Procedurally generating surface detail for 3D models using voxel-based cellular automata

by Ryan Mazzolini
Supervisors: James Gain and Patrick Marais

A thesis submitted for the fulfilment of the requirements of the degree of

Master of Science

The Department Of Computer Science,
Faculty Of Science,
University Of Cape Town

May 26, 2016

Abstract

Procedural generation is used extensively in the field of computer graphics to automate content generation and speed up development. One particular area often automated is the generation of additional colour and structural detail for existing 3D models. This empowers artists by providing a tool-set that enhances their existing work-flow and saves time.

3D surface structures are traditionally represented by polygon mesh-based models augmented by 2D mapping techniques. These methods can approximate features, such as caves and overhangs, however they are complex and difficult to modify. As an alternative, a grid of voxels can model 3D shapes and surfaces, similar to how 2D pixels form an image. The regular form of voxel-based models is easier to alter, at the cost of additional computational overhead.

One technique for generating and altering voxel content is by using Cellular Automata (CA). CAs are able to produce complex structures from simple rules and also easily map to higher dimensions, such as voxel datasets. However, creating CA rule-sets can be difficult and tedious. This is especially true when creating multidimensional CA. In our work we use a grammar system to create surface detail CA. The grammar we develop is similar to formal grammars used in procedural generation, such as L-systems and shape grammars.

Our system is composed of three main sections: a model converter, grammar and CA executor. The model converter changes polygon-mesh models to and from a voxel-based model. The grammar provides a simple language to create CA that can consider 3D neighbourhoods and query parameters, such as colour or structure. Finally, the CA executor interprets the produced grammars into surface-oriented CAs. The final output of this system is a polygon-mesh model, altered by the CA, which is usable for graphics applications.

We test the system by replicating a number of CA use-cases with our grammar system. From the results, we conclude that our grammar system is capable of creating a wide range of 3D detail CA. However, the high resolution of resulting meshes and slow processing times make the process more suited to off-line processing and pre-production.

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.

Acknowledgements

Firstly, I would like to thank my supervisors, James Gain and Patrick Marais, for all of their wisdom and guidance. Without their help, this work would not have been possible.

I would also like to thank my family for supporting me through my years of study. Without their support I would never have had the opportunity to pursue my academic goals.

A big thank you to the Computer Science department at UCT. I would especially like to thank my comrades in the post-graduated laboratory for their companionship, guidance and occasional (or sometimes frequent) distractions.

I would also like to thank the National Research Foundation (NRF) and the University of Cape Town for the financial assistance during this project. I would like to note that the opinions expressed and conclusions arrived at, are those of the authors and not necessarily attributed to the NRF or UCT.

Finally, I would like to thank the creators of all of the open-source and publicly available frameworks and artistic assets used within this work.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

The demand for bigger and better user experiences in computer graphics industries, such as video games, movie special effects and virtual environments have let to increasingly large and more detailed environments. This demand has been supported by a steady increase of available computational power capable of simulating and rendering larger environments. Naturally, the increased range and detail of these environments requires more and more content to be created. This accelerating drive for more content has significantly escalated the time and cost of development.

To mitigate the cost of increased content production, automated techniques for generating content are becoming more popular. A number of useful content generation techniques can be found in the field of *procedural generation*. Procedural generation is a collection of methods that generate content algorithmically and with minimal human interaction. Each method can create content from an initial set of inputs or parameters set by a user. The procedural algorithms can then produce content in a fraction of the time it would take a human designer. This can free designers to concentrate on more critical areas and reduce the number of overall content creators.

The addition of surface detail to 3D models is an example of content generation that is often automated. Adding surface detail to existing models can involve repetitive and small alterations, which can be tedious for an artist to do by hand. For example, adding rust or weather damage to a model of a ship can involve adding areas of similar detail, with the need for special care to avoid repetition and be sensitive to the context (eg. the sail of a ship does not rust).

Traditional methods of adding detail involve the creation of texture maps for polygon-based models. Texture maps, such as colour, lighting and displacement, are used at runtime to simulate surface effects. For example, texture maps emulate visual properties of different materials, such as cloth, metal or plastic. However, mapping texture maps onto the polygon mesh surface can be limiting and difficult to manage. For example, displacement maps can displace fragments of the surface but cannot recreate surface features such as occluding surfaces.

Voxel-based models and hybrid voxel-polygon solutions can better approximate these difficult features. A voxel-based model approximates a 3D surface/volume using a regular grid of volumetric elements, or cube volumes. In comparison to pure polygon-based solutions, voxel grids give simpler control of fine surface features at the cost of extra computation.

1.1 Problem description

This dissertation aims to contribute new solutions to one aspect of content production, the generation of surface detail in 3D models. The generation of 3D surface structures, such as occluding surfaces can be realistically approximated using voxel-based models. However, creating these surfaces using procedural methods can be difficult, especially when existing geometry is present. Additionally, procedural techniques can often require expert skill to create and control.

To address the problem of surface detail generation, we investigate the use of a procedural technique called *voxel-space cellular automata*. Voxel-based cellular automata (CA) are often used to automatically create volumetric structures/surfaces. Additionally, we look into creating CA that are sensitive to context of the models surface in terms of both texture (primarily colour) and structure. However, the creation of these 3D CA can be difficult to conceptualise and control.

To create the 3D surface-based CA, we investigate the use of procedural grammars. Procedural grammars, such as L-systems and shape grammars, are often used to create 2D cellular automata. However, the creation of 3D cellular automata using procedural grammars is still a challenge. We investigate extensions to these existing 2D CA procedural grammars to make creating 3D CA easier.

1.2 Grammar-based detailing framework

To address the issues raised in Section 1.1 we create a voxel engine which combines a number of tools into a single production chain, which contains tools for 3D model format conversions, creating cellular automata and executing the cellular automata on voxel-based models.

