are passed to the rule set. In our implementation these details are:

- Tags associated with the voxel.
- The normal of the voxel.
- The maximum resolution of the octree.
- The coordinates of the current voxel.
- The count of the current iteration of the rule set.
- The maximum number of iterations to be run.
- The above details for all neighbouring voxels, within a user-specified radius.

Note that the voxel's normal is calculated from the neighbouring voxels. By examining which of the neighbours are solid and which are empty, a rough normal can be derived.

In our implementation, the parameters of each voxel that the detailing rules can set or modify are:

- The name of a texture to use for the triangles generated from the voxel.
- A vector to displace the normals of generated triangles, which allows a voxel-level form of bump mapping.
- Displacement mapping information, used to modify the positions of vertices generated from the voxel.
- An RGB colour used to modify the lighting of triangles generated by that voxel.

We do not claim that the above inputs and outputs are sufficient for all situations, but in our work we found them to be adequate. By this we mean that in our testing we never encountered a need to modify other parameters beyond these. Still, there may be situations where other implementations of our algorithm would wish to add other parameters that can be read or modified.

Fortunately, it would be easy to add other inputs to the detailing rule sets and parameters that could be modified by them. A few options that future implementers may wish to consider are: parallax mapping information, bump maps for the textures, and the ability to add imported pieces of pre-made geometry to the generated mesh (such as placing manually created gun turrets on a generated space ship).

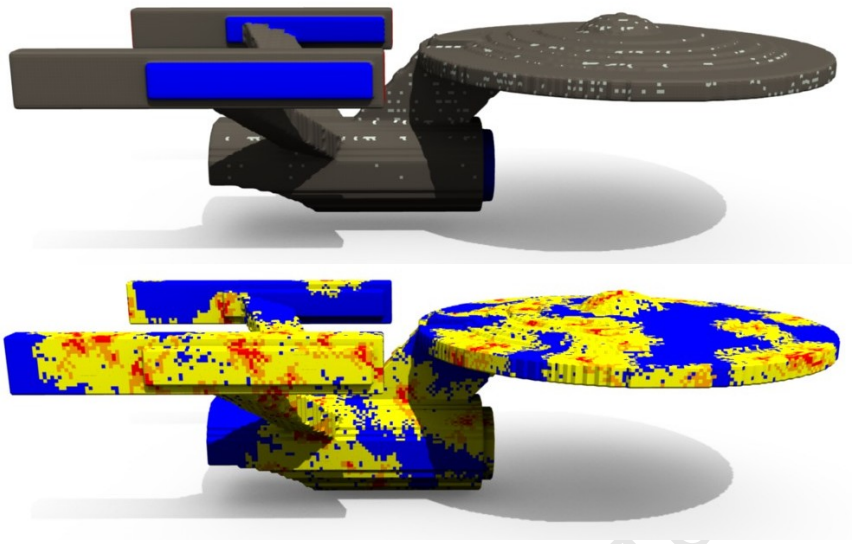


Figure 3.10: Our Enterprise model with two different detailing rule sets applied to it. On the top, with its original detailing, and below with a randomized plasma-like pattern. This shows how detailing rule sets that are not reliant on shape tags can be applied to any voxel grid.

3.4.1 Shape Tag Dependence

It should be noted that this detailing stage is independent of previous steps. A detailing rule set can be applied to any voxel grid, and does not need to concern itself with how that data set was produced.

It is possible to have a detailing rule set that is completely independent of metadata tags. For example, detailing that creates a consistent pattern across the entire model without using the context information from the tags. These detailing rules will work on any model provided to them, regardless of tags. An example of such a detailing rule set being applied to a model intended for a different rule set is shown in Figure 3.10 (below the model with its original detailing).

We also show pseudocode for the two detailing rule sets in figure 3.10 in algorithms 3.4 and 3.5. These rule sets consist of a single rule that operates on all pixels. In other rule sets, there may be multiple rules, which can call each other, similar to methods in a programming language.

The Enterprise detailing rule set works in a single pass, uses tags to determine how to detail the model, and randomness to assign the window's positions.

The plasma detailing is a multi-pass rule set, which produces randomized flame coloured blobs on a blue base colour. The number of iterations determines how big the plasma patches are. Numbers are used to indicate how bright a voxel in the blob should be: 0 indicates red, 1 indicates orange, 2 indicates yellow, and 3 indicates the base colour, blue. The translation from numbers into colours is done once the rule set has completed running.

Note how the Enterprise rule has conditional statements that rely on shape tags, while the plasma rule does not reference tags at all.

Algorithm 3.4 Example Detailing Rule Set: Enterprise

```
if any("disk", "neck", "body") ∈ voxel.tags then
  if abs(voxel.normal.y) < 0.6 and voxel.position.y mod 3 = 1 and
  randint(1, 4) = 1 then
    voxel.detailing_tags ← "window"
  else
    voxel.detailing_tags ← "grey"
  end if
else if any("thruster", "front_dish") ∈ voxel.tags then
  voxel.detailing_tags ← "glowing_blue"
else
  voxel.detailing_tags ← "grey"
end if
```

Algorithm 3.5 Example Detailing Rule Set: Plasma Pattern

```
if current_iteration = 1 then
  if randint(1, 500) = 1 then
    voxel.detailing_tags ← 0
  else
    voxel.detailing_tags ← 3
  end if
else
  brightest_neighbour ← 3
  for all i ∈ voxel.neighbours do
    if min(i.detailing_tags) < brightest_neighbour then
      brightest_neighbour ← min(i.detailing_tags)
    end if
  end for
  if brightest_neighbour < 3 and voxel.detailing_tags >
  brightest_neighbour then
    random_num = randint(1, 3)
    if random_num = 1 then
      voxel.detailing_tags ← brightest_neighbour
    else if random_num = 2 then
      voxel.detailing_tags ← brightest_neighbour + 1
    end if
  end if
end if
```

In most practical situations though, we expect that rules from the detailing rule set will be dependent on metadata tags in the voxel data set. For example, detailing a house's walls with brick textures and the doors with wooden ones requires that the two parts of the model be distinguished. Hence the user should ensure that the shapes in the shape grammar are properly tagged for the detailing stage.

3.4.2 Performance

Running snippets of code for each voxel in a grid can be extremely slow, especially so if the grid is large, or the rule's code is complex. For this reason, we only run the rule set on surface voxels in the grid. This typically decreases the input size from $O(N^3)$ to $O(N^2)$, in the size of the voxel grid.

We define a surface voxel to be any solid voxel in the grid which is 26-connected to at least one empty voxel. Because the marching cubes algorithm (used in the next stage of the algorithm to produce a mesh) does not generate triangles for completely empty or solid space, only voxels on the border between solid and empty space will affect the final model. All others can be safely ignored.

Using the hierarchical structure of the octree, surface voxels can be quickly identified. If a node of the octree does not have any children, then only the voxels around the edge of that node need be checked further. For all reasonable models, this dramatically reduces the number of surface voxel candidates that need to be checked, improving speed from $O(N^3)$ (for the brute force search) to $O(N^2)$.

The final output of this step is a voxel grid where all surface voxels have been assigned detailing information. This information must unambiguously provide all information required for rendering the grid, either as is, or when converted to a mesh.

3.5 Mesh Generation

In this stage of the algorithm, the voxel grid is converted into a mesh format, which is more suitable for use in most graphics applications than a voxel grid.

It is possible to make use of the generated model as a detailed voxel grid, terminating the algorithm without doing this step. There are many methods for rendering voxel grids, often based on ray casting [27], or point rendering [46]. The user may wish to render the model using one of these methods, or perhaps to use the model in some sort of voxel game engine, such as Minecraft.

In this case, there are some considerations that should be made about the detailing process. Firstly, texturing information is no longer relevant, as the data is a collection of points, which cannot be used with a texture. Voxels should be assigned colours or some other usable detailing information instead. Secondly, if rendering the voxel grid with volume ray casting, it may be necessary to retain and detail the interior voxels, as they will be used by the algorithm.

However, most graphics applications today work with triangle meshes, not voxel data sets. For this reason, we need to convert our voxel data set into a mesh that can be used in conventional raster graphics applications, such as modern 3D game engines.

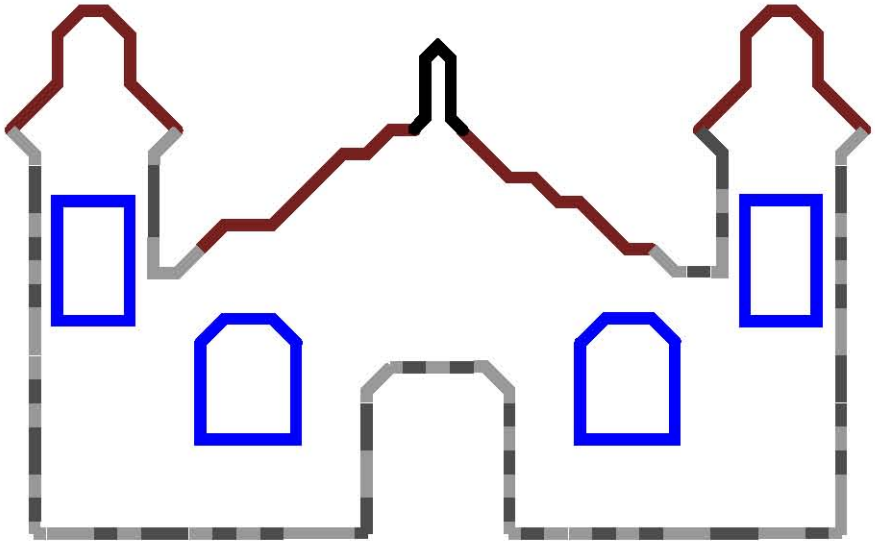


Figure 3.11: The 2D mesh produced by meshing the voxel grid shown in Figure 3.9.

The marching cubes algorithm [30] is a well-established solution to the problem of extracting a mesh representation of a voxel grid or isosurface. We make use of it to produce a mesh version of our generated model. A comparison between a voxel grid, and its mesh produced by the marching cubes algorithm is shown in figure 3.12. Additionally, a 2D example of a produced mesh is shown in Figure 3.11. Pseudocode for the mesh generation process is found in algorithm 3.6.

Algorithm 3.6 Mesh Generation

```
voxelGrid ← getVoxelGrid()
surfaceVoxels ← voxelGrid.getSurfaceVoxels()
outputMesh ← ∅
for all i ∈ surfaceVoxels do
    currTriangles ← marchCube(i)
    currDetailingInfo ← getDetailingInformation(i)
    currVoxelMesh ← buildGeometry(currTriangles, currDetailingInfo)
    outputMesh.addGeometry(currVoxelMesh)
end for
return outputMesh
```

One could also use the marching tetrahedra algorithm [12], a successor to marching cubes. It was designed to solve an ambiguity in the marching cubes algorithm (one item in the original lookup table was a symmetric copy of another), and to provide a patent-free algorithm for producing a mesh from an isosurface or voxel grid (the marching cubes algorithm was originally patented, but this has since lapsed).

Although the marching tetrahedra algorithm can produce a finer mesh than the marching cubes algorithm, in practice their results are very similar for large

voxel grids. We opted to use the marching cubes algorithm in our implementation, but there is no reason that other implementations could not use marching tetrahedra.

The marching cubes algorithm operates by iterating (or marching) over the voxel grid, and at each point, it determines what triangles to produce by examining which of the neighbouring voxels are solid or empty.

The marching cubes algorithm does not march over the voxels themselves though. Instead, it iterates over the spaces between voxels, specifically, the points exactly midway between the surrounding 8 voxels, where the corners of those voxels meet.

At each of those points in the grid, the algorithm examines whether the eight surrounding voxels are solid or empty. Depending on which of those voxels are solid or empty, a group of triangles that fit into a cuboid selection of geometry is selected. That cuboid is then placed at the current point, centred on the point where the corners of the surrounding voxels meet.

To minimize processing time, a pre-computed lookup table is used for determining what selection of triangles should be placed at the current point.

This process is repeated across the entire voxel data set, with the triangles created at each point being added to a mesh. Once the entire grid has been marched over, a final mesh version of the model will have been created.

Because this requires examining every voxel in the grid, it has a running time of $O(N^3)$, where N is the size of the voxel grid. This means that, in practice, the algorithm is slow to run.

With a tree data structure, such as an octree, large areas of empty and solid space can be skipped, but this will only provide a constant factor speed increase (dependent on the data set in question). It does not provide an order of magnitude improvement. Additionally, using the octree in this manner increases the implementation complexity of the algorithm.

The algorithm outputs a list of triangles, each of which can be associated with a voxel in the input grid. As each triangle is generated it is assigned textures, materials, and other detailing information that was assigned to the voxel it was generated from.

Hence, the final output of this step of the algorithm is a fully textured and detailed mesh representation of the voxel grid that the original shape grammar produced.

3.6 Mesh Post-Processing

The mesh produced by the marching cubes algorithm is suitable for direct use in graphics applications, but its visual quality could be improved by post-processing.

One of the problems with the marching cubes algorithm (and marching tetrahedra) is that the output mesh has visual artifacts caused by the discrete nature of the voxels. Curves in particular, are not fully captured during the meshing process, and will instead appear 'bumpy'. This is actually a form of aliasing [54] arising from the low resolution of a voxel grid, so increasing the resolution of the voxel grid can reduce this problem.

A less computationally intensive solution to this problem is using an appropriate mesh smoothing algorithm, which will remove such aliasing artifacts [20].

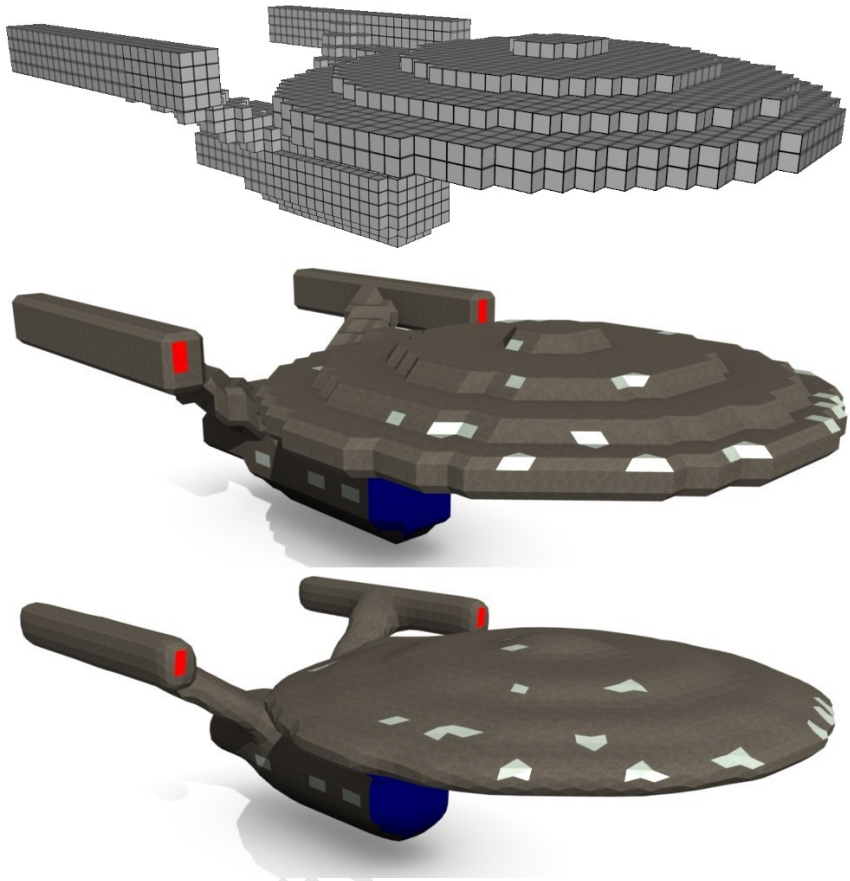


Figure 3.12: A comparison of a low resolution version of the enterprise model as a voxel grid, an unsmoothed mesh, and as a smoothed mesh. Note that this example is purposely at a low resolution to better display the stages of the algorithm. A model intended for production use would be generated at a higher resolution.

An example of how mesh smoothing can improve a model's visual quality is shown in figure 3.12.

However, as is seen in the image, using mesh smoothing on models produced by low resolution voxel grids can distort detailing elements (the windows in the smoothed model). This occurs because the triangles to which the textures are mapped will have the positions of their vertices changed by the smoothing algorithm. Different smoothing algorithms operate on the mesh in different ways, and hence will cause different amounts of distortion, but any smoothing will introduce some degree of distortion.