As the most popular formats for 3D models are polygon mesh-based, we need to be able to convert to and from a voxel-based format. We leverage the open-source library OpenVDB, and use their provided conversion tools to handle the format conversions. However, to retain colour present in the texture maps of mesh-based models, a number of extensions are required.

To create the CA we investigate the use of a procedural grammar. This grammar is based upon other procedural grammars, such as L-systems. This inspiration is evident in many of the features, such as the use of production rules, probability and context sensitivity. The grammar is able to create 3D CA, with symbols that represent voxel-neighbourhood navigation in 3D. A CA rule is represented by two sections, which first evaluate voxels in a neighbourhood, and then apply modifications to them depending on the evaluation outcome. Additionally, each CA grammar can contain multiple rules for a complex multi-pass CA.

A number of CA are created through a process of trial and error. When a CA is ready to be tested, the CA grammars are parsed and loaded into the voxel engine. A user can then execute the CA on models that have been converted. As CA can be tricky to control and very unpredictable, the CA are prototyped in near real-time (the grammars are parsed just before execution) on a low resolution grid. Once the grammar with desired behaviour is ready for macro testing, the resolution of the voxel is scaled up until the desired resolution.

A number of features in the engine allow for increased usability. First, a voxel selection mechanism allows the user to run CA on subsections of a model. Second, a tangent-space orientated surface navigation uses the tangent-space axes of the model instead of the standard global axes. This can simplify the input grammars, and provide more context-awareness. Finally, once a voxel-based model has been detailed, it can be converted into a mesh-based model.

1.3 Research questions

Our research evaluates the feasibility of using cellular automata for producing surface detail on 3D models. To evaluate the system the following research questions are addressed, with the final question highlighting the main focus of this work.

Can voxel based CA be used to create surface detail in 3D models?
While 3D CA are shown in other work, we need to determine whether 3D CA can be successfully applied in the context of surface detail generation. In this work, we focus our investigation to CA that can add structure and colour detail to existing geometry.

Can a grammar-based system be used to recreate CA present in other work?
Procedural grammars are often used to create CA, however, these are often limited to 2D or simple volumetric shape representations. We investigate how grammars can be used to create 3D CA, that are contextually aware of surface orientation and navigation. To do this we look at recreating 3D surface-based CA in a grammar-based framework.

Additionally, we investigate how the use of a grammar-based approach can be used in prototyping CA, which can often be tedious and difficult to control. The grammar is used to rapidly prototype CA at low resolutions, to measure if it can reduce the time it takes to find a stable CA.

Does our system produce surface detail that is almost impossible/extremely difficult to produce with mesh-based models?
The system we create needs to be capable of creating 3D detail that is almost impossible/extremely difficult to produce with texture mapping techniques. The mapping techniques project information in 3D to a 2D representation, which can result in a loss of information. This is especially evident when trying to add additional 3D detail to 2D texture maps. For example, occluding surfaces, such as overhangs and cavernous structures, can be almost impossible/extremely difficult to be accurately represented by mapping techniques.

3D detail can be created through the use of erosion and structure creating CA, which are capable of creating complex structures that cannot be recreated with texture maps. The system we create must be capable of accurately representing such structures.

To evaluate the research questions above we use two testing techniques: the creation or recreation of CA presented from other work and system performance testing.

In the CA creation tests, each CA is created using a procedural grammar. We choose a large variety of cellular automata to recreate to ensure that our procedural grammar system does not hinder the creation of CA. The grammars are transformed into executable voxel-based CA, which are executed over voxel-based models. A variety of public domain 3D polygon-mesh models are used to measure the robustness of the CA. These mesh-based models are converted to a voxel-form for processing, and finally converted back to a mesh form.

Performance tests are used to give an indication of the memory and processing requirements of the created CA. These tests are applied to each of the CA creation procedural grammars and include measuring execution time, memory requirements, processing performance and parallel speed-up.

1.4 Contributions

This research makes the following contributions.

A procedural grammar that can represent 3D CA.
We introduce a procedural grammar to produce voxel-based 3D CA that is based on L-systems and shape-grammars. The grammar defines each rule of the CA with a number of features to make writing CA easier. These features borrow from the typically 2D procedural grammars, with extensions to make voxel and 3D evaluation easier. For example, the grammar uses overloaded symbols for voxel evaluation and modification, as well as using short-hand 3D neighbourhood navigation symbols.

The neighbourhood navigation symbols can stack within scope bracers {} (similar to bracers in programming languages such as C++ or Java), as well as predefined neighbourhoods (such as spherical or cubic neighbourhood shapes). The complex stacking scope allows simple to advanced access to neighbours, which includes directly connected, indirectly connected or more neighbours.

A surface-oriented extension to the 3D CA grammar.
We introduce a tangent-space orientation extension to the 3D CA. This method augments the CA produced by grammars with neighbourhood navigation symbols. Here, the CA uses tangent-space axes u, v, w for the navigation symbols, instead of the grid's global x, y, z axes. The tangent space axes are calculated from the surface normal, tangent and bi-tangent vectors.

The tangent-space axes allow the CA to track the surface of input models, which expands how context-aware the CA is and simplifies navigation calculations. For example, a surface context-aware CA could track across a slanted plane using only the u, v (or tangent, bi-tangent) vectors. In contrast, a similar CA with a standard orientation would require calculations in all three x, y, z axes.

1.5 Overview of thesis

This thesis is structured as follows. Chapter 2 provides a background for related work to our own. Chapter 3 describes the voxel engine, which is used to manage the 3D models and convert between different model representations. Chapter 4 describes the grammar system we use to generate 3D cellular automata. Chapter 5 covers the cellular automata execution framework, which specifically examines how the 3D CA are executed by the voxel engine. Finally Chapters 6 and 7, present, analyse and discuss the results of the overall system, reinstate our findings and suggest some areas of future work.

Chapter 2

Background

In this chapter we discuss the relevant research that provides background knowledge for our work. We start by introducing the field of procedural generation in general (Section 2.1), while specifically discussing how solid textures (Section 2.4), voxel grids (Section 2.3), cellular automata (Section 2.5) and procedural grammars (Section 2.2) have been used within the field.