There are texture-aware mesh smoothing methods that aim to minimize texture disruptions [5], however in our implementation we opted not to use such an algorithm. This is because, in our testing, we found that the distortion of detailing elements was seldom a problem for voxel grid sizes of 256 or more, where the triangles are smaller and hence moving them affects the overall model

less. However, the distortion is noticeable at lower resolutions.

Figure 3.12 was produced at a very low resolution to aid in understanding of the algorithm, and thus the texture distortion is significant. At higher resolutions, the distortion is greatly reduced. For examples of this, see figures 3.10, and 4.6 to 4.14. These models were all created a higher resolution, with smoothing applied, and there is no texture distortion to the degree seen in figure 3.12.

Smoothing can be done either automatically, as part of the generation process, or by hand, using a standalone modelling software package, such as Blender (which we opted to use in our work).

Using a standalone program has the requirement of needing a human to perform the operations. This can be a benefit where fine control is required, or potential problems must be caught, such as smoothing a very unusual mesh where the process could deform it in unwanted ways.

However, if working with many models where the quality does not need to be perfect, or the smoothing will not cause problems, saving time by automation of this step may be preferable.

In addition, manual editing and tweaking of the final mesh can be undertaken at this point. Because it is now a regular textured mesh, conventional modelling software can be used, and no voxel-specific features are required.

3.7 Implementation and Optimizations

We implemented our algorithm largely in C++. However, Python was used for both the shape grammar and voxel detailing rule sets. This was because, although slower than a compiled language, the interpreted nature and ease of embedding Python greatly simplifies and reduces the programming work needed to create the shape grammar and detailing rule set interpreters.

The rules are described in Python code, and as a result, the legwork of parsing and running them can be offloaded to the Python interpreter.

The implementation of our algorithm was intended to be a proof-of-concept, and not necessarily suitable for practical use. This, combined with time constraints, meant that we did not focus on optimization, although there is large scope for significant performance gains.

In particular, we have identified a number of potential optimizations that could dramatically reduce running times. We discuss them here.

3.7.1 Voxelization

The voxelization process is not suitable for parallelization, due to the order of shapes being important, which would require frequent locking of sections of the octree to keep the voxel grid consistent with the shape grammar. The high occurrence of locking data would hamper parallelization speed-ups, most likely making them negligible.

However, voxelization of shapes can be performed extremely efficiently in a non-parallel manner by exploiting the hierarchical nature of the octree.

Beginning at the root of the tree, query the intersection between each of the eight child octants of the octree node, and the shape to be added. If the area covered by a child octant is entirely within the shape, then that child, and

hence all the voxels that it contains, are set with the shape's information. If there is a partial intersection between the child node and the shape, then the algorithm is recursively called on that node. See algorithm 3.7 for pseudocode of this process.

Algorithm 3.7 Fast Recursive Shape Voxelization

```
tree_root ← getOctreeRoot()
shape ← getShapeToBeAdded()
for i = 1 → 8 do
  octant_boundaries ← tree_root.children[i].bounds
  degree_of_overlap ← getDegreeOfOverlap(octant_boundaries, shape)
  if degree_of_overlap = FULL then
    tree_root.children[i] ← Solid
  else if degree_of_overlap = NONE then
    tree_root.children[i] ← Empty
  else
    recurseOnChild(tree_root.children[i])
  end if
end for
```

While faster, this method of voxelization is more complex to implement, and requires a method of exact collision detection, such as the well-known GJK algorithm [9]. We recommend that any future implementations make use of this method to significantly reduce execution times.

3.7.2 Voxel Detailing

Voxel detailing is also highly parallelizable, because each voxel can be examined independently of the others, and no message passing or locking of data is needed. A multi-threaded implementation would increase performance linearly with the number of cores available.

However, the voxel detailing is completely unsuitable for acceleration with graphics hardware. Writing a rule interpreter for graphics processing units (GPUs) would be extremely challenging, due to the complexity and size of the resulting computational kernel. Also, there is likely to be significant amounts of branching in the code, which impacts negatively on performance when using graphics hardware.

3.7.3 Storage of Shape Tags and Detailing Information

In our implementation, the shape tags and output of the detailing rule set are collections of arbitrary strings that are associated with individual voxels. Other implementations could store these tags differently.

A point that should be borne in mind when deciding how to store this information is that having multiple items of data for each voxel can require a large amount of storage. One appropriate method of compression is to store the information in the octree, to benefit from the compression it provides. This should work well for most detailing rule sets, but the compression ratio will begin to deteriorate for surface detailing patterns with a high entropy, such as high frequency random speckles.

Another method of compressing this data, that can be used with the octree storage method, or independently, is data deduplication [31]. In most reasonable data sets, we expect that the same pieces of detailing information will appear often and be associated with many voxels in the grid. Instead of storing that information repeatedly in each voxel, we can store a reference to that data, which takes up less space than the data itself. The actual data itself then only needs to be stored once.

For maximum compression, this reference should be a 1 byte identifier, but that assumes that the total number of pieces of detailing information is at most 256 (the maximum number of possible states for a single byte). While we expect that most models will not require more than 256 detailing information strings, it may be necessary to have pointers as a fallback means of referencing the data.

Data deduplication can provide significant memory savings, especially when using a single byte for each reference. When combined with octree compression, the amount of memory required to store the detailing information will be drastically reduced for all but the most extreme detailing rule sets.

3.7.4 Mesh Conversion

The implementation of marching cubes can also be substantially accelerated by only marching over the surface voxels of the voxel grid. Since entirely solid or empty regions will not produce any triangles, the voxels of those regions need not be marched over. If a list of the surface voxels was stored during the detailing step, it can be re-used here, since there will be no change to the voxel grid between the time that list of surface voxels is generated, and the time that mesh generation is performed.

In addition to the above optimization, the marching cubes algorithm could be accelerated using graphics hardware, a technique which has been investigated previously [19]. Because each group of eight voxels can be examined independently of the others, the problem is highly parallelizable. In particular, because each group of voxels requires a small amount of work, and can be examined without branching, the algorithm is especially suitable for graphics hardware.

The above optimizations to the marching cubes algorithm are also equally applicable to the marching tetrahedra algorithm, should an implementation opt to use that technique instead.

3.7.5 Large Scale Generation

When using our algorithm in a production environment, where large numbers of models are being generated by a cluster, it would be advisable to pipeline our algorithm into the five different stages to achieve better resource utilization.

In a production cluster, more nodes can be assigned to perform the more computationally heavy stages of the generative process (detailing, mesh generation), while fewer nodes work on the quicker stages (shape grammar interpretation, voxelization, mesh post-processing). Additionally, the outputs from the different stages can be stored in stage-specific queues.

Once a node has finished processing an item, it will push it onto the back of the next stage's input queue and pull a new work item from the front of the current stage's input queue. This, combined with dynamic load-balancing of nodes, will ensure optimal utilization of computational resources.

There is another benefit to explicitly storing the outputs from each stage: Should a designer wish to experiment with a particular stage of the algorithm, they can store the output from the previous stage and work with that, instead of needlessly re-running the earlier stages repeatedly.

For example, if a model designer wished to experiment with tweaking the parameters of a detailing rule set, they could instruct the storage queue between the voxelization and detailing stages to not process the voxel grid that they wanted to experiment with, only to store it. After experimenting and deciding on the desirable parameters of the rules set could they mark the voxel grid as being suitable for moving onto the next stage. This way, the grammar interpretation and voxelization stages do not need to be re-run redundantly.

In a production environment, with large amounts of models being generated, a pipelined setup such as the above would improve resource utilization and reduce needless computational expenditure.

3.8 Summary

This chapter has presented our algorithm for interpreting shape grammars in a voxel-space, instead of the conventional mesh-based method.

The inputs to the process are a shape grammar, consisting of the axiom, rules, and global parameters, and a detailing rule set, with its associated global parameters. The final output from our algorithm is a textured mesh, suitable for use in other graphics applications.

The algorithm is broken up into 5 stages: shape grammar interpretation (where the shape grammar is executed to produce a collection of tagged shapes), shape voxelization (where the grammar output is converted into a voxel grid), voxel detailing (where the voxels are assigned detailing information), mesh generation (where the voxel grid is converted into a polygon mesh), and optionally, mesh post-processing (where the generated mesh is smoothed and refined).

There are a number of points to consider when implementing our framework, including the choice of tree for storing the voxel grid, the exact syntax for the rule sets, and mesh smoothing algorithms. Additionally, there are a host of optimizations that can be used to accelerate the model generation (the performance of which is discussed in the following chapter), some of which introduce associated constraints, and some of which do not.

Chapter 4

Testing and Results

This chapter covers the testing and experimentation we undertook to validate our shape grammar extensions. In order to evaluate our voxel-space extensions to shape grammars, and their suitability for practical use, we undertook three branches of experimentation:

- **Performance Testing:** (Section 4.1) We analyzed the running time and memory required for our implementation of the algorithm, across a range of input sizes and types. This included examining the running time and memory usage of each of the stages of the algorithm.
- **Variation Testing** (Section 4.2) This was oriented at ensuring our ability to produce multiple models of acceptable quality from a single stochastic grammar/detailing rule set pair. For these tests, we developed stochastic grammars and ran them repeatedly to evaluate their outputs.
- **Output Range Testing** (Section 4.3) These tests were aimed at ensuring that our extensions increase the generative range of shape grammars, and do not limit the outputs. We developed a variety of shape grammars and detailing rule sets for this purpose.

We believe that these three tests cover the key areas of evaluation for our algorithm extensions, and provide an adequate examination of whether our work has practical application because they tie into the success metrics defined by our research questions presented in section 1.7.

Performance testing examines whether our extensions are computationally viable, which was one of the main research questions in this work. Variation testing forms part of the evaluation for another one of our research questions (whether our extensions introduce limitations) by testing whether our extensions still allow the creation of a range of models in a similar, which is an important ability of shape grammars. Finally, the output range testing provides an answer to the final research of our work, by testing whether our shape grammar extensions do allow the creation of models that are hard to create under conventional shape grammar systems.

Therefore, each of these three areas of testing is directly tied to our research questions (and hence to our success criteria), and collectively cover the key evaluation areas of our algorithm.

We cover each kind of testing individually in Sections 4.1 to 4.3, discuss the limitations of our shape grammars extensions in Section 4.4, and draw conclusions from our testing in Section 4.5.

4.1 Performance Testing

We analyzed our algorithm's performance across a variety of voxel grid sizes and user inputs (see Table 4.1). The goal was to cover a wide variety of inputs and options, to ensure that our testing was representative of real-world use cases.

The two main results of interest from the performance testing are the time taken to generate a model, and the peak memory usage of the process.

We do not concern ourselves with testing secondary storage requirements, because the algorithm itself does not require any secondary storage, beyond what is needed to store the input and output files. The input grammar files are very small (on the order of a few kilobytes), but the output models can be sizable, depending on factors such as the voxel grid resolution and the number and complexity of textures. However, the model sizes are as large as they would be if created by other methods, and thus we do not consider this aspect any further.

A final point that the reader should note regarding these performance tests, is that our implementation was strictly intended as a single-threaded proof-of-concept. Thus, performance was not a priority, and there is large scope for improvements in this area (as mentioned in Section 3.7). It is possible that those optimizations could offer order of magnitude improvements to running time, and noticeable savings on memory usage.

Nonetheless, we still provide the results of testing our implementation. This provides a baseline comparison point for future implementations, and shows that the performance of an implementation that is not fully optimized is still sufficient for actual use.

The following three subsections present the details of our testing setup and the results for running time and memory usage.

4.1.1 Testing Setup

Performance testing was conducted with a selection of 15 shape grammar and detailing rule-set combinations, at 4 different voxel grid resolutions. The details of the different testing inputs are shown in Table 4.1.

Only the files sizes for mesh models produced at a grid size of 256 are shown in the table. The file sizes of the mesh models produced at other grid sizes are, on average, proportional to the cube of the size of the grid. For example, the file sizes for models from a grid of size 512 are roughly 8 times larger than those shown in the table for grids of size 256. This is to be expected since the number of voxels, and hence the number of triangles produced, is proportional to the cube of the grid's resolution.

The selection of grammars and rule sets used in our testing (those in the table) was specifically chosen to encompass a wide range of complexity and types of generated content, from simple mathematical shapes, to complex models of structures. Many of the grammars and rule sets also included elements that we intended our shape grammar extensions to be able to create easily, such

Shape Grammar	Detailing Rule Set	Iterations	Mesh Size
castle	castle	5	34.5 Mb
tank	tank	1	25.9 Mb
space_station	station_detail	5	58.8 Mb
skyscraper	skyscraper_detail	1	54.6 Mb
enterprise	enterprise_detail	1	16.2 Mb
enterprise	camouflage	1	16.3 Mb
enterprise	plasma	15	16.3 Mb
enterprise	rule110	20	16.3 Mb
fractal_tree	grey	1	15.2 Mb
spiral	stripes	1	9.1 Mb
spike_ship	spaceship_windows	1	24 Mb
spike_ship	tiger_stripes	15	23.8 Mb
c_station	station_detail	5	33.5 Mb
flat_rectangle	rule30	25	31.6 Mb
flat_rectangle	rule90	25	31.6 Mb

Table 4.1: This table shows the different shape grammar/detailing rule set combination that were used for testing our research. The first column is the name of the grammar used to create the model, and the second is the detailing rule set used to detail it. The third column is how many iterations the detailing rule set took, and the fourth is how large the generated mesh was (in OGRE's native format). The file sizes shown are those created with a voxel grid of size 256. Finally, note that these are the unoptimized mesh file sizes.

as complex multi-pass cellular automata patterns (such as the 'tiger_stripes' pattern shown in figure 4.13) and recursive use of Boolean geometry operations to create complex shapes (such as the 'c_station' model in figure 4.8). For the above reasons, we believe that it is an adequate and appropriate selection of test cases.

The voxel grids used were all cuboid, with resolutions ranging from 64 voxels per side, suitable for rapid prototyping of ideas, to 512 voxels per side, suitable for producing high quality models for real-world use.

We limited our resolutions to 512 because our initial testing of a larger size (1024) indicated that cases with a high number of detailing passes would take an infeasible amount of time to run (given the time constraints we were under), and would require more memory than we had available. As mentioned above, our implementation was not heavily optimized, and we believe that a fully optimized implementation using the techniques mentioned in Section 3.7 would be able to run high resolution examples in a reasonable amount of time.

Additionally, in testing we found that a resolution of 512 is sufficient to generate high quality models. That is, when combined with mesh smoothing, a resolution of 512 is the threshold for when generated models have no obvious artifacts.

For the above reasons, and because our implementation was intended as a proof-of-concept and not a production implementation, we did not pursue testing at resolutions higher than 512. Still, we expect that performance characteristics at higher resolutions would follow the trends we discovered below.

Testing was performed on a computer with an Intel Core 2 Duo clocked at 2.4Ghz, 3 gigabytes of RAM, and an Nvidia GTX 280. The operating system was 64-bit Ubuntu Linux, version 11.04.

4.1.2 Algorithm Running Time

Timing information for the C++ component of our implementation was obtained using the C Time Library (ctime), while the Python shape grammar interpreter used the Python datetime module. Both of these are able to provide timing information of more than sufficient accuracy for the purposes of timing sections of our code. Timestamps were recorded at various points during the execution, and the differences between them were computed to determine how long sections of code took to run.

We decomposed the running times of the test cases into the time taken for each of the four principal stages of the algorithm. Post-processing was excluded as it is an optional step, and because our implementation used the external modelling software, Blender, for that stage of the algorithm.

The timing information reflects only the time taken for each stage of the algorithm to run. It does not include time to load and save input and output files, handle program arguments, initiate the graphics display, etc. In practice though, the time for that setup was negligible compared to the time needed to run the algorithm.

Grammar Interpretation

The first thing to note is that the shape grammar interpretation is orders of magnitude faster than the other stages, as the histogram in Figure 4.1 shows.

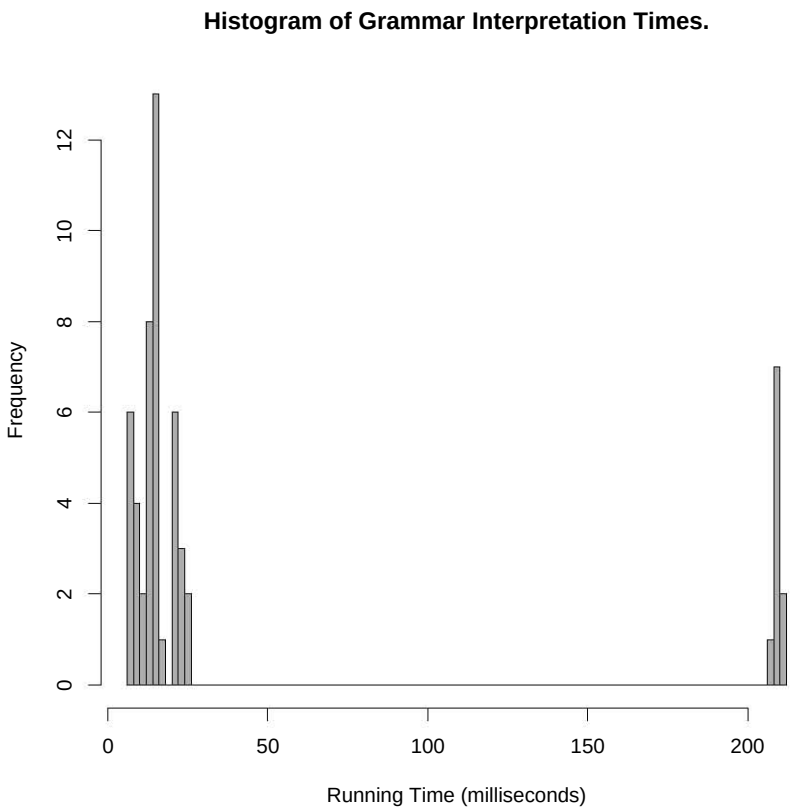


Figure 4.1: A histogram showing the distribution of running times for the grammar interpreter, taken across all of our test case grammars. Note how the majority of running times were very quick (under 25 milliseconds), except for the models exhibiting fractal behaviour, which took around 200 milliseconds to generate.

This speed is because the interpretation is independent of the voxel grid resolution, and only involves operating on strings and some basic mathematics.

It is possible for the interpretation of some grammars to have a running time that is exponential in the number of grammar iterations, as is commonly the case in L-systems. However, in practice, most reasonable shape grammars do not exhibit this exponential behaviour, and when they do, the number of iterations is often low enough that the exponential nature is not apparent (less than 10).

The majority of the interpretations times are clustered between 0 and 25 milliseconds, with 10 outlier running times of slightly more than 200 milliseconds. These outliers were the interpretations of the fractal tree and skyscraper models shown in figures 4.11 and 4.12. Both of these grammars have a large amount of branching, due to the fractal nature of the tree, and the repeated subdivisions to create the skyscraper's grid-like façade of windows. Hence they exhibit the exponential behavior described above, although for the fractal tree, the number

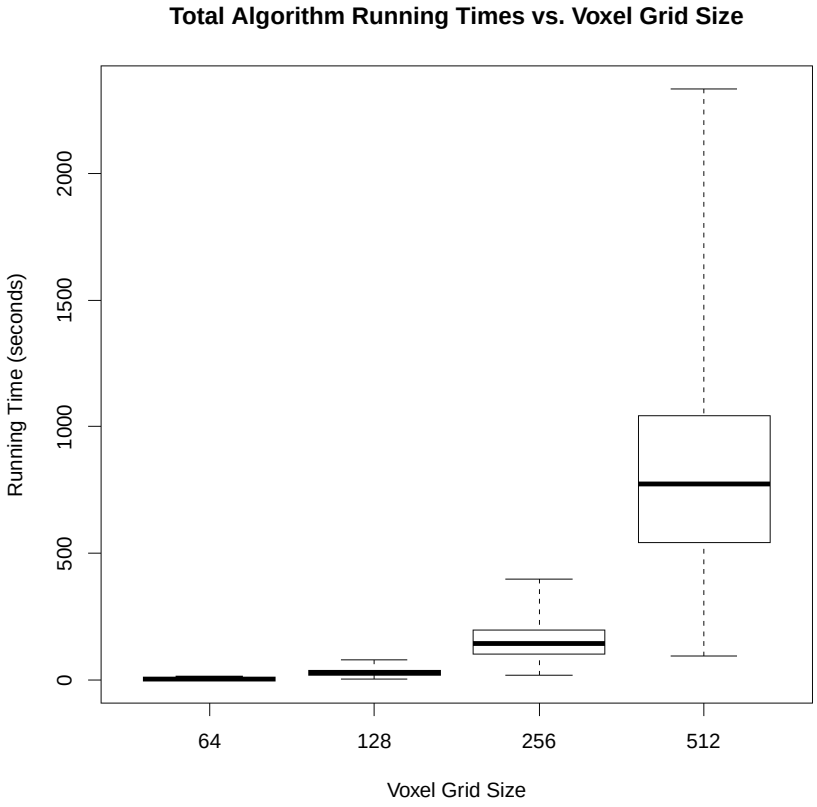


Figure 4.2: A box and whisker plot of the total algorithm running times (the sum of the voxelization, voxel detailing, and mesh creation times), across the four different resolutions. The minimum, maximum, and average running times all grow sharply as the resolution increases, and cover a time range of nearly instant to 35 minutes. Similar breakdowns of the time for each algorithm stage, across the different sizes, are included in figures A.1 to A.3 in Appendix A.

of iterations was capped to 10 iterations to keep running times reasonable. The skyscraper grammar terminates naturally after 5 iterations.

All of the interpretation running times are relatively quick, and more than sufficient for the purposes of an offline procedural generation algorithm.

Due to the tiny relative time required, we do not consider grammar interpretation relative to the times taken for the other stages of the algorithm. Hence it is not shown in subsequent performance graphs.

Voxelization, Voxel Detailing, and Mesh Generation

The last three stages of the algorithm are much slower than the grammar interpretation.

Figure 4.2 shows the range of combined running times for voxelization, voxel detailing, and mesh generation, plotted against the voxel grid sizes.

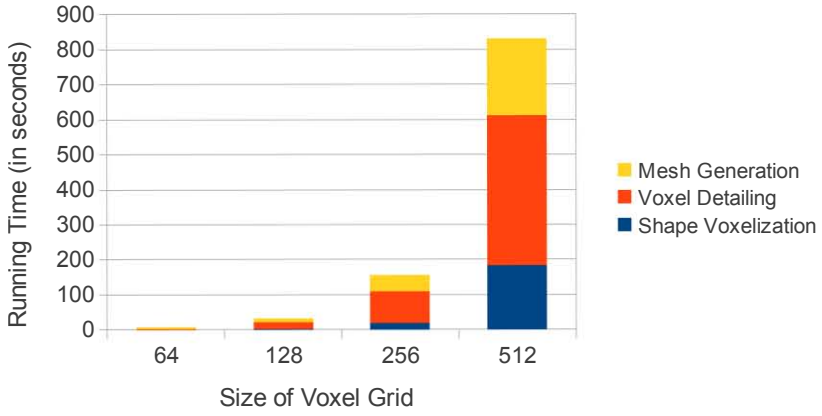


Figure 4.3: A cumulative graph of the average times taken for our algorithm to run on 36 different inputs, covering a range of complexities. Shape grammar interpretation is not shown as it was negligible compared to the other three stages. This graph also shows the relative durations of the different stages, with voxel detailing being the longest by far. A more nuanced breakdown of these times can be gleaned from the box and whisker plots in figures A.1 through A.3.

The most obvious observation from the graph is that voxel grid resolution is a major factor in the running time of the algorithm. The slowest, median, and fastest running times all grow at a rate that is somewhere between quadratic and cubic in the sizes of the voxel grid.

This is to be expected, as the running time of each of the algorithm stages is directly proportional to either the number of voxels in the grid, or the number of surface voxels. These scale cubically and quadratically, respectively, with the resolution of the grid.

The overall collective shape output by the shape grammar can affect the running time because the shapes are scaled into the cuboid voxel grid. If the overall shape (or bounding box) is roughly cuboid, then the scaled shapes will collectively be a closer fit to the cuboid voxel grid, and take up more of it than they would if their overall bounding box was rectangular. Taking up more of the grid means more voxels are used, and the running time will be increased. Hence, the rough overall dimensions of the shape grammar output can affect the running time.

These two parameters are the major factors for determining the running time of our implementation. Others, such as the number of detailing iterations, do also influence the running time, but their impact is minor compared to that of the grid resolution and overall collective shape of the shape grammar's output.

The second observation is that there is a huge range of running times, from a few seconds to roughly 40 minutes. The size of the voxel grid is the biggest factor, but the shape grammar output and number of iterations required for detailing also play a significant role.

Figure 4.3 shows the average times across all of the inputs, broken down into the different stages. This allows us to compare the running times of the

individual stages to each other.

We can draw a number of observations from this graph. Firstly, the voxel detailing stage is, on average, the longest running. This is expected, as it requires the execution of a section of arbitrary code for each surface voxel, potentially multiple times.

Secondly, the running times of the voxelization stage of the algorithm is approximately cubic in the size of the voxel grid. This is expected, as the running time of our method of voxelization is directly proportional to the number of voxels in the grid, which scales as the cube of the grid's resolution.

Because the detailing and mesh creation stages of the algorithm only operate on the surface voxels, we should expect their running time to be quadratic in the size of the voxel grid. The volume of an object will increase cubically when scaled along all three axes, but its surface area only increases quadratically.

However, the fact that many of the surface detailing rule sets call for multiple passes (as high as 25, in our data set) means that, on average, the surface detailing times appear to scale faster than quadratically with increasing voxel grid size.

This is a constant factor specific to multi-pass detailing however, and asymptotically, the quadratic rate of growth would dominate.

Figures A.1 to A.3 in Appendix A present a more nuanced look at the running times of the individual stages of the algorithm. These allow us to get a better idea of the range of relative times for the voxelization, voxel detailing, and mesh generation stages.

Apart from reinforcing our earlier observations about running times from the other two graphs, we can also note some other trends. Across the different voxel grid sizes, the ranges of running times for the voxel detailing and mesh creation stages stay very similar relative to each other.

However, the running times for voxelization grow significantly as the size of the voxel grid increases, with some cases taking even longer than mesh creation for large voxel grids. This is expected because, as mentioned above, the method we use for voxelization requires iterating over each voxel in the shape to be voxelized. This total number of voxels grows cubically with increases in the voxel grid size.

In contrast, the voxel detailing and mesh creation stages only operate on surface voxels. The number of surface voxels grows quadratically with increases in the size of the voxel grid. These different rates of growth explain how the voxelization running times increase so rapidly, relative to the other two stages.

4.1.3 Memory Usage

Our main concern regarding the memory usage of our program is the peak memory usage, i.e. the largest amount of memory that the program uses at any one point. This is the metric that we measured during testing.

Memory usage data was gathered by reading the Linux '/proc/self/status' file, which contains details of the peak and current memory usage of the program reading the file. This is not a completely accurate measurement of the program's memory usage, since some memory pages may be counted more than once, but it provides sufficient accuracy for our purposes. Additionally, it is conservative, and so will not underestimate memory usage, which is desirable for measuring worst cases.

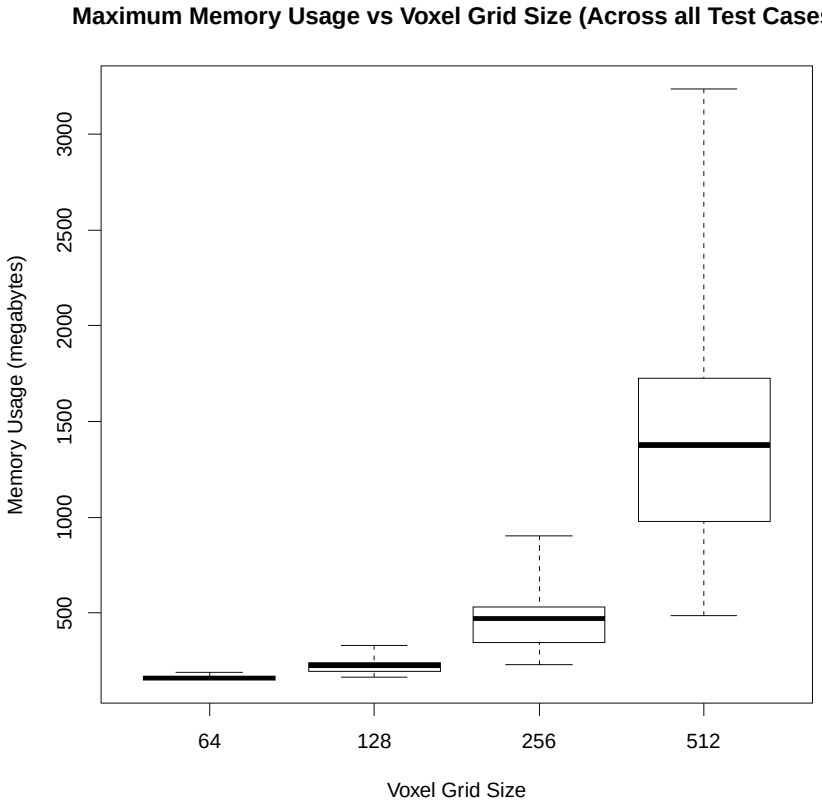


Figure 4.4: A box and whisker plot of the peak memory usage of our program across all test cases, plotted against the size of the voxel grid. The pattern is very similar to that of the running times, as memory use increases sharply with grid resolution.

Exact memory usage can be determined by running the program under a memory debugger, such as Valgrind[1], but these are extremely slow for memory intensive programs, such as ours. Because a conservative approximation of memory use is sufficient for our purposes, we opted not to use a memory debugger.

Figure 4.4 shows a box and whisker plot of the distribution of peak memory usage vs. the size of the voxel grid, measured across all of our test cases. It shows the order-of-magnitude range of memory requirements possible for the program. Although some test cases required more than three gigabytes of memory, the majority of them needed less than 1750 megabytes.

When measuring peak memory use, we once again considered the peak memory usage of the program for each stage of the algorithm individually. This meant that the peak memory usage of the program was recorded after each stage of the algorithm had been run.

[1]Software website: `valgrind.org`

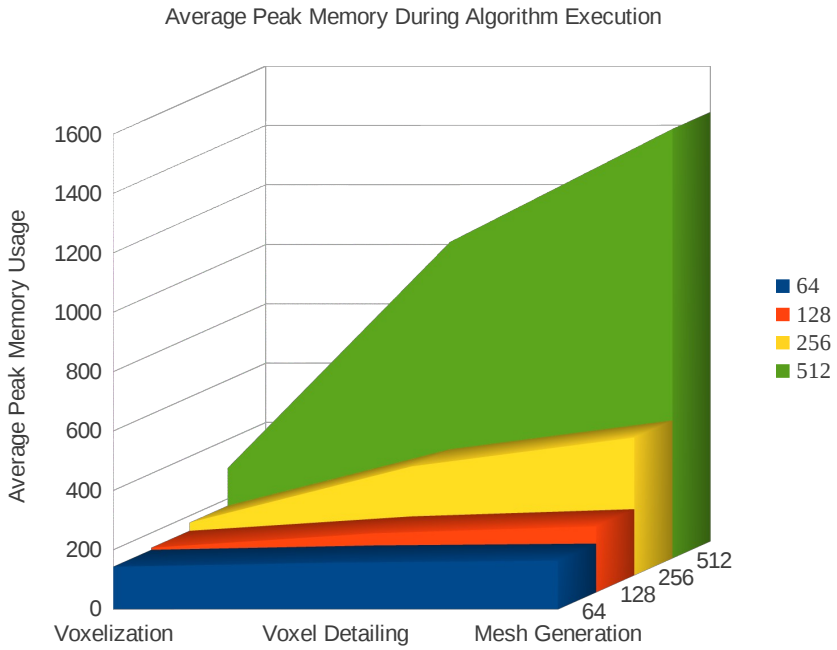


Figure 4.5: This diagram shows the average peak memory used by our implementation, after each stage of the algorithm has been run, for each voxel grid size. As can be seen, each stage of the algorithm requires more memory than the previous stage, with the increase being proportional to the grid size.

Figure 4.5 shows a breakdown of the average peak memory after each stage of the algorithm, across all four voxel grid sizes.

As can be seen, the average peak memory usage grows extremely quickly with increases in voxel grid size, particularly for voxel detailing and mesh generation.

The major factor affecting memory usage is the number of active voxels in the grid. This is because the amount of storage required for the octree, shape tags, detailing tags, and the mesh output is directly proportional to the number of voxels that are processed and used in the grid. Hence, factors that affect this will cause the biggest changes in memory requirements.

Primary among these factors is the size of the voxel grid. As is seen in Figure 4.4, the memory requirements grow quickly with increases in the voxel grid size.

Also, as with the running time of the algorithm (see Section 4.1.2), the collective shape of the shape grammar's output also affects the memory requirement. Because the shapes are scaled into a cuboid space, and the closer they collectively are to a cube, the tighter the fit into the octree's space, and hence the more voxels are used.

Another big factor in the memory requirements is the number of surface voxels in the voxel grid. This is in fact, the biggest contributor to the algorithm's memory requirements in our implementation. As mentioned in Section 3.7.3, our implementation uses a simple method to store detailing tags, which is not optimal for saving memory. This means that when a detailing rule set

assigns multiple detailing tags to many of the voxels in the grid, the memory requirements quickly ramp up.