To begin, we set the scene by introducing procedural generation in Section 2.1. We note how important it is to our work, while highlighting a number of relevant application areas. Section 2.4 then looks at automatically generating volumetric textures. This provides the closest related work to the goals of this research. Next, Section 2.3 focuses on voxel grids, which are conceptually similar to volumetric textures. We discuss the circumstances under which it makes sense to use a voxel grid in procedural generation, instead of the more commonly used 3D model and texture representation. Certain techniques, such as cellular automata, are easily mapped since they are defined on a regular grid. In Section 2.5, we describe cellular automata and how they can be used in procedural generation. Finally, creating specific outcomes from procedural techniques can be difficult. Section 2.2 shows how the use of grammars in procedural generation can make this process simpler.

2.1 Procedural generation

Procedural content generation, or procedural modelling, is a collection of methods that algorithmically create content in computer graphics. This is typically visual content, like 3D models, but can also apply to other compositions, such as audio, puzzles and artificial intelligence behaviours [23]. Procedural generation uses algorithms that allow creation through minimal user interaction, typically limited to simple parameter modifications. These algorithms are used to simulate many different forms of content, including natural, mechanical, painted and cartoon-style scenes or models. They also allow the user to rapidly create a multitude of large and complex assets, which reduce the overall cost of manually producing them. This streamlines the work-flow of asset creators and allows them to focus more time on other areas.

The following subsections introduce some important applications of procedural generation. These serve to illustrate the range of methods available and to provide some context for our own procedural modelling system.

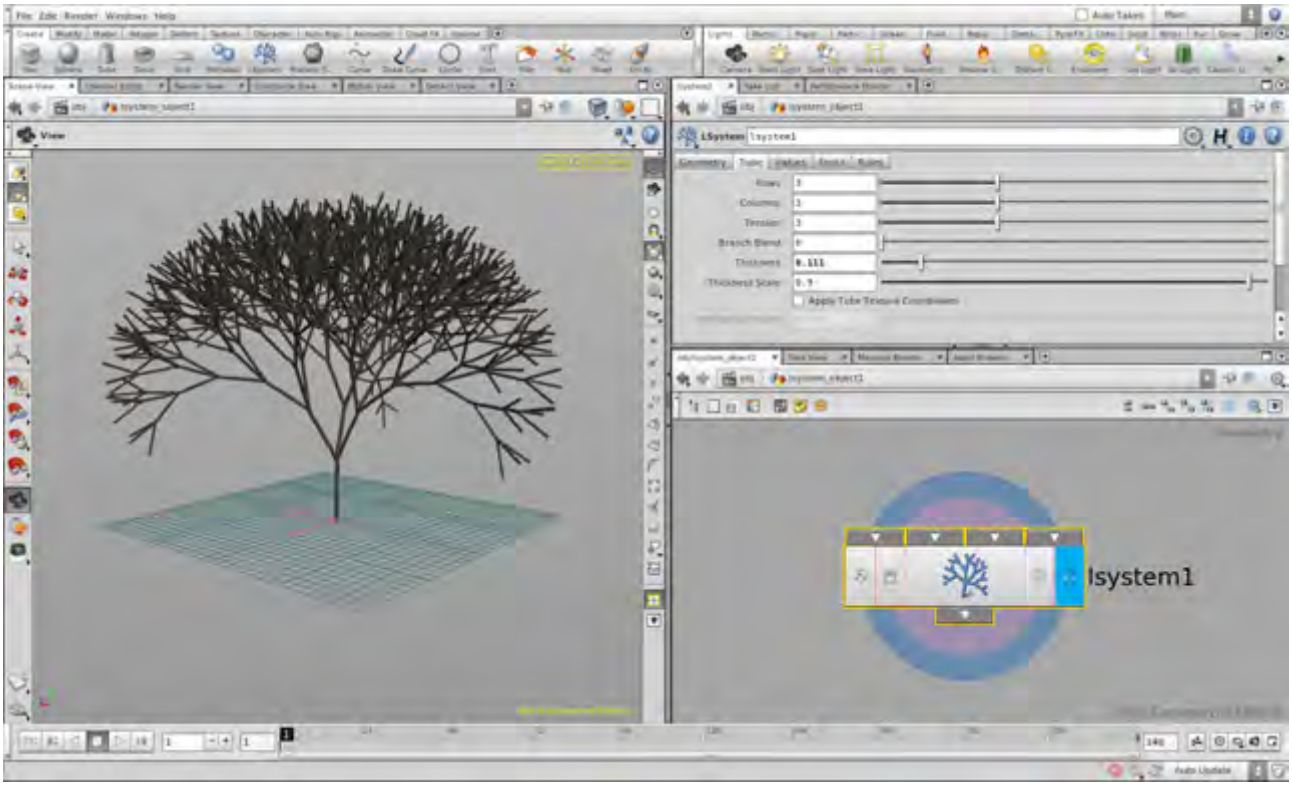


Figure 2.1: *This figure is a screen-captured image of the node-based user interface from Side Effect's Houdini. In this example a single node is shown, which produces a basic L-system. Within the GUI, the left view-port displays a tree model produced by an L-system. The bottom-right view-port shows a visual representation of the L-system node.*

2.1.1 Procedural generation tools

The demand for digital content has increased exponentially with the growth of the computer graphics industry. The video game *Grand Theft Auto V*, for example, was reported to have over 1000 people on staff during development [1]. The majority of these people are content creators, due to the amount of content in the explorable environment. This demand has also driven the need for techniques that are better at reducing the increasing cost and development time. This has led to the development of many commercial applications to generate content. Houdini [2], by Side Effects Software , and Softimage [3], by Autodesk, are examples of applications that provide a collection of modelling tools for film and computer game production. These tools are used to pre-generate content before release of the film or game. A subset of these tools, such as Softimage's ICE and Houdini's Procedural Workflow, provide node-based procedural generation tools. These node-based approaches provide simple interfaces that allow creators to easily combine procedural techniques together. Typical procedural components include the generation of cloud volumes, landscapes, textures and tree models. An example of a simple node within Houdini is shown in Figure 2.1. Besides broad goaled modelling packages, other middle-ware solutions exist with a more specific focus. *SpeedTree* by IDV Inc. [4]is one such system. SpeedTree focuses on rapid creation of animated vegetation and tree models. It is widely adopted by industry and incorporated into many modelling packages and game engines. This has led to the

[1]Michael French, *Inside Rockstar North - Part 2: The Studio*, `http://www.webcitation.org/6TJjNudD9`. [Online; accessed 15-October-2014]

[2]Side Effects Software, *Houdini*, `http://www.webcitation.org/6TLCpPYVx`, [Online; accessed 15-October-2014]

[3]Autodesk, *Softimage*, `http://www.webcitation.org/6TdX74Lwr`, [Online; accessed 13-May-2013]

use of SpeedTree in various high profile feature films and video games. Another middle where solution is *CityEngine* by ESRI [5]. CityEngine provides an intricate framework to generate large-scale city models. The framework incorporates tools that include texturing, visual editing and city planning or zoning. *Terragen* by Planetside Software [6]is s a similar package, which focuses more broadly on generating landscapes and environments. Figure 2.2a shows an example landscape generated using Terragen 3.

(a) *This is a showcase image of a landscape created through Planetside Software's Terragen.*

(b) *This figure is a screen captured image of a procedurally generated environment in Proteus.*

(c) *This is a screen-shot from the video game .kkrieger, by demo group .theprodukkt.*

(d) *This figure is a screen captured image of a procedurally generated voxel environment in Minecraft.*

Figure 2.2: *Various screenshots from procedurally generated enviroments.*

2.1.2 Real-time procedural generation

Similar methods to those employed in pre-production tools, are used at runtime in computer games. These methods are required to be as fast as possible to limit waiting time, unlike pre-generated content, which often emphasises quality over speed. The independent video game *Proteus* by Ed Key and David Kanaga [7], creates an open-ended audio-visual experience, with a focus on exploring a procedurally generated world. Proteus generates a virtual world before every play-through, with multiple environment types to explore. One of these environments is shown in Figure 2.2b. Proteus also procedurally generates music during game-

[4]IDV Inc., *SpeedTree*, `http://www.webcitation.org/6UiUYFw4o`, [Online; accessed 10-November-2014]
[5]ESRI, *CityEngine*, `http://www.webcitation.org/6TLBGnBUI`, [Online; accessed 15-October-2014]
[6]Planetside Software, *Terragen*, `http://www.webcitation.org/6UiehUX6M`, [Online; accessed 15-October-2014]
[7]Ed Key and David Kanaga. *Proteus*. `http://www.webcitation.org/6TdMvPzaW`. [Online; acces1ed 17-October-2014]

play according to environmental interactions, such as wind blowing, grass growing and animals walking. Procedural content generation is taken to the extreme within a movement called the *Demoscene*. This computer art subculture aims to produce very small applications that result in large audio-visual experiences or games. A computer game *.kkrieger* by demo group *.theprodukkt* manages to create a playable first-person shooter in a total application size of 96Kb [8], even smaller than the size of an average texture. This low memory footprint is achieved solely through procedural techniques that generate all the content at runtime. This saves space at the cost of greater pre-processing time before each play-through. The content generated in *.kkrieger* includes textures, models, animations, particle effects, sound effects and music. Some of the visual content in .kkrieger is highlighted by the screen-shot in Figure 2.2c.

2.1.3 Search-based procedural generation

Togelius et al. [63] describe a taxonomy of search-based procedural content generation. In this work they define the growing use of search-based algorithms and machine learning techniques in procedural generation. Additionally, they make the distinction between *constructive* and *generate-and-test* procedural generation algorithms. Constructive methods generate content, and do not test the outcome. However, the results do need to be evaluated by means, such as, user evaluation. In contrast, generate-and-test methods incorporate a test mechanism with the generation process. If the test fails, the result is discarded and regenerated until the desired outcome is achieved.

Search-based procedural content generation is specific type of generate-and-test algorithms. Here, the test function does not accept or reject results in a binary method. Rather, methodologies are borrowed from machine learning and evolutionary computation. A fitness function assigns a "fitness" grade to the result, and the results are ranked according to their fitness values. The best candidates are kept and the worst are discarded. Copies of the best candidates are mutated by random modifications, and combined using crossover techniques.

This work falls into the constructive procedural generation categorization, where the results are evaluated by the grammar designer at each iteration. This constructive approach can take overall less time, without the need for automated testing. However, this could result in a more time consuming (and frustrating) process for the designer.

2.2 Procedural grammars

Procedural grammars are commonly used as tools within procedural generation [23, 30, 57]. The ability to create complex forms from simple rules, has made procedural grammars successful modelling tools. Procedural grammars work by applying symbol replacement rules to an axiom, or starting symbol. From the replacement rules, a starting symbol can multiply or change form completely, creating a new string of symbols. The rules then act upon the new string of symbols, creating even more information. The symbols can represent many virtual objects, such as branches in L-systems, shapes in shape grammars or grid areas in cellular automata grammars. The following sections describe a few of these grammars and symbol types.

2.2.1 Lindenmayer Systems

A *Lindenmayer System* (L-system) is a procedure for creating complex structures by simulating growth. A grammar rewriting procedure accomplishes growth simulation through a set of replacement rules. Biologist Aristid Lindenmayer, first introduced L-systems [36] as a grammar capable of describing plant growth. Later,

[8] .theprodukkt. *.kkrieger*. `http://www.webcitation.org/6TcWrje4A`. [Online; accessed 26-October-2014]

Przemyslaw Prusinkiewicz proved their importance for modelling general growth systems [52]. Prusinkiewicz shows how versatile L-systems are by applying them to a variety of applications including: turtle graphics, Koch curves and realistic pictures of plants.