The number of iterations does not noticeably affect memory use. Because the amount of active voxels in the grid is not proportional to the number of iterations, it has very little effect on our implementation's memory requirements.

The worst case scenario for memory usage among all of our test cases was one of the space station shown in figure 4.7, generated with a grid size of 512. In addition to having a very cuboid overall shape, it also featured a very large surface area. These factors, combined with every voxel having at least one detailing tag, and a high grid resolution, produced the perfect storm of conditions, requiring 3.2 gigabytes of memory, 700 megabytes more than the next highest case.

Looking at the memory requirements for the individual stages of the algorithm in Figure 4.5, in all cases, each stage of the algorithm required more memory than the previous stage. This is to be expected, as each stage requires all information from the previous stage, and adds new information. The voxelization stage only requires an octree, but the voxel detailing stage requires that detailing information be added the octree. Finally the mesh creation stage requires the full, detailed octree, but then needs to store the mesh it creates as well.

Figures A.4 and A.5, included in Appendix A, show a more detailed breakdown of the range of peak memory values measured during testing, and allow a better comparison of the relative amounts of memory required for the different stages of the algorithm.

As can be seen from the graphs, the relative size and position of the boxes in the plots remain very similar across all voxel grid sizes. This means that the majority of our test cases shared a very similar pattern of peak memory usage.

This is useful, as it would theoretically allow us to estimate the peak memory requirements of an input, given the size of the voxel grid. The ability to do this is helpful, especially when producing high quality models at large voxel grid sizes.

In all cases, voxelization requires the least memory. This is expected, as octrees are highly efficient at storing voxel data. From there, voxel detailing requires significantly more memory. This is because the python interpreter is loaded into the program for running the detailing rules, and the output of the rules has to be stored for each surface voxel in the grid. Depending on the size of the grid, and rule output, this can be a large amount of memory. As mentioned in Section 3.7.3, there is scope for decreasing the memory required for this stage, through optimizations.

Mesh generation requires a further significant increase in memory. In most cases (those contained within the boxes of the box and whisker plots), the increase is not as pronounced as that seen when moving from voxelization to voxel detailing. This extra memory is required to store the output mesh, which can be of substantial complexity and size, especially at the larger resolutions.

4.2 Variation Testing

Variation testing was performed to test the variety of models that could be generated, with all models conforming to a similar style, when using our shape

Figure 4.6: A selection of castles produced by our voxel-space shape grammar implementation. These were all generated from the same grammar, with different random seeds supplied during interpretation. Note how they share commonalities, but the exact details (such as wall heights, dimensions, and number of towers) vary between the models.

grammar extensions (as discussed in Section 3.2.3).

It is, of course, difficult to show that creating shape grammars to exhibit this variation behaviour is always possible or easy, but we have chosen a selection of shape grammars/detailing rule set combinations that produce a range of types of models (buildings, vehicles, and space stations). Each of these combinations can successfully create a variety of models in a common theme, using stochastic rules.

Figure 4.6 shows some of the outputs of our castle generating shape grammar, used with a mossy-brick surface detailing rule set. The actual Python source code of the shape grammar that created the castles is attached in appendix B, Section B.1. As can be seen, almost all parameters of the castle are randomly set. The height and dimensions of the walls, the number of towers and their layout, the position and size of the keep, and the layout of the wall gates are all stochastically chosen from a defined range. Because the rules all operate with relative coordinates of parent shapes, the shapes will not overlap or interact in unwanted ways unless expressly instructed to. This does require some additional work on the part of the grammar designer, but not much more than would be required to create the basic shape grammar.

Some slightly more complex examples are shown in figure 4.7. Again, the

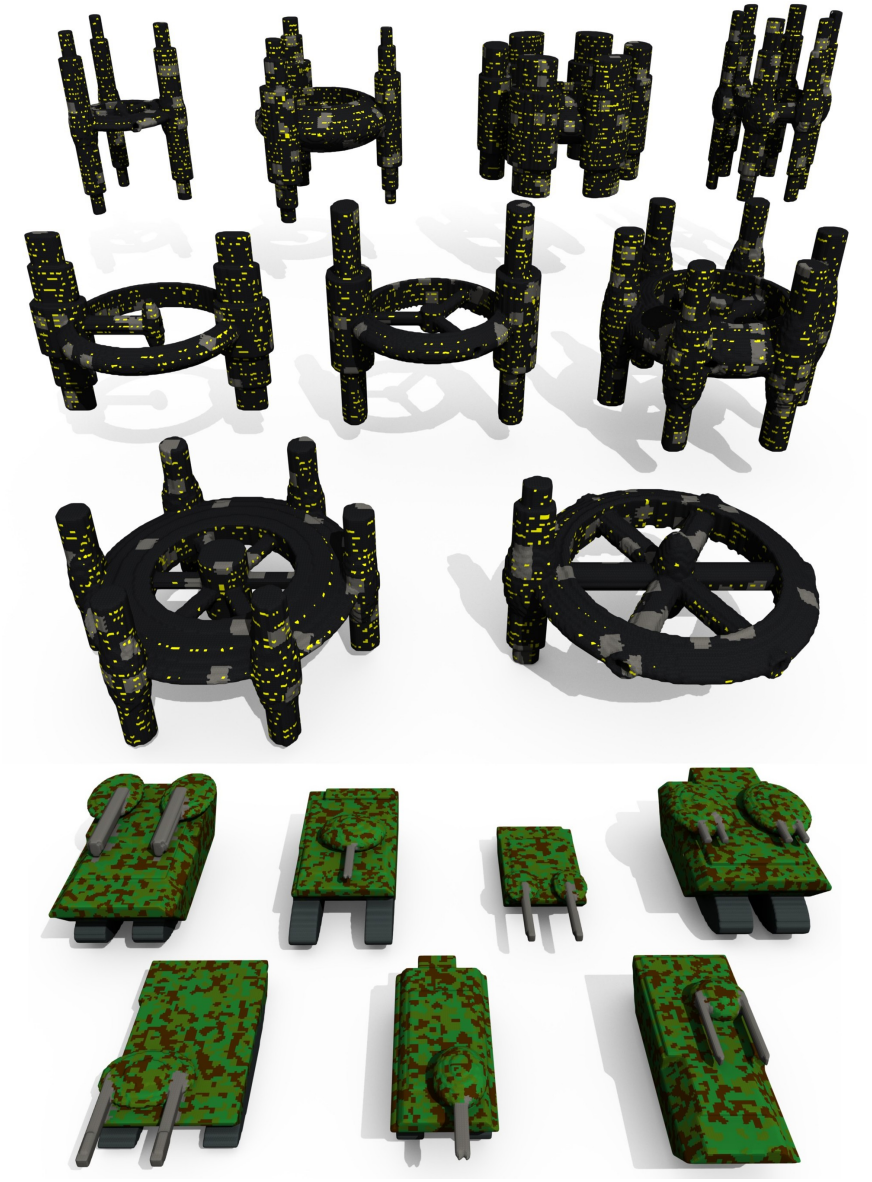


Figure 4.7: A selection of tanks and space stations produced by two of our shape grammars, using randomized parameters in their rules. This shows how a single grammar can produce multiple models in the same style.

shape grammars that generated these models are included in appendix B (sections B.2 and B.3). The rotationally symmetric space stations are detailed using a rule set that creates a base layer of dark metal and then overlays lighter patches, as well as windows on the vertical surfaces. This shape grammar makes heavy use of the symmetry operators, often with randomized parameters, as demonstrated by the varying number of spires and ring spokes. As can be seen from the space stations, the symmetry operators can be incorporated into the stochastic rules, with good results.

Finally, Figure 4.7 shows the output of our tank shape grammar/detailing rule set combo. Once again, most parameters are randomized, resulting in a wide range of outputs that all share a consistent style, both in geometry and detailing.

These examples also illustrate another strength of our shape grammar extensions: the ability to easily apply the same detailing rule set to different models and achieve a consistent output across them, as discussed in Section 3.4.1. In some cases, this is only possible if the detailing rule set is not dependent on tags from the shape grammar, but generally across models created from the same stochastic shape grammar, a detailing rule set that relies on shape tags will work correctly.

Both of these situations are shown our variation testing examples. The detailing rule sets used on the castles in figure 4.6 and that used on the space stations in Figure 4.7 are not reliant on tags assigned by the shape grammar. However, the rule set used to detail the tanks in Figure 4.6 is. Because the tank models are created from the same shape grammar, the tags assigned to their shapes (and hence voxels) are all used in a consistent manner, and the tank detailing rule set will work correctly for all of them.

The detailing rule sets for the castles and space stations do not make use of any shape tags, but this does give them the benefit of broader applications. They could be applied to any voxel grid, and the results would be consistent with what was expected (the voxel grid would be covered in a brick texture with patches of moss, or a metallic texture with patches of lighter metal and windows on vertical surfaces). However, if the tank's detailing rule set was applied to an arbitrary voxel grid, the result would be entirely covered in the camouflage pattern, because none of the voxels would be tagged as treads or gun barrels.

In short, a detailing rule set that relies on tags will be able to create more customized outputs tailored to the models intended for it, but at the cost of working properly on arbitrary voxel grids that may lack the shape tags.

4.3 Output Range Testing

In this type of testing we attempted to generate a wide variety of models, covering a range of complexity in both the geometric structure and surface detailing. The objective was to demonstrate that our shape grammar extensions add additional expressive power to conventional shape grammars and do not restrict previously existing capabilities at all.

A selection of these generated models are shown in figures 4.6 to 4.14.

Figures 4.9 and 4.10, contain models demonstrating cellular automata (CA) patterns. The two rectangles in figure 4.10 demonstrate 25 iterations of the well-known CA patterns rule 30 (a chaotic pattern that occurs in nature) and

Figure 4.8: A model of a rotationally symmetric space station detailed with windows and bright metal patches. Note the hollowed out areas created in the wings using subtractive shapes.

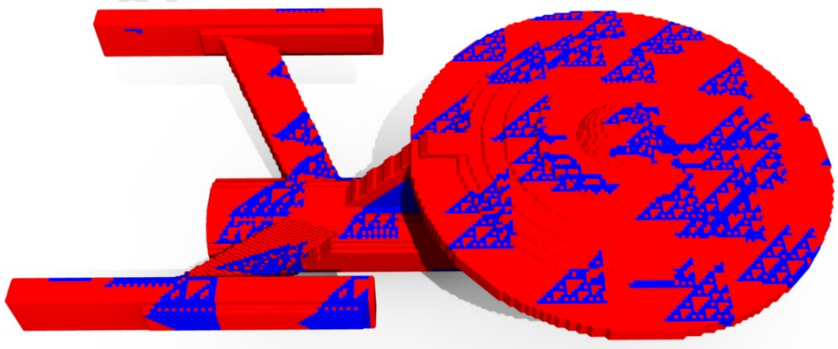


Figure 4.9: A model of the enterprise, from Star Trek, detailed in red and blue with the cellular automata pattern Rule 110.

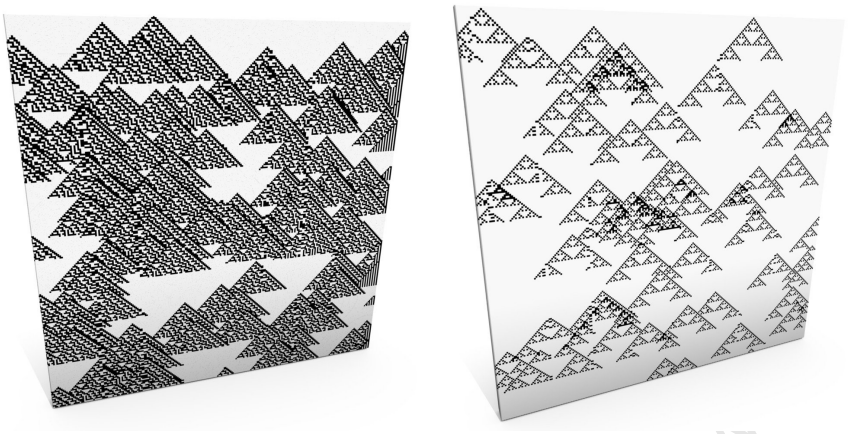


Figure 4.10: Two simple rectangles detailed with the cellular automata patterns Rule 30 (left), and Rule 90 (right).

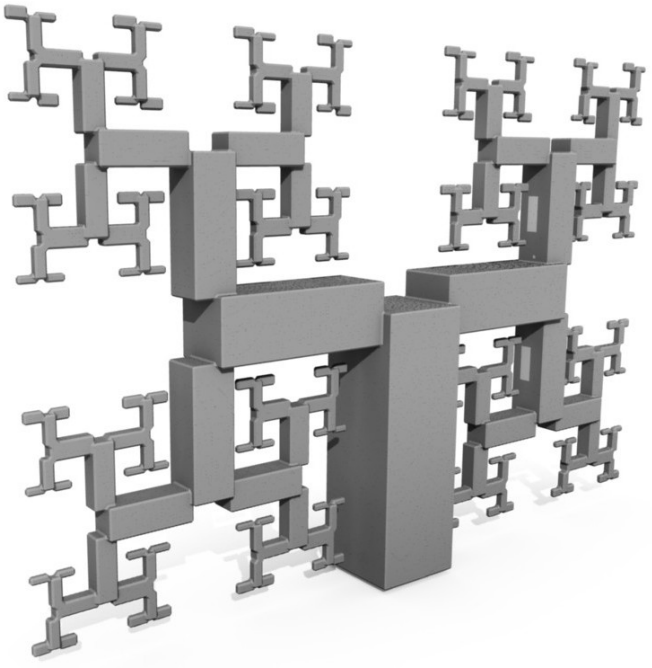


Figure 4.11: A model of a simple fractal tree, detailed with a plain grey colour.

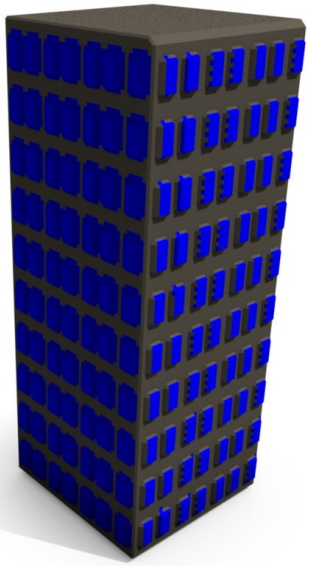


Figure 4.12: A simple skyscraper model, generated using a basic split grammar.

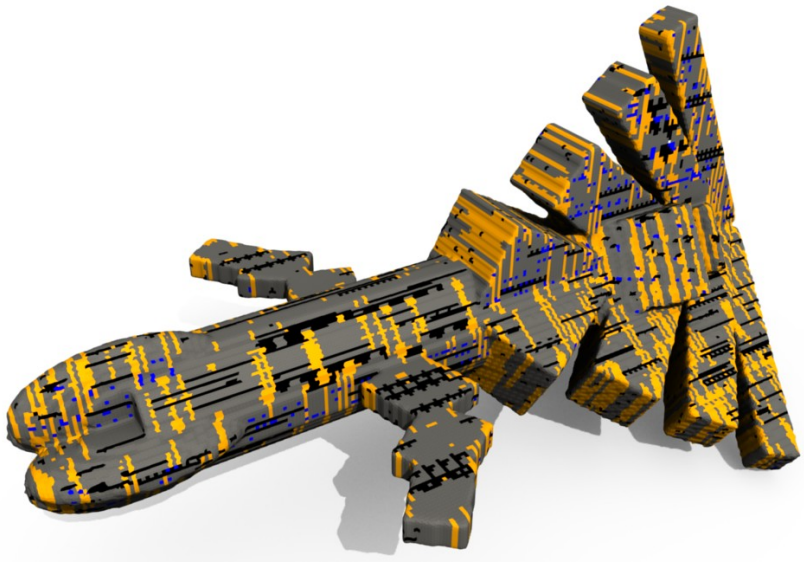


Figure 4.13: A spaceship model, created with multiple symmetry rules. It is detailed with a pattern of randomized horizontal and vertical 'tiger stripes', and blue windows on the vertical faces of the ship.