The L-system grammar uses a starting symbol, or axiom, and a set of match-and-replace rule sets, or productions. Production rule sets replace matched symbols by a set of symbols, possibly empty. A symbol is *terminal* when no production matches exists to change it. Alternatively a symbol is *non-terminal* when a production can match it. Applying all the productions to the current set of symbols is a *generation*. When applying productions, the process finally halts when only terminal symbols remain or after a set number of generations. This process is similar to a formal (Chomsky) grammar. L-systems differ from formal grammars by expanding each production in parallel instead of sequentially. A simple L-system example is shown in Algorithm 1.

Algorithm 1 A simple L-system

1: **Alphabet** : a, b
2: **Axiom** : a
3: **Rule1** : $a \rightarrow ab$
4: **Rule2** : $b \rightarrow ba$
5: **Output** :
6: $n = 0 :$ a
7: $n = 1 :$ ab
8: $n = 2 :$ $abba$
9: $n = 3 :$ $abbabaab$
10: $n = 4 :$ $abbabaabbaababba$
11: $n = 5 :$ $abbabaabbaababbabaababbaabbabaab$

Once a list of symbols are created by the L-system grammar, it can then be interpreted to form content. The most common form is a visual representation, created using turtle graphics [52]. In this representation lines are drawn when moving a turtle, or cursor, according to different symbols within the grammar. These movement symbols typically change the direction of the turtle and indicate whether to draw a line or not. Together the collection of produced lines form a path of the moving turtle. The path serves as a basis, which is typically extended and used in further representations. For example, the path may be converted into textured cylinders to represent a 3D tree.

L-systems are commonly extended with additional functionality. Important extensions include: stochastic behaviour, context sensitivity and parametrisation. *Stochastic behaviour*, introduced independently by Yokomori [67] and Eichhorst and Savitch [16], allows the creation of random output from a single rule set. By controlling or limiting the random behaviour, content variation can be gained while maintaining similarity.

Environment interaction is also an important factor when considering growth simulations, which led Měch and Prusinkiewicz to describe *context sensitivity* within L-systems [43]. In real-world situations growth is often aided or constrained by limited resources in the surrounding environment. To model this, rewards or constraints are applied from resource competition, including: space, light and water availability. A similar extension is *self-sensitive* grammars. Parish and Müller [47] use a self-sensitive grammar within the context of road generation. This enables them to create cycles in road systems, to create a road network instead of independent branches.

Finally, *parametrisation* of L-system rules [53], allow additional information to be used during generation. This information could, for example, describe the thickness and tapering of branches, bark textures or growth knots.

2.2.2 Shape grammars

Another grammar system often used in procedural generation is *shape grammars*. Shape grammars were introduced by Stiny and Gips [60], as a tool to create complex shapes. This grammar system consists of rules that combine one or more shapes, in order to create a new shape. Once these new shapes are created, the new shapes can combine to create even more complex shape combinations. This process is emergent, capable of creating interesting complex models from simple shapes. For example, a cylinder and cone can combine to create a simple tree model.

One drawback of using shape grammars in this way, is interpreting newly created shapes. When new shapes are created, it can be difficult to identify or categorise the resulting shape. This can lead to unnecessary repetition and inappropriate use of shapes. A common way to address this is to use symbolic representations of shapes, similar to L-systems and formal grammars. Resulting shapes are then only determined when the grammar is complete.

Extensions to the shape grammar definitions are also introduced. Similar to L-systems, the most popular extensions are: parametrisation, context sensitivity and stochastic behaviour. An overview of these extensions are available in Section 2.2.1. Similar to L-systems, these extensions alter how the shapes are produce.

With *stochastic behaviour*, random behaviour is introduced when creating each new shape. The first type of randomness chooses between shape(s) to replace a selected shape. This could alter the quantity, type or frequency of replacing shapes. Another aspect of stochastic behaviour is to add variety to parameters. This could include dimensions, colour or any other parameter of the shape.

2.2.3 Split grammars

Split grammars, introduced by Wonka et.al [65], are a variant of shape grammars. Split grammars also operate though shape replacement, however they work through the subdivision of shapes. This technique works particularly well for the generation of building façades, such as windows. However, this technique performs less well when creating the structure of buildings or interiors. This is primarily due to the enforced regularity of the split-grammar results, which is less appropriate in these areas.

Muller et.al [40] provide a few important extensions to basic split grammars. Notably these include an occlusion test and snap-lines. The occlusion extension tests for shape intersection. This can help to determine where certain shapes, such as doors and windows, could be placed. The snap-lines extension finds lines which span across façade (or shape) boundaries. This can help align elements across the surface of complex models.

2.2.4 Example-based grammars

Instead of manually creating grammars, automated techniques are used to generate or reverse-engineer grammars from input examples. Muller et. al. [41] show how images can be used as exemplars. In this work a split grammar is derived from a side-view image of a building. First, a 3D model is generated from an image, which is then analysed to produce the split grammar rules. This technique also benefits from the colour texture information from the image, which can enhance the look of façades.

In contrast, the method by Bokeloh et al. [5] generates a shape grammar from a 3D model exemplar. In an analysis phase, the technique subdivides the model into smaller shapes along locally symmetric areas. These smaller shapes are used to define the rules of a shape grammar, which can generate a new model with a similar style and look to the exemplar. This technique can prove to be quite effective on models with large amounts of symmetry, such as buildings.

2.2.5 Grammatical evolution

In addition to grammar usage in procedural generation, evolutionary computation can be applied to automate the process of grammar creation. The combination of evolutionary computation and grammar design is called grammatical evolution (GE) [45].

Shaker et. al [56] use GE to procedurally generate levels for the plat-former game Super Mario Bros. Here, the level is broken down into a two-dimensional grid of blocks, or prefabricated "chunks" of blocks. A context free-grammar is used to represent the construction of chunks in a level. The grammar is then evolved according to constraints, such as gap length and platform height. The results are evaluated according to to a fitness function, where the main goal is to create an acceptable amount of chunks. This results in procedural grammars which are reusable and modifiable by designers.