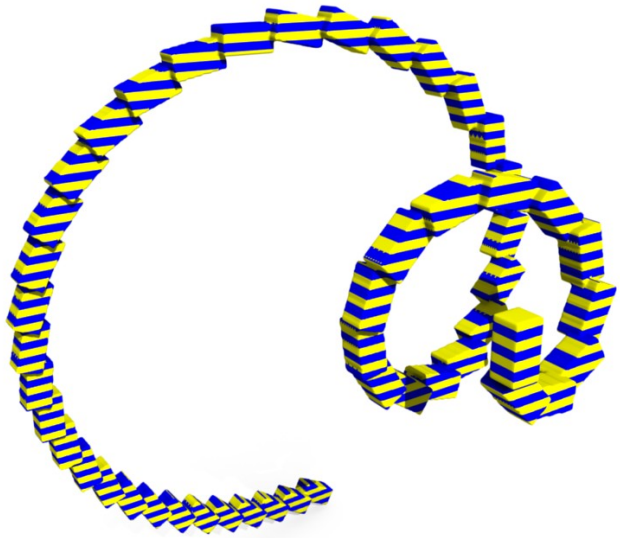


Figure 4.14: A simple model of a spiral, detailed using a global horizontal stripe pattern.

rule 90 (which produces a 2D Sierpinski triangle). Figure 4.9 shows an example of the CA pattern, rule 110, being applied in a 3D context on a model of the Enterprise. There is no way to create consistent rule-based detailing patterns such as these, across an entire model with conventional mesh-based shape grammar implementations.

In figures 4.11 and 4.12 we also display a fractal tree and a model of a simple skyscraper-style building, both created by our implementation. These models show that with our extensions, we are still able to generate models using L-systems (for the tree), and split grammars (for the building), and have not lost any generative ability.

The model of a spiral in Figure 4.14 was created from overlapping rectangles, with a detailing pattern based on the y-coordinate of the voxels. Note that, in spite of the rectangles overlapping, there are no seams in the detailing and the pattern is not disrupted. Creating such a global detailing pattern based on the coordinates of pieces of geometry would be difficult to impossible to do with conventional shape grammars. Other examples of this global detailing with no detailing seams between joined geometric primitives can be seen in the castles of figure 4.6 and the space stations of figure 4.7.

Another good example of the power of context-sensitive detailing is shown in the space stations in figures 4.7 and 4.8. The detailing rule set specifies that windows should only be created on voxels whose normalized normals have an X component of 0.8 or greater. This means that windows are only created on surfaces which are nearly vertical. This level of context sensitivity for the surface detailing of models cannot be achieved using conventional shape grammar implementations.

Staying with the space station model in Figure 4.8, it shows how powerful

subtractive shapes can be. The rotational wings start as ellipsoids, but then have subtractive rectangles applied, to carve out the indentations on their sides. Then, a subtractive cylinder is used to hollow out the holes at the end of the wings and further rectangles used to make docking bays in the hollowed out area. These subtractive capabilities allow us to easily create an interestingly shaped wing, whereas, if we wanted to create the same wing with a conventional mesh based grammar, we would need far more rules, and would have issues with texture seams between shapes.

The space ship decorated with 'tiger stripes' of black and orange in Figure 4.13 shows a good example of a multi-pass detailing pattern. The stripes are randomly generated and grown to overlap each other across 15 passes. A conventional shape grammar implementation would be unable to create dynamic multi-pass detailing such as this.

These examples illustrate the major benefits that our extensions bring to shape grammars, and confirm that they can increase their generative range beyond what is possible with conventional implementations.

4.4 Limitations

In spite of the increased generative power of our shape grammar extensions, there are still some caveats to their use. During the process of testing the output range of our implementation, we identified two limitations to our voxel-space shape grammar algorithm that could restrict its potential applications.

4.4.1 Voxel Grid Size Requirements

In order to obtain a high quality model from a voxel data set, the set must be created at a high resolution. This is because the discrete nature of a voxel grid causes "blocky" visual artifacts in meshes produced by the meshing algorithms. These artifacts are especially visible for low resolution voxel grids (sizes of 128 or less), but are much less pronounced for high resolution voxel grids (size 256), and virtually negligible for very high resolutions (sizes of 512 or more). Hence, to generate models for production use, a large voxel grid size should be used. When experimenting or designing the model, a lower resolution should be sufficient though.

An example of this issue can be seen in figure 3.12, where a model of the Enterprise was intentionally created at a low resolution to better illustrate how the different stages of the algorithm work. The unsmoothed mesh is visibly bumpy, but those artifacts are removed after smoothing (although the smoothing does distort texture details as mentioned in Section 3.6).

Mesh smoothing as a post-process can help with the problem of removing the artifacts from the mesh, but it is not sufficient on its own. However, artifacts can be more than adequately dealt with by creating the model at a larger voxel grid resolution, and then applying mesh smoothing.

Unfortunately, the higher the voxel grid resolution, the slower the algorithm, especially the voxel detailing stage: each surface voxel in the model must be detailed, and the number of surface voxels is quadratic in the dimensions of the voxel grid.

Consequently, when generating high quality models for production use, they must be generated at a relatively high resolution, and mesh smoothing is recommended. As a result, their generation is likely to be slow, and may require out-of-core processing (which can easily be implemented with an octree).

4.4.2 Curves and Very Fine Detailing

A second limitation is that texturing at a per-voxel level may be insufficient for certain types of detailing, in certain cases.

This is once again to do with the discrete nature of the underlying voxel grid. The detailing process is also bound by this discreteness, and this can cause issues for very fine details and curves.

The detailing rule set cannot create details at a sub-voxel scale, because it cannot operate at a level lower than a single voxel. This means that it may be impossible to create very fine detail without increasing the voxel grid size, and possibly modifying the detailing rule set to handle the changed voxel grid size.

Related to this, curves created at small scales by the detailing rule set may be visibly "blocky". This is for the same reasons as described in the previous section (4.4.1), but in the context of the surface, as opposed to the structure of the voxel grid itself.

For example, an elaborate spiral design with very fine curved detail on the side of a spaceship would almost certainly run into sampling issues if created with a detailing rule set.

A possible solution to this problem would be allowing the addition of decal textures to the final version of the mesh. The detailing rule system could be extended to allow the specification of how arbitrary textures could be projected onto parts of the generated mesh. These decals would then replace, or blend with, the textures assigned at a voxel-level and display detail that could not be created within the detailing rule set framework. However, we did not implement this solution, and leave it to future work.

Alternatively, the desired detailing features could be added by hand, using external mesh editing software.

4.5 Summary

The results of our testing are very encouraging. There are some areas of minor concern, but our shape grammar extensions do bring new functionality to the procedural design process, and all indications are that our algorithm is completely viable.

In terms of performance, the running times and memory usage do scale very quickly with the voxel grid size, but as mentioned in Sections 3.7.3 and 3.4.2, there are optimizations that could reduce these costs significantly.

Nevertheless, the running times are reasonable for an offline generation workflow (even at large voxel grid sizes), and the memory requirements are well within reach of medium- to high-end computers, where 4 gigabytes of RAM is common.

Additionally, in practice, users can prototype their grammars and rule sets at lower voxel grid resolutions and then do off-line generation of a high resolution

model for actual use. This means the long running times for large models will not significantly disrupt work-flow.

The variation and output range testing both indicate that our extensions increase the generative range of shape grammars without limiting any existing features. We are able to produce a wide range of models that exploited the features of our extensions to produce content that would be much more difficult, or impossible, to create with a conventional shape grammar implementation.

There are limitations to our shape grammar extensions, mostly linked to the discrete nature of the underlying voxel grid. However, these limitations are not critical, and none of them should be problematic in the majority of cases.

Overall, the results of our testing provide positive answers to our original research questions, which indicates that we have met our success criteria. Hence we conclude that our algorithm meets the original design goals: It is viable for real-world use, and successfully addresses the weaknesses in conventional shape grammars that we originally identified.

Chapter 5

Conclusion

This thesis has covered novel extensions to shape grammars, aimed at addressing two deficiencies that occur when using mesh-based representations of shapes.

Firstly, there are issues with complex 3D models created by means of Boolean (CSG) operations. Boolean geometry is difficult to perform with meshes as it requires clipping of edges, removal of vertices, and the creation of new edges and vertices to adequately join shapes. Beyond these geometric difficulties, there are still challenges with texture seams and issues of robustness.

Secondly, there are difficulties with sub-, or trans-, shape detailing using mesh-based shape grammars. Mesh detailing is done by texturing the faces of shapes with 2D images. As a result, it is difficult to have adjoining detailing that spans multiple faces or operates at scales smaller than a single mesh face.

Our extensions address these issues by voxelizing the shapes output from the shape grammar. This allows more robust Boolean geometry operations (as these operations are much simpler to perform accurately with voxels) and a new per-voxel, rule-based approach to detailing the surface of generated models, where each voxel can be detailed independently of those around it, allowing fine-grain control and detailing elements to easily span shapes.

There is an extensive body of research on shape grammars and voxels in procedural generation, but to the best of our knowledge, our extensions are novel.

Our algorithm operates in five stages. In the first stage, the shape grammar is interpreted to produce an output set of shapes. This step is very similar to the way in which conventional shape grammars operate, and ideas and enhancements from those may be used.

The second stage voxelizes the shape output from the first stage into a voxel grid. There are several concerns to address in this stage, such as storing the voxel grid efficiently and the order of shape addition. Metadata can also be associated with voxels, to aid later detailing.

The voxels are detailed in the third stage of the algorithm. This is done via a detailing rule set, which operates on each voxel individually, using contextual information.

Stage four involves the conversion of the voxel grid into a mesh, suitable for rendering. This is done via the marching cubes algorithm.

The fifth stage, mesh smoothing, is optional but does generally increase the visual quality of the produced mesh, and is recommended for most situations.

Our research also included a comprehensive suites of tests to validate our shape grammar extensions and ensure that they are suitable for real-world use. This testing produced positive results, with only a few qualifiers about using our algorithm in certain situations.

Performance testing was aimed at ensuring that our algorithm's execution time and memory requirements are acceptable (it can achieve reasonable performance on high-end hardware). The results show that our algorithm is slower and more memory intensive than conventional shape grammar implementations, but not outside acceptable limits. Specifically, the running time complexity is a low-order polynomial, and the memory requirements are within the ability of current medium to upper end hardware. In light of the increased generative range our algorithm allows, these increased requirements are acceptable.

In addition, we have outlined a number of possible optimizations that would decrease memory usage and running times. These would be a benefit to normal generation, but could possibly also allow real-time, 'on the fly' generation of models.

From a practical perspective, we recommend our shape grammar extensions only be used when their increased generative range is required. For the generation of simpler models, conventional mesh-based shape grammars would be sufficient, and users could switch to using our advanced extensions when necessary to generate more complex models.

The ability to produce a range of outputs from a single algorithm is an important capability in most areas of procedural generation, including shape grammars. Our evaluation included variation testing to ensure that we could use stochastic shape grammars to produce a range of models from a single shape grammar. The results of this testing indicate that our voxel-space extensions fully support stochastic grammars, as we were able to to generate groups of models that shared a similar style and look, but noticeably differed in their specifics.

The final type of testing, output range testing, was undertaken to verify that our enhancements allow an increased generative range. In this we created grammars targeted at using our extensions to create models that would be much more difficult to produce with a mesh-based shape grammar. The resultant models exhibited both geometric and surface detail features that would be extremely challenging to produce under a conventional grammar framework, such as components whose easy creation relies heavily on CSG and complex cellular automata patterns that span the entire model seamlessly. As outlined in the introduction chapter, model aspects such as these are challenging to create under a conventional mesh-based shape grammar system.

5.1 Discussion

The success of our research, and the voxel-space shape grammars it presents, can be determined by the answers to the three pivotal research questions we originally laid in section 1.7. We examine the answers to these questions here.

- **Question 1:** Does using a voxel representation of shapes, instead of a mesh, solve the limitations of shape grammar implementations as identified? Can our voxel-space shape grammar extensions produce models that are difficult or impossible to produce with meshes?

As Section 4.3 shows, this research question is answered in the affirmative. Figures 4.6 to 4.14 show a range of models, produced by our implementation, that conventional shape grammar implementations would struggle to create. This is due to features such as: complex rule-based detailing, nested Boolean geometric operations, overlapping shapes with detailing that moves between them, and having the same complex detailing pattern successfully applied to multiple models.

- **Question 2:** What limitations, if any, arise from working in a voxel-space? Further, are any of these limitations severe enough to make our algorithm nonviable?

The main limitations that our extensions bring are increased running times and higher computational resource requirements. There are also potential requirements for the minimum resolution of the voxel grid, to avoid aliasing issues in the generated mesh. However, none of these make our algorithm nonviable.

- **Question 3:** Are our extension computationally efficient enough for real-world use?

In spite of the greater running times and memory usage, our shape grammar extensions are still efficient enough for offline real-world use, especially if the optimizations that we were unable to implement due to time constraints are incorporated. Resource usage is higher than in conventional shape grammar implementations, but powerful computers that can handle this more intensive usage are the norm in content generation environments, so the increased requirements are not an obstacle to real-world use.

By examining the results of our testing and evaluation, it can be seen that our novel shape grammar extensions add new functionality to shape grammars, without losing existing capabilities, and significantly increase the range of achievable content.

They also meet the three major requirements that we originally set out for them in our research questions. Their computational performance is within acceptable bounds, they successfully solve the two deficiencies we originally identified in conventional implementations, and they do not add any limitations that cannot be overcome.

These testing results, and the answers to our research questions, show that using voxel representations of shapes in shape grammars provides a new method of generating models, that overcomes the limitations we identified in mesh-based shape grammars. Hence, our shape grammar extensions could be used in content production environments for the creation of complex models, that are currently difficult to generate with conventional shape grammars.

The fact that no major limitations are introduced by our extensions, and that their computational performance is within acceptable bounds, shows that there are no issues preventing the use of our extensions, and the benefits that they bring, in a real-world production environment.

5.2 Future Work

There are many avenues for future work on our voxel-space shape grammar extensions.

To begin with, the per-voxel detailing stage could be expanded to address the interior of generated models. In our work, we have only performed detailing on the surface voxels of the model, but the method could be extended to detail the interior voxels. This would allow the creation of details such as rooms inside generated buildings, but would increase the time needed to generate the model significantly.

Post-processing could also be applied to the voxel grid before it is detailed. A separate group of rules could be provided to add, remove, and tag voxels before voxel detailing took place. These rule sets could create detail corresponding to damage, wear and tear over time, growth of mold, and more. This would be similar to the voxel detailing, except that this rule set could modify voxels themselves instead of only their detailing tags.

As our algorithm currently stands, it is not suitable for real-time use. The memory requirements and long running times mean it could not be used to procedurally generate models 'on the fly'. However, because real-time use was never one of our design criteria, this is acceptable for our research.

Still, it is possible that a highly optimized implementation, possibly using graphics hardware acceleration and with restricted output scope, could generate low resolution models in real-time.

It should be possible to develop a hybrid system that combines our voxel-based approach with a conventional mesh technique. Under this scheme, users could apply our voxel extensions only where required. Computation costs would be decreased, without losing the benefits of the voxel extensions.

The voxel detailing rules could be extended to operate at multiple resolutions of the voxel grid. Octree nodes could easily be coalesced to form a lower-resolution version of the model, to which the rule set could then be applied. This would allow the creation of large-scale detailing initially, working down to finer details as the rules were run at higher resolutions. Additionally, it would be an easy way to produce lower level-of-detail models.

Finally, there is scope for a user-friendly interface that allows easy use of our extensions to shape grammars. Such an interface would assist users in designing surface detailing rules, and handling complex Boolean geometry operations, especially those in which the order of shape addition is important.

Bibliography

[1] Barnes, B. 'avatar' is no. 1 but without a record. `http://www.nytimes.com/2009/12/21/movies/21box.html`, December 2009 (Accessed 2012-01-12).

[2] Baxter, R., Crumley, Z., Neeser, R., and Gain, J. Automatic addition of physics components to procedural content. In *Proceedings of the 7th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa* (New York, NY, USA, 2010), AFRIGRAPH '10, ACM, pp. 101–110.

[3] Bokeloh, M., Wand, M., and Seidel, H.-P. A connection between partial symmetry and inverse procedural modeling. In *ACM SIGGRAPH 2010 papers* (New York, NY, USA, 2010), SIGGRAPH '10, ACM, pp. 104:1–104:10.

[4] Cai, K., Liu, Y., Wang, W., Sun, H., and Wu, E. Progressive out-of-core compression based on multi-level adaptive octree. In *Proceedings of the 2006 ACM international conference on Virtual reality continuum and its applications* (New York, NY, USA, 2006), VRCIA '06, ACM, pp. 83–89.

[5] Clarenz, U., Diewald, U., and Rumpf, M. Processing textured surfaces via anisotropic geometric diffusion. *Image Processing, IEEE Transactions on 13*, 2 (feb. 2004), 248 –261.

[6] Drebin, R. A., Carpenter, L., and Hanrahan, P. Volume rendering. *SIGGRAPH Comput. Graph. 22*, 4 (June 1988), 65–74.

[7] Eichhorst, P., and Savitch, W. J. Growth functions of stochastic lindenmayer systems. *Information and Control 45*, 3 (June 1980), 217–228.

[8] Fang, S., and Liao, D. Fast csg voxelization by frame buffer pixel mapping. In *Proceedings of the 2000 IEEE symposium on Volume visualization* (New York, NY, USA, 2000), VVS '00, ACM, pp. 43–48.

[9] Gilbert, E. G., Johnson, D. W., and Keerthi, S. S. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal of Robotics and Automation 4*, 2 (1988), 193–203.

[10] Greeff, G. Interactive voxel terrain design using procedural techniques. Master's thesis, University of Stellenbosch, 2009.

[11] Greuter, S., Parker, J., Stewart, N., and Leach, G. Real-time procedural generation of 'pseudo infinite' cities. In *Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia* (New York, NY, USA, 2003), GRAPHITE '03, ACM, pp. 87–ff.

[12] Guéziec, A., and Hummel, R. Exploiting triangulated surface extraction using tetrahedral decomposition. *IEEE Transactions on Visualization and Computer Graphics 1* (December 1995), 328–342.

[13] Hadap, S., Eberle, D., Volino, P., Lin, M. C., Redon, S., and Ericson, C. Collision detection and proximity queries. In *ACM SIGGRAPH 2004 Course Notes* (New York, NY, USA, 2004), SIGGRAPH '04, ACM.

[14] Hahn, E., Bose, P., and Whitehead, A. Persistent realtime building interior generation. In *Sandbox '06: Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames* (New York, NY, USA, 2006), ACM, pp. 179–186.

[15] Hecker, C., Raabe, B., Enslow, R. W., DeWeese, J., Maynard, J., and van Prooijen, K. Real-time motion retargeting to highly varied user-created morphologies. In *ACM SIGGRAPH 2008 papers* (New York, NY, USA, 2008), SIGGRAPH '08, ACM, pp. 27:1–27:11.

[16] Hultquist, C., Gain, J., and Cairns, D. Affective scene generation. In *Proceedings of the 4th international conference on Computer graphics, virtual reality, visualisation and interaction in Africa* (New York, NY, USA, 2006), AFRIGRAPH '06, ACM, pp. 59–63.

[17] Ilčík, M., Fiedler, S., Purgathofer, W., and Wimmer, M. Procedural skeletons: kinematic extensions to cga-shape grammars. In *Proceedings of the 26th Spring Conference on Computer Graphics* (New York, NY, USA, 2010), SCCG '10, ACM, pp. 157–164.

[18] Itoh, T., Miyata, K., and Shimada, K. Generating organic textures with controlled anisotropy and directionality. *IEEE Comput. Graph. Appl. 23* (May 2003), 38–45.

[19] Johansson, G., and Carr, H. Accelerating marching cubes with graphics hardware. In *Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research* (New York, NY, USA, 2006), CASCON '06, ACM.

[20] Jones, T. R., Durand, F., and Desbrun, M. Non-iterative, feature-preserving mesh smoothing. In *ACM SIGGRAPH 2003 Papers* (New York, NY, USA, 2003), SIGGRAPH '03, ACM, pp. 943–949.

[21] Kaufman, A., Cohen, D., and Yagel, R. Volume graphics. *Computer 26*, 7 (jul 1993), 51 –64.

[22] Kelly, G., and McCabe, H. A survey of procedural techniques for city generation. *Institute of Technology Blanchardstown Journal* (2006).

[23] Kopf, J., Fu, C.-W., Cohen-Or, D., Deussen, O., Lischinski, D., and Wong, T.-T. Solid texture synthesis from 2d exemplars. In *ACM SIGGRAPH 2007 papers* (New York, NY, USA, 2007), SIGGRAPH '07, ACM.

[24] Laidlaw, D. H., Trumbore, W. B., and Hughes, J. F. Constructive solid geometry for polyhedral objects. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1986), SIGGRAPH '86, ACM, pp. 161–170.

[25] Langton, C. G. Studying artificial life with cellular automata. *Physica D 22D*, 1-3 (1986), 120–49.

[26] Laycock, R. G., and Day, A. M. Automatically generating large urban environments based on the footprint data of buildings. In *Proceedings of the eighth ACM symposium on Solid modeling and applications* (New York, NY, USA, 2003), SM '03, ACM, pp. 346–351.

[27] Levoy, M. Efficient ray tracing of volume data. *ACM Trans. Graph. 9* (July 1990), 245–261.

[28] Li, M., and Vitnyi, P. M. *An Introduction to Kolmogorov Complexity and Its Applications*, 3 ed. Springer Publishing Company, Incorporated, 2008.

[29] Lipp, M., Wonka, P., and Wimmer, M. Interactive visual editing of grammars for procedural architecture. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers* (New York, NY, USA, 2008), ACM, pp. 1–10.

[30] Lorensen, W. E., and Cline, H. E. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph. 21* (August 1987), 163–169.

[31] Mandagere, N., Zhou, P., Smith, M. A., and Uttamchandani, S. Demystifying data deduplication. In *Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion* (New York, NY, USA, 2008), Companion '08, ACM, pp. 12–17.

[32] Martin, J. Procedural house generation: A method for dynamically generating floor plans. In *Proceedings of the Symposium on Interactive 3D Graphics and Games* (2006), Citeseer.

[33] Merrell, P. Example-based model synthesis. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2007), I3D '07, ACM, pp. 105–112.

[34] Müller, P., Wonka, P., Haegler, S., Ulmer, A., and Van Gool, L. Procedural modeling of buildings. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers* (New York, NY, USA, 2006), ACM, pp. 614–623.

[35] Müller, P., Zeng, G., Wonka, P., and Van Gool, L. Image-based procedural modeling of facades. In *SIGGRAPH '07: ACM SIGGRAPH 2007 papers* (New York, NY, USA, 2007), ACM, p. 85.

[36] Musgrave, F. K. *Methods for realistic landscape imaging.* PhD thesis, New Haven, CT, USA, 1993.

[37] Musgrave, F. K., Kolb, C. E., and Mace, R. S. The synthesis and rendering of eroded fractal terrains. In *Proceedings of the 16th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1989), SIGGRAPH '89, ACM, pp. 41–50.

[38] Musgrave, F. K., Peachey, D., Perlin, K., and Worley, S. *Texturing and modeling: a procedural approach.* Academic Press Professional, Inc., San Diego, CA, USA, 1994.

[39] Měch, R., and Prusinkiewicz, P. Visual models of plants interacting with their environment. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1996), SIGGRAPH '96, ACM, pp. 397–410.

[40] Olsen, J. Realtime procedural terrain generation. Tech. rep., Department of Mathematics And Computer Science (IMADA), University of Southern Denmark, 2004.

[41] Ong, T. J., Saunders, R., Keyser, J., and Leggett, J. J. Terrain generation using genetic algorithms. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation* (New York, NY, USA, 2005), ACM, pp. 1463–1470.

[42] Parish, Y. I. H., and Müller, P. Procedural modeling of cities. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2001), SIGGRAPH '01, ACM, pp. 301–308.

[43] Perlin, K. Improving noise. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2002), SIGGRAPH '02, ACM, pp. 681–682.

[44] Prusinkiewicz, P., and Lindenmayer, A. *The algorithmic beauty of plants.* Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[45] Rappoport, A., and Spitz, S. Interactive boolean operations for conceptual design of 3-d solids. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1997), SIGGRAPH '97, ACM Press/Addison-Wesley Publishing Co., pp. 269–278.

[46] Rusinkiewicz, S., and Levoy, M. Qsplat: a multiresolution point rendering system for large meshes. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2000), SIGGRAPH '00, ACM Press/Addison-Wesley Publishing Co., pp. 343–352.

[47] Samet, H. *The design and analysis of spatial data structures.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[48] Smelik, R., de Kraker, K., Groenewegen, S., Tutenel, T., and Bidarra, R. A survey of procedural methods for terrain modelling. *3D Advanced Media In Gaming And Simulation (3AMIGAS)* (2009), 25.

[49] Stiny, G., and Gips, J. Shape grammars and the generative specification of painting and sculpture. *Information processing 71* (1972), 1460–1465.

[50] Sun, J., Yu, X., Baciu, G., and Green, M. Template-based generation of road networks for virtual city modeling. In *Proceedings of the ACM symposium on Virtual reality software and technology* (New York, NY, USA, 2002), VRST '02, ACM, pp. 33–40.

[51] Turk, G. Generating textures on arbitrary surfaces using reaction-diffusion. In *Proceedings of the 18th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1991), SIGGRAPH '91, ACM, pp. 289–298.

[52] Wakefield, P. Close encounters of the 3d kind. `http://www.webcitation.org/5q2nTnS9g`, December 2009 (Accessed 2012-01-12).

[53] Wei, L.-Y., and Levoy, M. Fast texture synthesis using tree-structured vector quantization. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2000), ACM Press/Addison-Wesley Publishing Co., pp. 479–488.

[54] Whitaker, R. T. Reducing aliasing artifacts in iso-surfaces of binary volumes. In *Proceedings of the 2000 IEEE symposium on Volume visualization* (New York, NY, USA, 2000), VVS '00, ACM, pp. 23–32.

[55] Wolfram, S. Statistical mechanics of cellular automata. *Reviews of Modern Physics 55*, 3 (July 1983), 601–644.

[56] Wonka, P., Wimmer, M., Sillion, F., and Ribarsky, W. Instant architecture. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers* (New York, NY, USA, 2003), ACM, pp. 669–677.

[57] Zhou, H., Sun, J., Turk, G., and Rehg, J. M. Terrain synthesis from digital elevation models. *IEEE Transactions on Visualization and Computer Graphics 13*, 4 (July/August 2007), 834–848.

Appendix A

Additional Performance Testing Graphs

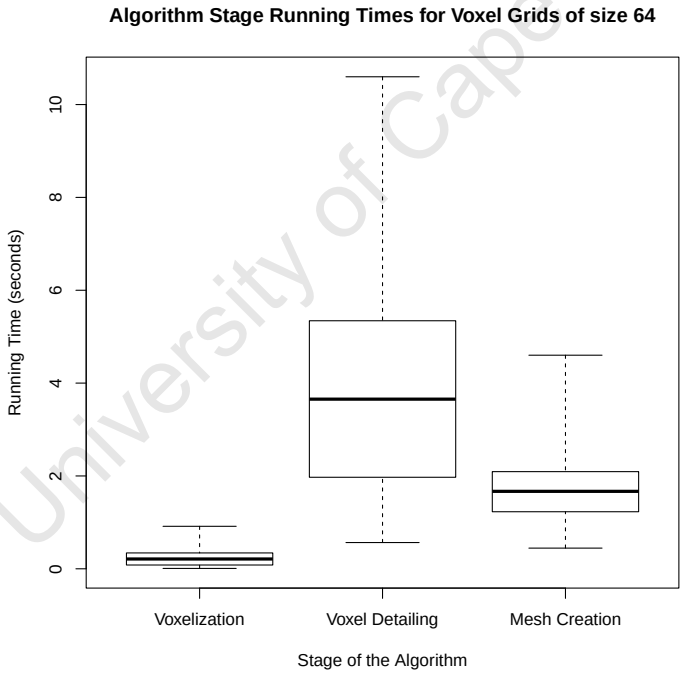


Figure A.1: A box and whisker plot showing the range of running times for the voxelization, voxel detailing, and mesh generation stages of the algorithm, on voxel grids of size 64.

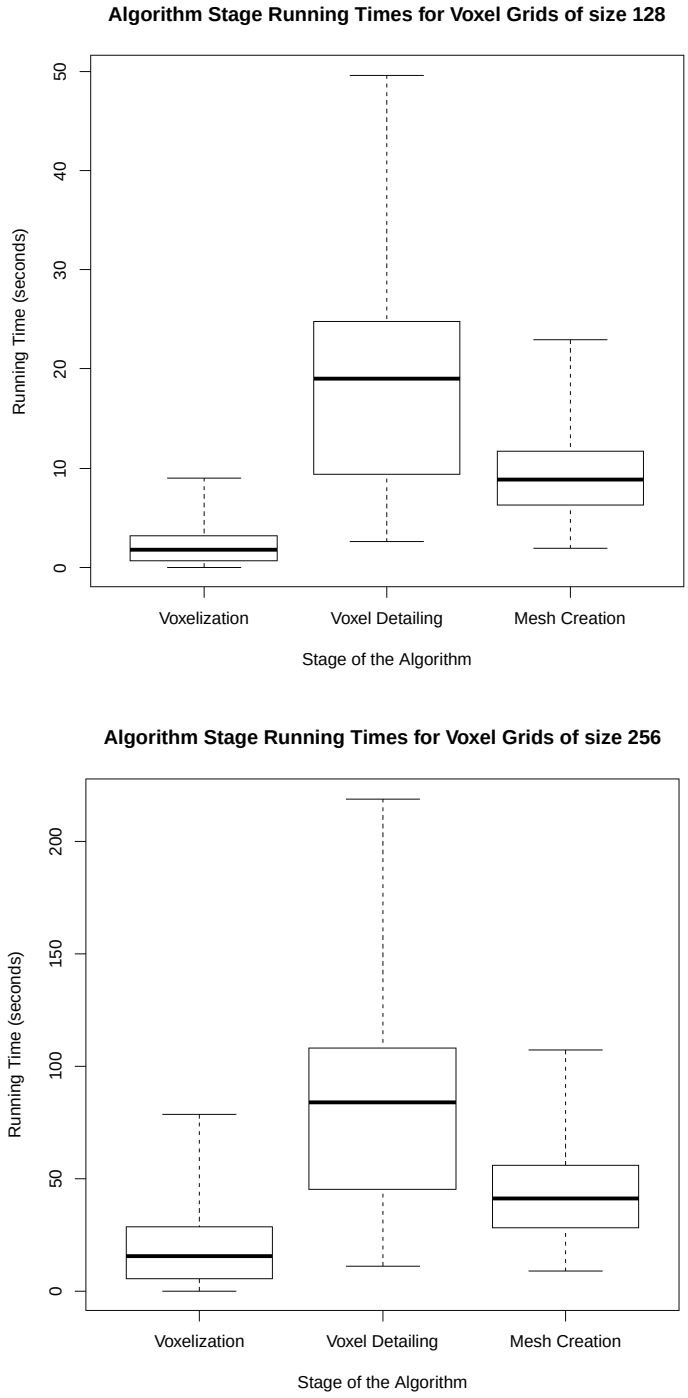


Figure A.2: Box and whisker plots showing the range of running times for the voxelization, voxel detailing, and mesh generation stages of the algorithm, on voxel grids of size 128 (top) and 256 (bottom).

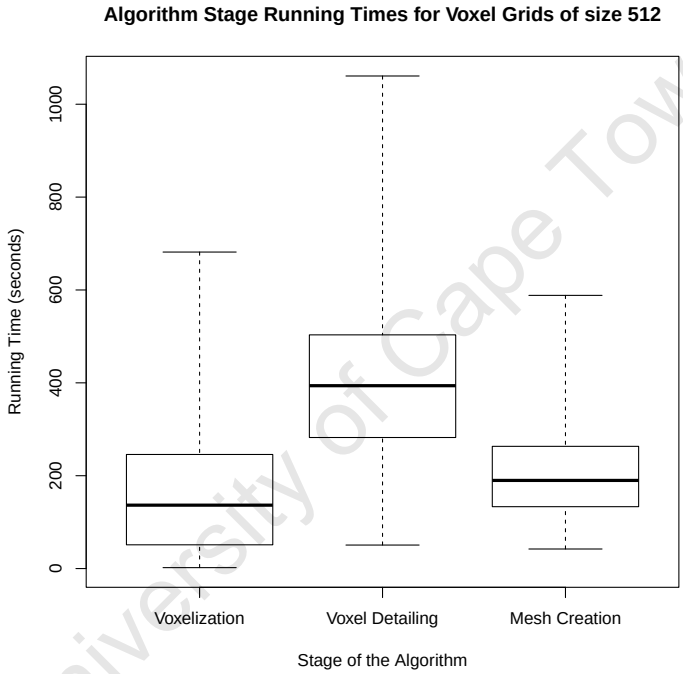
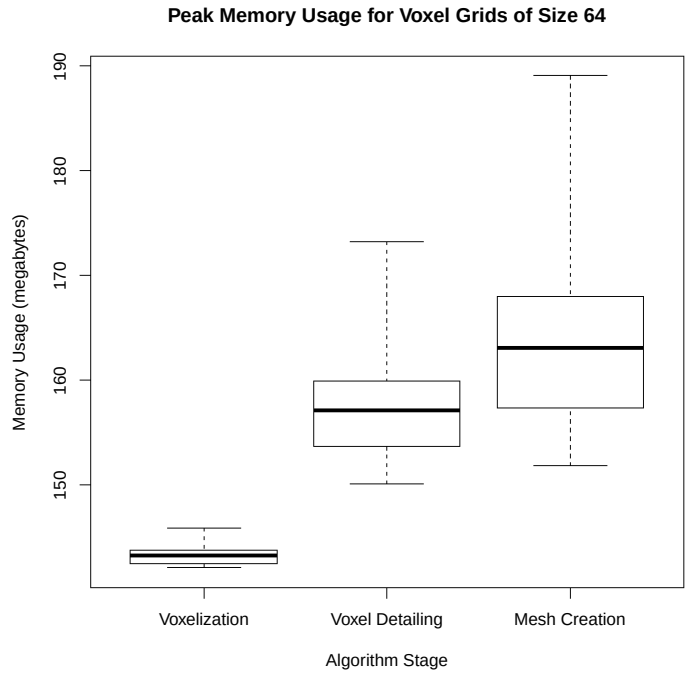


Figure A.3: Box and whisker plots showing the range of running times for the voxelization, voxel detailing, and mesh generation stages of the algorithm, on voxel grids of size 512.