2.2.6 Formal grammars

The study of syntax investigates how sentences are constructed in particular languages [12]. The syntax governs the structure of the language, and does not require that the text has meaning or makes logical sense. A formal, or Chomsky, grammar is a set of rules that describe how to form text that is valid according to a languages syntax. These rule sets are known as productions. The symbols within each language are known as the alphabet, much like the "a-z" alphabet of the English language. Formal grammars may only apply productions serially and not in parallel.

The Chomsky hierarchy is a hierarchy of classes of formal grammars [11]. These classes are recursively enumerable, context-sensitive, context-free and regular. These are also known as types $0 - 3$, respectively. In the following subsections we will look at each of these grammar types.

Recursively enumerable

Recursively enumerable, also known as type-0, grammars are the superset of all grammars that can be computed. These include context-sensitive, context-free and regular grammars (types 1-3). The Type-0 grammar category is known as the unrestricted grammar category, as it is the least restrictive grammar class. A language is said to be computable (type-0) if it can be recognised by Turing machines. Otherwise stated, given infinite time a recursive function could generate all of the strings in them. The languages are of the form $A \rightarrow B$, where A and B are strings of terminal and non-terminal symbols and A is not empty.

Context-sensitive

Context sensitive, or type-1, grammars are a subset of type-0 grammars. This subset of grammars are capable of describing natural languages. They are recognizable by linear-bounded automata, which is a Turing machine which has left and right bounds that may not be crossed. They are of the form $YAZ \rightarrow YXZ$, where A is a non-terminal, and X, Y and Z can contain both terminals and non-terminals. Both X and Y may also be empty.

Context-free

Context-free, or type-2, grammars are a subset of type-1 grammars. They are frequently used to describe programming languages. Type-2 grammars are recognizable by non-deterministic push-down automaton, which is similar to a linear-bounded automata but can also interact with the symbol at the top of an infinite

stack. A context-free grammar has only non-terminals on the left side of a production, and may transform into a number of terminals or non-terminals. For example the form $A \rightarrow xyz$, where A can only be a non-terminal, and x, y and z can contain both terminals, non-terminals or the empty set.

Regular

Regular, or type-3, grammars are the most restrictive set of grammars, a further subset of type type-2 grammars. These grammars can also describe programming languages and are used for string processing. They are described in a similar way to mathematical sets, with stings created by characters in its alphabet put to together by concatenation, parentheses, the "or" operator and the Kleene star "*" operator.

Finite

A finite grammar is the set of grammars that can only describe finite sets. This is a subset of Regular grammars without operations such as the Kleene star, which can create infinite sets.

Procedural grammars

L-systems, which are a type of formal grammar, do not fit neatly into the Chomsky hierarchy. Instead, they can be grouped into two main classes, OL and IL [53]. These partially intersect several main classes within the hierarchy. L-systems apply productions in parallel, which is a distinguishing feature from standard formal grammars. Additionally, L-systems apply a form of symbol rewriting, where symbols in the left side of a production are replaced by symbols on the right.

OL-systems, or context-free L-systems, is the most restrictive class of L-systems. The deterministic versions of these L-systems are known as DOL-systems. In this class, symbols in the left side of a production contain empty left and right contexts (symbols at either side of the symbol). It should be noted that context free OL-systems are capable of creating languages that context-free Chomsky grammars can not. This is a benefit of the parallel rewriting capability of L-systems.

IL-systems, or context-sensitive L-systems, are the least restrictive class of L-systems. Similar to OL-systems, context free IL-systems can generate languages that context-free Chomsky grammars can not. Additionally, DIL-systems are the deterministic versions of these L-systems.

Shape and split grammars have a similar class categorisation to L-systems. Both shape and split grammars apply rewriting rules to shapes, which are represented by symbols. Additionally, they can be context-free, context-sensitive, deterministic and non-deterministic. Shape grammars deviate slightly from L-systems by running all rule sets until only non-terminal symbols remain, rather than running for a fixed amount of iterations. Additionally, matched shapes within a context are often merged into a new shape, which the grammar treats as a new symbol. In contrast, split-grammars split shapes into new sets of shapes, which is more similar to the rewriting rules of L-systems. The shape splitting is applied for a number of generations, similar to L-systems.

2.3 Voxel grids

In our system, we utilise voxel grids since they are easy and intuitive to manipulate and provide powerful 3D modelling functionality. A voxel grid is a set of discrete position values, arranged on a regular grid, which represent a solid 3D object. Each position is associated with a small cubic region, a voxel, analogous to the area attached to a pixel coordinate in a 2D image. An example of a voxel grid is shown in Figure 2.3.

Figure 6.14a shows a standard mesh model of a cube. This mesh model stitches together polygonal shapes, such as triangles and quadrilaterals, to form a surface in 3D space. In contrast 6.14c shows the voxel grid representation of the same cube. Note the aliasing artefacts on the edges of the chequered pattern, introduced by the coarse voxel representation. In the final image 6.14b, each voxel is highlighted by a yellow wire mesh, with lines connecting each vertex along the cubes edges.

Voxel grids have a uniform structure, with each grid element having regular sizes and positions. Regularity makes finding and storing voxel grids consistent and therefore predictable. This contrasts with methods, such as meshes or point cloud models, which provide faster processing and memory benefits at the cost of additional complexity.