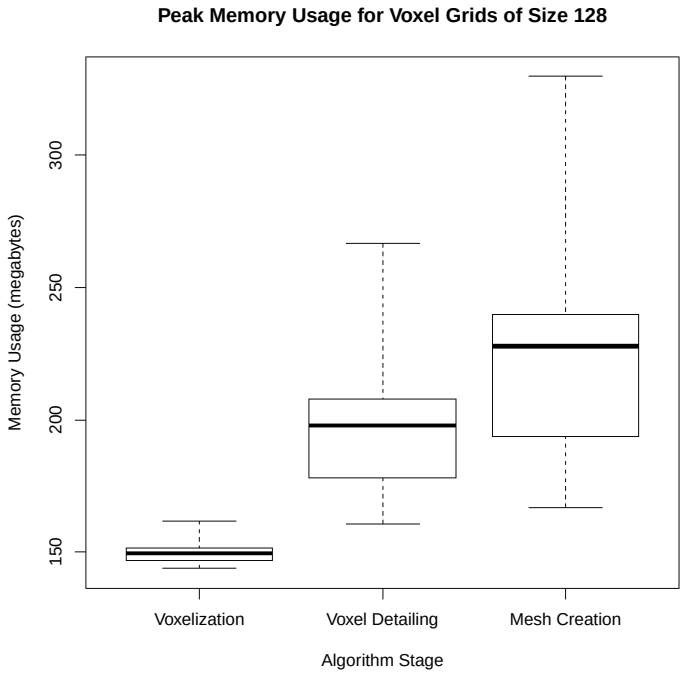
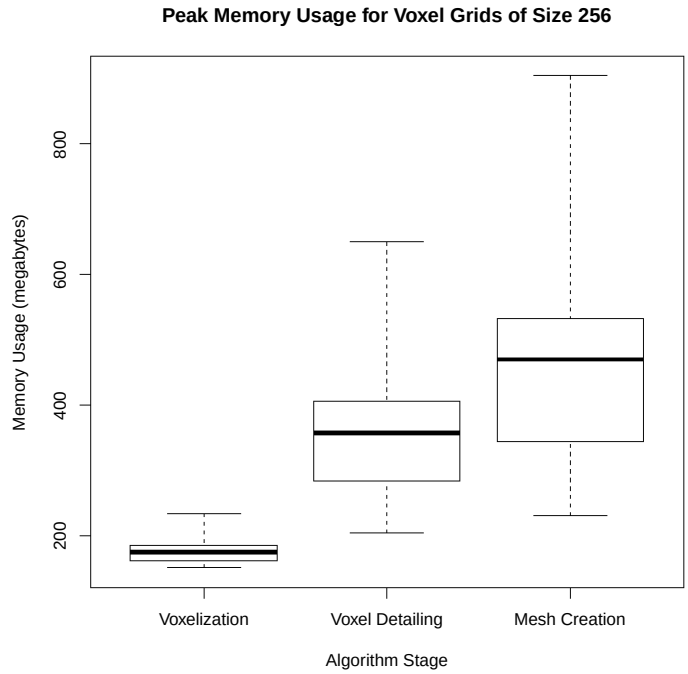


Figure A.4: Box and whisker plots showing the range of peak memory usage for the voxelization, voxel detailing, and mesh generation stages of the algorithm, on voxel grids of size 64 and 128.



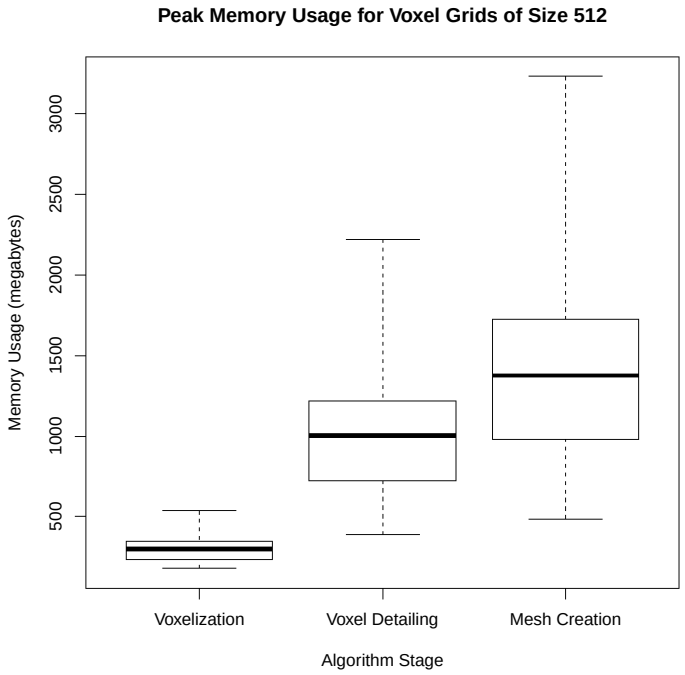


Figure A.5: Box and whisker plots showing the range of peak memory usage for the voxelization, voxel detailing, and mesh generation stages of the algorithm, on voxel grids of size 256 and 512.

Appendix B

Stochastic Shape Grammars

B.1 Castle

```
# castle shape grammar

#################################################################
# Global parameter assignment, including stochastic stuff.
#################################################################

castle_keep_extents = vec3(random.randint(300,500),
    random.randint(50,500), random.randint(300,500))

castle_keep_centralized = True
if random.randint(1,2) <= 1:
    castle_keep_centralized = False

castle_keep_offset = vec3(random.uniform(0,0.25),
    random.uniform(0,0.25), random.uniform(0,0.25))
castle_keep_wall_bordering = vec3(random.randint(-1,1), 0,
    random.randint(-1,1))

castle_keep_central_tower = True
if random.randint(1,2) > 1:
    castle_keep_central_tower = True
else:
    castle_keep_central_tower = False

castle_keep_corner_towers = [True, True, True, True]
for i in range(4):
    if random.randint(1,2) <= 1:
        castle_keep_corner_towers[i] = True
    else:
        castle_keep_corner_towers[i] = False

castle_top_wall_extents = vec3(random.randint(750,1500), 100, 50)
castle_side_wall_extents = vec3(random.randint(750,1500), 100,
    50)
castle_wall_height = random.randint(100,250)

castle_tower_width = random.randint(100,200)
castle_tower_extents = vec3(castle_tower_width,
    random.randint(300,500), castle_tower_width)
castle_tower_position_relative = vec3(-1,0,0)
if random.randint(1,2) > 1:
```

```
37      castle_tower_position_relative =
            vec3(random.uniform(-1,0),0,random.uniform(-1,1))
38
39  castle_tower_mirror = True
40  if random.randint(1,3) <= 1:
41      castle_tower_mirror = False
42
43  castle_gate_extents = vec3(50, 50, castle_tower_extents[2])
44
45  castle_gate_position_relative = vec3(0,0,0)
46  if random.randint(1,2) > 1:
47      castle_gate_position_relative =
            vec3(random.uniform(-1,-0.33333),0,0)
48
49  castle_gate_mirror = False
50  if random.randint(1,3) <= 1:
51      castle_gate_mirror = True
52
53  #relative to wall
54  castle_battlements_scale_factor = vec3(1, 0.3, 0.3)
55  castle_battlements_present = True
56  if random.randint(1,3) <= 1:
57      castle_battlements_present = False
58
59  #relative to tower
60  castle_tower_battlements_scale_factor = vec3(1.33333, 0.2,
        1.33333)
61
62  ###################################
63  # The actual shape grammar rules.
64  ###################################
65
66  def castle(parent):
67      parent.active = False
68
69      keep = Node("castle_keep")
70      keep.extents = castle_keep_extents
71      if castle_keep_centralized:
72          keep.position = castle_keep_offset
73          keep.position = vec3(castle_top_wall_extents.x *
                castle_keep_offset[0] + castle_keep_extents[0] *
                -castle_keep_offset[0], 0, castle_side_wall_extents.x
                * castle_keep_offset[2] + castle_keep_extents[2] *
                -castle_keep_offset[2])
74      else:
75          keep.position = vec3(castle_top_wall_extents.x *
                castle_keep_wall_bordering[0] +
                castle_keep_extents[0] *
                -castle_keep_wall_bordering[0], 0,
                castle_side_wall_extents.x *
                castle_keep_wall_bordering[2] +
                castle_keep_extents[2] *
                -castle_keep_wall_bordering[2])
76      keep.position[1] = keep.extents[1]
77      keep.tags.append("castle_keep")
78      keep.priority = 0
79
80      top_wall = Node("castle_wall")
81      top_wall.extents = castle_top_wall_extents
82      top_wall.extents.y = castle_wall_height
83      top_wall.position = vec3(0, castle_wall_height,
            castle_side_wall_extents.x)
```

```
84      top_wall.tags.append("castle_wall")
85      top_wall.priority = 0
86      top_wall.setSymmetry("reflective", vec3(0,0,0), vec3(0, 0,
            1))
87
88      side_wall = Node("castle_wall")
89      side_wall.extents = castle_side_wall_extents
90      side_wall.extents.y = castle_wall_height
91      side_wall.position = vec3(castle_top_wall_extents.x,
            castle_wall_height, 0)
92      side_wall.orientation = mat3.rotation(math.pi / 2,
            vec3(0,1,0))
93      side_wall.tags.append("castle_wall")
94      side_wall.priority = 0
95      side_wall.setSymmetry("reflective", vec3(0,0,0), vec3(1, 0,
            0))
96
97      return [top_wall, side_wall, keep]
98
99  def castle_wall(parent):
100     parent.name = "rectangle"
101
102     tower = Node("castle_tower")
103     tower.extents = castle_tower_extents
104     tower.position =
            parent.corner_relative(castle_tower_position_relative[0],
            castle_tower_position_relative[1],
            castle_tower_position_relative[2])
105     tower.position[1] = tower.position[1] + tower.extents[1] -
            parent.extents[1]
106     tower.tags.append("castle_tower")
107     if castle_tower_mirror:
108         tower.setSymmetry("reflective", parent.position,
                vec3(1,0,0))
109
110     battlements = Node("castle_battlements")
111     battlements.extents = vec3(parent.extents[0] *
            castle_battlements_scale_factor[0], parent.extents[1] *
            castle_battlements_scale_factor[1], parent.extents[2] *
            castle_battlements_scale_factor[2])
112     battlements.position = parent.corner_relative(0,1,1)
113     battlements.tags.append("castle_battlements")
114     battlements.setSymmetry("reflective", parent.position,
            vec3(0,0,1))
115
116     gate = Node("castle_gate")
117     gate.extents = castle_gate_extents
118     gate.extents[1] = gate.extents[1] * 0.66666
119     gate.position =
            parent.corner_relative(castle_gate_position_relative[0],
            castle_gate_position_relative[1],
            castle_gate_position_relative[2])
120     gate.position[1] = gate.position[1] + gate.extents[1] -
            parent.extents[1]
121     gate.tags.append("castle_gate")
122     if castle_gate_mirror:
123         gate.setSymmetry("reflective", parent.position,
                vec3(1,0,0))
124     gate.additive = False
125
126     ret = [tower, gate]
127     if castle_battlements_present:
```

```
128             ret.append(battlements)
129         return ret
130
131 def castle_tower(parent):
132         parent.name = "cylinder"
133
134         tower_battlements = Node("castle_tower_battlements")
135         tower_battlements.position = parent.corner_relative(0,1,0)
136         tower_battlements.extents = vec3(parent.extents[0] *
                castle_tower_battlements_scale_factor[0],
                parent.extents[1] *
                castle_tower_battlements_scale_factor[1],
                parent.extents[2] *
                castle_tower_battlements_scale_factor[2])
137         tower_battlements.tags.append("tower_battlements")
138
139         return [tower_battlements]
140
141 def castle_battlements(parent):
142         parent.name = "rectangle"
143
144         return []
145
146 def castle_tower_battlements(parent):
147         parent.name = "cylinder"
148
149         subtractive_area = Node("cylinder")
150         subtractive_area.additive = False
151         subtractive_area.extents = vec3(parent.extents[0] * 0.8,
                parent.extents[1] * 2, parent.extents[2] * 0.8)
152         subtractive_area.position = parent.corner_relative(0,2,0)
153         subtractive_area.priority = parent.priority - 1
154         subtractive_area.tags.append("tower_battlements_subtractive_area")
155
156         return [subtractive_area]
157
158 def castle_gate(parent):
159         parent.name = "rectangle"
160
161         arch = Node("cylinder")
162         arch.orientation = mat3.rotation(math.pi / 2, vec3(1,0,0))
163         arch.extents = vec3(parent.extents[0], parent.extents[2],
                parent.extents[0])
164         arch.position = parent.corner_relative(0,1,0)
165         arch.additive = False
166         arch.tags.append("gate_subtractive_area")
167
168         return [arch]
169
170 def castle_keep(parent):
171         parent.name = "rectangle"
172
173         ret = []
174
175         roof = Node("rectangle")
176         roof.orientation = mat3.rotation(math.pi / 4, vec3(0,0,1))
177         roof.extents = vec3(parent.extents[0] / 2.0,
                parent.extents[0] / 2.0, parent.extents[2])
178         roof.position = parent.corner_relative(0,1,0)
179         roof.tags.append("keep_roof")
180
181         ret.append(roof)
```

```
182
183     central_tower = Node("castle_tower")
184     central_tower.extents = castle_tower_extents
185     central_tower.position = parent.corner_relative(0,1,0) +
            vec3(0,castle_tower_extents[1],0)
186     central_tower.tags.append("keep_tower")
187     if castle_keep_central_tower:
188         ret.append(central_tower)
189
190     count = 0
191     for i in castle_keep_corner_towers:
192         count = count + 1
193         if i:
194             corner_tower = Node("castle_tower")
195             corner_tower.extents = castle_tower_extents
196             corner_tower.tags.append("keep_tower")
197
198             t_pos = vec3(0,0,0)
199             if count == 1:
200                 t_pos = vec3(-1,0,-1)
201             elif count == 2:
202                 t_pos = vec3(-1,0,1)
203             elif count == 3:
204                 t_pos = vec3(1,0,-1)
205             elif count == 4:
206                 t_pos = vec3(1,0,1)
207
208             corner_tower.position =
                    parent.corner_relative(t_pos[0], 0, t_pos[2])
209             corner_tower.position[1] = -parent.extents[1] +
                    corner_tower.extents[1]
210             ret.append(corner_tower)
211
212     ground_subtractive_thing = Node("rectangle")
213     ground_subtractive_thing.extents = parent.extents
214     ground_subtractive_thing.position = vec3(0,
            -ground_subtractive_thing.extents[1] * 2, 0)
215     ground_subtractive_thing.additive = False
216     ground_subtractive_thing.priority = parent.priority - 5
217     ret.append(ground_subtractive_thing)
218
219     return ret
```

B.2 Space Station

```
1  # Stochastic Space Station Grammar
2
3  ##############################################################
4  # Global parameter assignment, including stochastic stuff.
5  ##############################################################
6
7  space_station_num_spires = random.randint(1,7)
8  space_station_num_spokes = random.randint(1,7)
9
10 space_station_radius = random.randint(50,80)
11 space_station_extents = vec3(space_station_radius,
       random.randint(10,25), space_station_radius)
12
```

```
13 space_station_subtractive_disk_radius = space_station_radius *
       random.uniform(0.5, 0.85)
14 space_station_subtractive_disk_extents =
       vec3(space_station_subtractive_disk_radius,
       space_station_extents[1] * 1.2,
       space_station_subtractive_disk_radius)
15
16 ss_spire_extents = vec3(random.randint(10,15),
       random.randint(20,50), random.randint(10,15))
17 ss_spire_shape = "ellipsoid"
18 if random.randint(1,2) <= 1:
19     ss_spire_shape = "cylinder"
20
21 ss_spoke_extents = vec3(random.randint(3,8),
       space_station_radius / 2.0, random.randint(3,8))
22
23 ss_centre_radius = random.randint(7,15)
24 ss_centre_extents = vec3(ss_centre_radius,
       random.randint(10,30), ss_centre_radius)
25 ss_centre_shape = "ellipsoid"
26 if random.randint(1,2) <= 1:
27     ss_centre_shape = "cylinder"
28
29 ss_spire_tower_scale_factor = vec3(random.uniform(0.5, 0.9),
       random.uniform(0.4, 0.6), random.uniform(0.5, 0.9))
30
31 ss_tower_top_scale_factor = random.uniform(0.5,0.8)
32 ss_tower_top_present = True
33 if random.randint(1,2) <= 1:
34     ss_tower_top_present = False
35
36 ####################################
37 # The actual shape grammar rules.
38 ####################################
39
40 def space_station(parent):
41     parent.active = False
42
43     #make ring using subtraction
44     ring = Node("ellipsoid")
45     ring.position = vec3(0,0,0)
46     ring.extents = space_station_extents
47     ring.tags = ["ring"]
48     ring.priority = 0
49
50     subtractive_ring = Node("cylinder")
51     subtractive_ring.position = vec3(0,0,0)
52     subtractive_ring.extents =
           space_station_subtractive_disk_extents
53     subtractive_ring.additive = False
54     subtractive_ring.priority = ring.priority - 1
55
56     #make spire
57     spire = Node("spire")
58     spire.position = vec3(space_station_radius,0,0)
59     spire.extents = ss_spire_extents
60     spire.tags.append("spire")
61     spire.setSymmetry("rotational", vec3(0,0,0), vec3(0,1,0),
           space_station_num_spires)
62     spire.priority = subtractive_ring.priority - 1
63
64     #add spokes
```

```
65      spoke = Node("space_station_spoke")
66      spoke.position = vec3(-space_station_radius / 2.0, 0, 0)
67      spoke.extents = ss_spoke_extents
68      spoke.orientation = mat3.rotation(math.pi / 2, vec3(0,0,1))
69      spoke.tags.append("spoke")
70      spoke.setSymmetry("rotational", vec3(0,0,0), vec3(0,1,0),
            space_station_num_spokes)
71      spoke.priority = -5
72
73      #add central blob
74      cent = Node(ss_centre_shape)
75      cent.position = vec3(0,0,0)
76      cent.extents = ss_centre_extents
77      cent.tags.append("centre")
78      cent.priority = subtractive_ring.priority - 1
79
80      return [ring, subtractive_ring, spire, spoke, cent]
81
82  def space_station_spoke(parent):
83      parent.name = "cylinder"
84
85      subspoke = Node("cylinder")
86      subspoke.position = vec3(0,0,0)
87      subspoke.additive = False
88      subspoke.extents.x = parent.extents.x * 0.5
89      subspoke.extents.y = parent.extents.y * 1.2
90      subspoke.extents.z = parent.extents.z * 0.5
91      subspoke.priority = -10
92      subspoke.tags.append("spoke_passage")
93
94      return [subspoke]
95
96  def spire(parent):
97      parent.name = ss_spire_shape
98
99      tow = Node("tower")
100     tow.position = parent.corner_relative(0,1,0)
101     tow.extents = vec3(parent.extents.x *
            ss_spire_tower_scale_factor.x, parent.extents.y *
            ss_spire_tower_scale_factor.y, parent.extents.z *
            ss_spire_tower_scale_factor.z)
102     tow.setSymmetry("reflective", parent.position, vec3(0,1,0))
103
104     return [tow]
105
106 def tower(parent):
107     parent.name = "cylinder"
108
109     ret = []
110
111     if ss_tower_top_present:
112         tower_top = Node("cylinder")
113         tower_top.tags.append("tower")
114         tower_top.position = parent.corner_relative(0,1,0)
115         tower_top.extents = parent.extents *
                ss_tower_top_scale_factor
116
117         ret.append(tower_top)
118
119     return ret
```

B.3 Tank

```
# a simple tank, for use with the camouflage pattern.

#############################################################
# Global parameter assignment, including stochastic stuff.
#############################################################

tank_body_length = random.randint(100, 180)
tank_body_width = random.randint(60, 100)
tank_body_height = random.randint(15, 35)
tank_position = vec3(0,0,0)

tank_body_bump_size_factor = vec3(random.uniform(0.5, 0.9),
    random.uniform(0.1, 0.3), random.uniform(0.5,0.9))

tank_side_indentation_offset = random.uniform(-1,1)
tank_side_indentation_length = random.uniform(0.5, 1.5)
tank_side_indentation_rotation = math.pi / 4 *
    random.uniform(0.5, 1.5)

#relative to body
tank_turret_offset = vec3(random.uniform(-0.5,0.5), 1.5,
    random.uniform(-0.75, 0.75))
tank_turret_size_factor = vec3(random.uniform(0.5,0.75),
    random.uniform(0.5,0.8), random.uniform(0.5,0.75))
tank_turret_double = False
if random.randint(1,2) <= 1:
    tank_turret_size_factor = vec3(random.uniform(0.3,0.5),
        random.uniform(0.3,0.5), random.uniform(0.3,0.8))
    tank_turret_offset = vec3(random.uniform(0.5,0.8), 1.5,
        random.uniform(-0.75, 0.8))
    tank_turret_double = True

#relative to the body
tank_tread_offset = vec3(random.uniform(0.4,0.8), -1,
    random.uniform(-0.5, 0.5))
tank_tread_size_factor = vec3(0.25, 1, random.uniform(0.75, 1.1))

#relative to the body length
tank_gun_barrel_width = random.uniform(0.05, 0.1)
tank_gun_size_factor = vec3(tank_gun_barrel_width,
    random.uniform(0.4,0.6), tank_gun_barrel_width)
tank_gun_offset = vec3(random.uniform(-0.5,0),
    random.uniform(0,0.5), 0)

tank_double_guns = False
if random.randint(1,2) <= 1:
    tank_gun_size_factor = vec3(tank_gun_barrel_width * 0.6,
        random.uniform(0.3,0.5), tank_gun_barrel_width * 0.6)
    tank_gun_offset = vec3(random.uniform(-0.75,0),
        random.uniform(0,0.5), 0)
    tank_double_guns = True

#relative to the gun
tank_gun_tip_size_factor = vec3(random.uniform(0.5,1.2),
    random.uniform(0.2,0.5), random.uniform(0.5,1.2))
tank_gun_tip_offset = vec3(0, random.uniform(0.5,1), 0)
tank_gun_shape = "cylinder"
tank_gun_tip_type = random.randint(1,3)
if tank_gun_tip_type == 1:
    tank_gun_shape = "rectangle"
```

```
49  elif tank_gun_tip_type == 2:
50      tank_gun_shape = "ellipsoid"
51
52  tank_back_area_size = vec3(random.uniform(0.3,1.1),
        random.uniform(0.3,0.75), random.uniform(0.1,0.3))
53  tank_back_area_offset = vec3(random.uniform(-0.2,0.2),
        random.uniform(0.5, 0.75), random.uniform(-1,-0.8))
54
55  ####################################
56  # The actual shape grammar rules.
57  ####################################
58
59  def tank(parent):
60      parent.active = False
61
62      ret = Node("tank_main_body")
63      ret.extents = vec3(tank_body_width, tank_body_height,
            tank_body_length)
64      ret.position = tank_position
65      ret.tags.append("main_body")
66      ret.priority = 0
67
68      return [ret]
69
70  def tank_main_body(parent):
71      parent.name = "rectangle"
72
73      top_raised_area = Node("rectangle")
74      top_raised_area.extents = vec3(parent.extents.x *
            tank_body_bump_size_factor.x, parent.extents.y *
            tank_body_bump_size_factor.y, parent.extents.z *
            tank_body_bump_size_factor.z)
75      top_raised_area.position = parent.corner_relative(0,1,0)
76
77      back_raised_area = Node("rectangle")
78      back_raised_area.extents = vec3(parent.extents.x *
            tank_back_area_size.x, parent.extents.y *
            tank_back_area_size.y, parent.extents.z *
            tank_back_area_size.z)
79      back_raised_area.position =
            parent.corner_relative(tank_back_area_offset.x,
            tank_back_area_offset.y, tank_back_area_offset.z)
80      back_raised_area.priority = -10
81
82      front_indentation = Node("tank_indentation")
83      front_indentation.extents = vec3(tank_body_width,
            tank_body_height, tank_body_height)
84      front_indentation.position = parent.corner_relative(0,-1,1)
85      front_indentation.orientation = mat3.rotation(math.pi / 4,
            vec3(1,0,0))
86      front_indentation.setSymmetry("reflective", tank_position,
            vec3(0,0,1))
87      front_indentation.additive = False
88      front_indentation.priority = -5
89
90      side_indentation = Node("tank_indentation")
91      side_indentation.extents = vec3(tank_body_height,
            tank_body_length * tank_side_indentation_length,
            tank_body_height)
92      side_indentation.position =
            parent.corner_relative(1.3,1.3,tank_side_indentation_offset)
93      side_indentation.orientation = mat3.rotation(math.pi / 2,
```

```
        vec3(1,0,0)) * mat3.rotation(math.pi / 4, vec3(0,0,1))
94      side_indentation.setSymmetry("reflective", tank_position,
            vec3(1,0,0))
95      side_indentation.additive = False
96      side_indentation.priority = -15
97
98      turret = Node("tank_turret")
99      turret.extents = vec3(tank_body_width *
            tank_turret_size_factor[0], tank_body_height *
            tank_turret_size_factor[1], tank_body_width *
            tank_turret_size_factor[2])
100     turret.position =
            parent.corner_relative(tank_turret_offset[0],
            tank_turret_offset[1], tank_turret_offset[2])
101
102     turret.tags.append("turret")
103     if tank_turret_double:
104         turret.setSymmetry("reflective", tank_position,
                vec3(1,0,0))
105     turret.priority = -20
106
107     tread = Node("tank_tread")
108     tread.extents = vec3(tank_body_height *
            tank_tread_size_factor[1], tank_body_width *
            tank_tread_size_factor[0], tank_body_length *
            tank_tread_size_factor[2])
109     tread.position =
            parent.corner_relative(tank_tread_offset[0],
            tank_tread_offset[1], tank_tread_offset[2])
110     tread.setSymmetry("reflective", parent.position, vec3(1,0,0))
111     tread.orientation = mat3.rotation(math.pi / 2, vec3(0,0,1))
112     tread.priority = -20
113     tread.tags.append("tread")
114
115     return [front_indentation, side_indentation, turret, tread,
            top_raised_area, back_raised_area]
116
117 def tank_indentation(parent):
118     parent.name = "rectangle"
119
120     return []
121
122 def tank_turret(parent):
123     parent.name = "ellipsoid"
124
125     gun = Node("tank_gun")
126     gun.extents = vec3(tank_body_length *
            tank_gun_size_factor[0], tank_body_length *
            tank_gun_size_factor[1], tank_body_length *
            tank_gun_size_factor[2])
127     gun.position = vec3(0,0,gun.extents[1]) +
            parent.corner_relative(tank_gun_offset[0],
            tank_gun_offset[1], tank_gun_offset[2])
128
129     gun.orientation = mat3.rotation(-math.pi / 2, vec3(1,0,0))
130     gun.tags.append("gun")
131     if tank_double_guns:
132         gun.setSymmetry("reflective", parent.position,
                vec3(1,0,0))
133     gun.priority = -20
134
135     return [gun]
```

```
136
137 def tank_tread(parent):
138     parent.name = "cylinder"
139
140     return []
141
142 def tank_gun(parent):
143     parent.name = "cylinder"
144
145     gun_tip = Node("tank_gun_tip")
146     gun_tip.extents = vec3(parent.extents[0] *
            tank_gun_tip_size_factor[0], parent.extents[1] *
            tank_gun_tip_size_factor[1], parent.extents[2] *
            tank_gun_tip_size_factor[2])
147     gun_tip.position =
            parent.corner_relative(tank_gun_tip_offset[0],
            tank_gun_tip_offset[1], tank_gun_tip_offset[2])
148     gun_tip.tags.append("gun_tip")
149     gun_tip.priority = -20
150
151     return [gun_tip]
152
153 def tank_gun_tip(parent):
154     parent.name = tank_gun_shape
155
156     return []
```

Attribution of Re-used Images

Please note that my use or adaption of the images created by the individuals below does not, in any way, imply that they endorse me or my use of their work.

Figure 1.2 was originally created by Wikipedia user 'Lobsterbake', based on an image created by Wikipedia user 'Rchoetzlein', and was released under the Creative Commons Attribution-Share Alike 3.0 Unported license. It was obtained from the URL `https://en.wikipedia.org/wiki/File:Mesh_overview.svg` on 2012-01-07.

Figure 1.3 was created by the Wikipedia user 'Vossman', and is licensed under the Creative Commons Attribution-Share Alike 2.5 Generic license. It was obtained from the following URL: `https://en.wikipedia.org/wiki/File:Voxels.svg` on 2011-11-21.

Figure 2.4 is a screenshot from a demo created by the Demoscene group 'Conspiracy' (`http://www.conspiracy.hu/`). It was uploaded to Wikipedia by user 'Gargaj' (who is a member of Conspiracy) under the Creative Commons Attribution-Share Alike 2.5 Generic license. It was obtained from the following URL: `https://en.wikipedia.org/wiki/File:Beyond_-_Conspiracy_-_2004_-_64k_intro.jpg` on 2012-01-12.

Figure 2.5 is a screenshot from a demo by the Demoscene group 'Black Maiden' (`http://www.blackmaiden.de/`). It was uploaded to Wikipedia by Tim 'Avatar' Bartel (who is a member of Black Maiden), under the Creative Commons Attribution-Share Alike 2.0 Generic license. It was obtained from the following URL: `https://en.wikipedia.org/wiki/File:Demo_PC_BlackMaiden_Interceptor.jpg` on 2012-01-12.

Figure 2.7 was derived from the Wikipedia article on L-systems (`https://en.wikipedia.org/wiki/L-system`, accessed on 2012-01-16). The list of authors who have worked on the example can be found in the page's history. I modified the figure by changing some of the names of the labels, removing some of the labels on the visual diagram, and slightly reformatting the text to make it fit better as a figure in this thesis. I release all text in the modified figure (not including the caption) under the Creative Commons Attribution-Share-Alike License 3.0 (full text available at `http://creativecommons.org/licenses/by-sa/3.0/`).

Figure 2.9 was derived from the diagram created by Wikipedia user 'WhiteTimberwolf', which was released under the Creative Commons Attribution-Share Alike 3.0 Unported license (obtained from the following URL: `https://en.wikipedia.org/wiki/File:Octree2.svg` on 2011-11-21). I release this derivative work under that same license.

Figure 2.10 was originally created by Wikipedia user 'Mysid', based on an image created by Wikipedia user 'Cyp', and was released under the Creative Commons Attribution-Share Alike 3.0 Unported license. It was obtained from the URL `https://en.wikipedia.org/wiki/File:Coloured_Voronoi_2D.svg` on 2012-01-12.

Figure 2.12 was based on an image originally created by Sam Jervis, and uploaded to Wikipedia under the NetHack GPL license. I release this modified image under the same license.

Figure 2.15 was created by the Wikipedia user 'WhiteTimberwolf', and is licensed under the Creative Commons Attribution-Share Alike 3.0 Unported license. It was obtained from the following URL: `https://en.wikipedia.org/wiki/File:Octree2.svg` on 2011-11-21.

Figure 2.16 was derived from the diagram created by Wikipedia user 'WhiteTimberwolf', which was released under the Creative Commons Attribution-Share Alike 3.0 Unported license (obtained from the following URL: `https://en.wikipedia.org/wiki/File:Octree2.svg` on 2011-11-21). I release this derivative work under that same license.

Figure 2.17 was made by combining an image created by Wikipedia user 'KiwiSunset' (released under the Creative Commons Attribution ShareAlike 3.0 License, obtained at the URL: `https://en.wikipedia.org/wiki/File:Kdtree_2d.svg` on 2011-11-21) with an image created by Wikipedia user 'MYguel' (released into the public domain, found at the URL: `https://en.wikipedia.org/wiki/File:Tree_0001.svg` on 2011-11-21). I release this derivative work under the Creative Commons Attribution ShareAlike 3.0 License.

Figure 2.18 was originally created by Stefan de Konink (Wikipedia username: 'Skinkie'), and was released into the public domain. It was obtained from the following URL: `https://en.wikipedia.org/wiki/File:R-tree.svg` on 2011-11-21.