Voxel grids are used with procedural generation in many applications including medical imaging, computational astronomy and computer animation [25, 48]. Voxels are also used as a basis for many procedural generation techniques, ranging from texturing and modelling to scene construction. In medical imaging, voxel representations have been used for applications such as medical simulation [38] and texturing [51]. Kniss et al. [31] use voxel grids with volumetric rendering to accurately render translucent models as well as volume displacement to produce effects like fluffy clouds and mouldy rocks. Many procedural techniques have been used to texture volumetrically and on surfaces. For example, Buchanan [8] demonstrate this by procedurally generating volumetric wood textures in voxel models based on off-axis radial functions.In computer games voxels are popularly adapted by independent game developers. These developers use voxels to create computer games in art-styles reminiscent of past games, constrained to low resolutions. *Voxatron* [9], created by Lexaloffle Games, is an example of a collection of games using voxels as an art-style. The voxel engine created by Lexaloffle also acts as game creation toolkit. A few of tools in the tool-kit include voxel modelling and animation systems. Another example is *Minecraft* [10], by developers Mojang, which uses voxel-like blocks to create a procedurally generated landscape, as shown in Figure 2.2d.

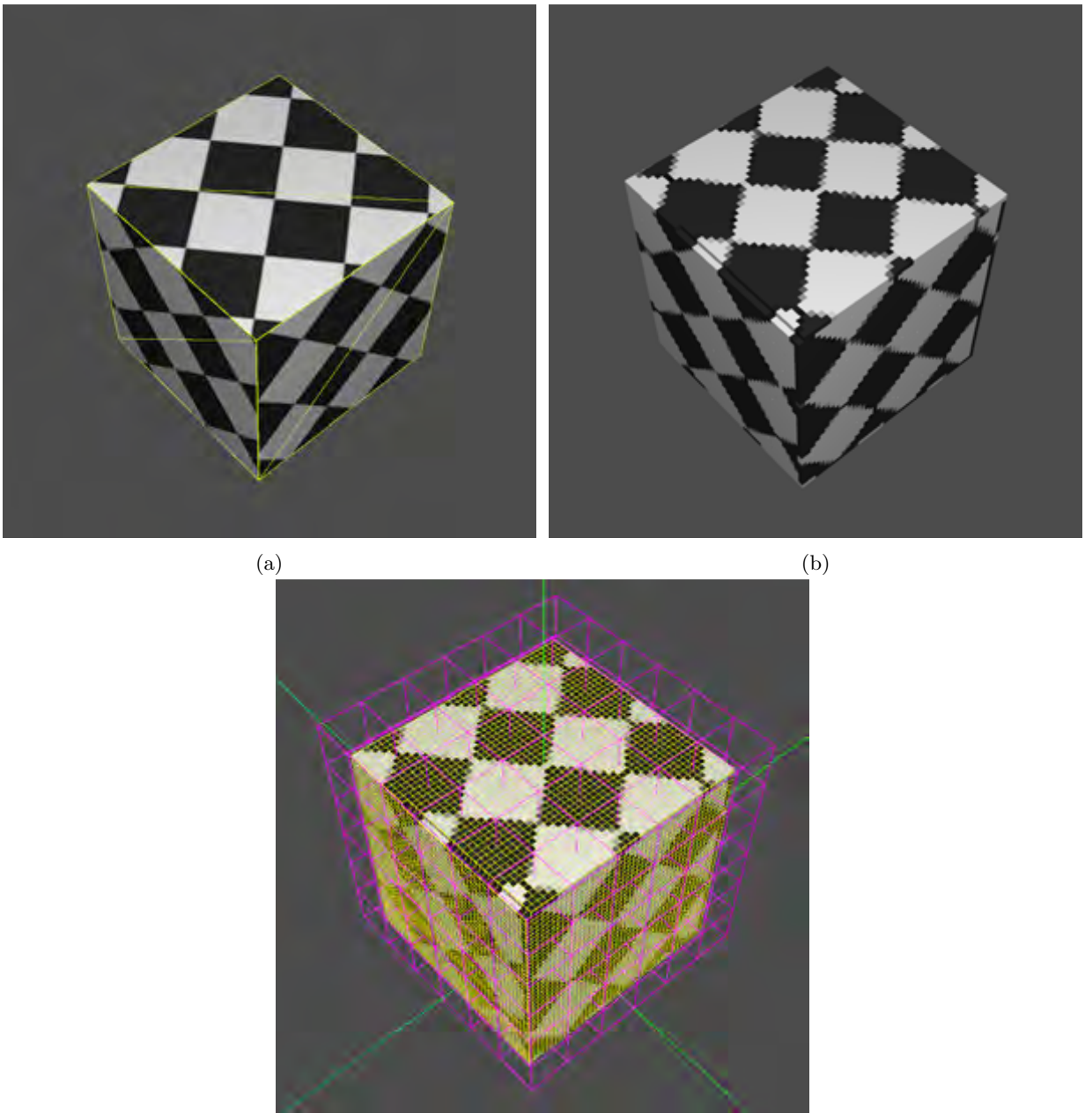
2.3.1 Voxel storage

Using voxel data introduces difficulties when storing, processing and managing voxels. These problems arise because of the enormous number of voxels contained in the average dataset. To mitigate them, complex data-structures are used. In a simple example, a grid of size of 128^3 voxels (typically regarded as a small dataset) contains over two million voxels. If each voxel contains a simple integer of information, then the dataset would encompass 8 megabytes of memory. The voxel data size quickly grows as $O(N^3)$, with average dimension size N.

A nave method of storing a voxel grid, is to create a three-dimensional array. Each element in the array holds all the information of a single voxel. One advantage of this method is that the position in the array easily maps to the elements world position. However, the storage of redundant information, such as inactive or similar voxels, outweighs this advantage. One way to combat redundant information is by compressing the array using data compression. For example, Xu et al. [24] use run-length encoding to compress 3D textures. The run length encoding groups together elements with similar grayscale values. From these groups they are able to extract desirable features contained in the volume While compression is useful in well defined datasets, such as medical scans, it proves less useful in other situations. For example, random datasets with high entropy make compression less effective. This is because clustering similar elements is far more difficult. Similarly, it is inefficient to continually process and maintain compressed data: with, each change the data will need to be uncompressed, processed, then re-compressed again.

Hierarchical tree data-structures combat these issues by storing a hierarchy of spacial nodes [54]. Each node represents an area, or volume, of space, which is recursively subdivided according to a heuristic. The node structures created are only store information about active regions, by subdividing only when needed.

[9] Lexaloffle Games. *Voxatron*, `http://www.webcitation.org/6TOOaNq8V`, [Online; accessed 17-October-2014]
[10] Notch Development AB. *Minecraft*, `http://www.webcitation.org/6TOOVjrXP`, [Online; accessed 17-October-2014]



(a) (b)

(c)

Figure 2.3: *Three figures showing an example of voxel representations produced using OpenVDB: (a) a cube mesh with each polygon outlined in yellow, (b) a voxel representation of the cube mesh, (c) a visualisation of the sparse octree. Yellow lines show the finest level of the octree, the bounding boxes of individual voxels. Pink lines show the next node level of the octree, each pink bounding box (or node) containing a chunk of voxels. Similarly, the green bounding boxes contain chunks of the pink nodes.*

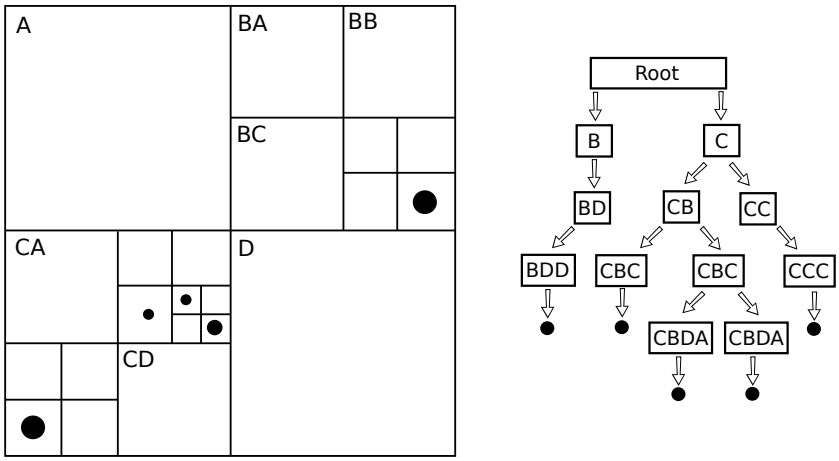


Figure 2.4: *A graphical and spacial representation of an octree.*

Additionally quick access routes for processing are created by traversing through the hierarchy of nodes. Many architectures are used, some of the most popular techniques include: octrees, point region octrees (PR-octrees), kd-trees, and R-trees.

Octrees divide space (a cubic volume) into 8 cubes of equal size, this yields the root and first level nodes, respectively. The first level nodes split further into 8 cubes. This occurs recursively until the space defined by nodes is the size desired for individual voxels. Dividing volume up in this manner saves space, since subdivision only needs to occur in places containing active voxels.

Point region octrees (PR-octree) are an extension of octrees, where the volume does not necessarily split in the centre. The split occurs to make each divided space have a similar number of active voxels within it. This allows the resulting tree structure to be better balanced.

A *binary space partitioning* (BSP tree), divides space into two half spaces. Space is split by an arbitrary placed plane, where each binary node is made up of the space to either side of the plane. The separated space is recursively split by arbitrary planes. A special case of BSP tree, called a *Kd-tree*, only splits space with axis aligned planes. This is useful with voxel data to avoid aliasing issues caused by planes with arbitrary rotations. An advantage of these binary trees is that less information is stored at each node level in a binary tree than an octree equivalent. However, the depth of the tree increases, which increases traversal time.

A *bounding volume hierarchy* (BVH), is another type spatial partitioning tree. A BVH wraps objects in bounding volumes, which serve as the leaf nodes of the tree. The leaves are then grouped into sets according to proximity. At this point, a larger bounding volume wraps around each set. The process continues until it creates a single large bounding volume, which will act as the root node.

A special case of BVHs are *R-trees*, conceptually similar to a spacial B-tree. One typical feature of R-trees is limiting bounding volumes to axis-aligned boxes. Another feature is a high degree of leaf nodes to each node, similar to B-trees. The degree of leaf nodes makes a shallow tree, which reduces traversal time. However, this can also increase the information and search time at each node.

2.3.2 Voxel grid and polygon mesh conversions

Converting between voxel-based and polygon-based models is useful for algorithms and techniques suited to a particular representation. However, mapping between polygon meshes and voxel grids is a non-trivial and

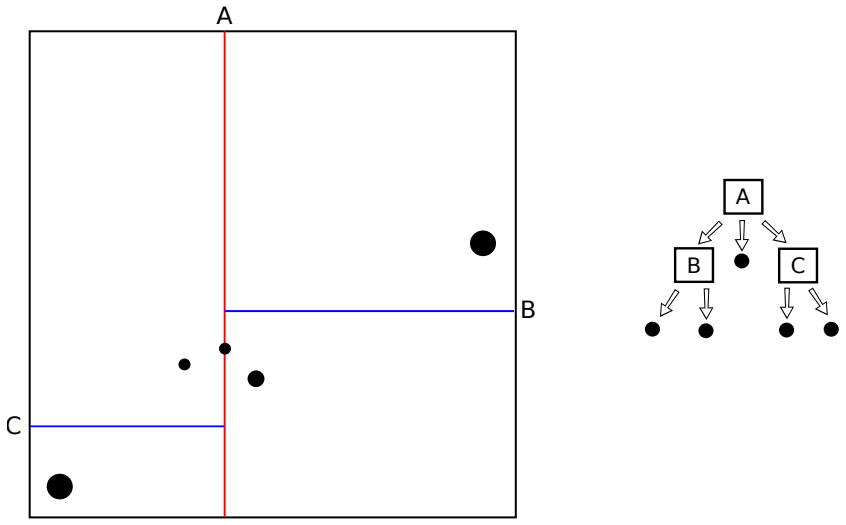


Figure 2.5: *A graphical and spacial representation of a kd-tree.*

often lossy process. These overheads often impede the use of techniques which are more naturally suited or faster in a different representation. This is shown in the trade-off between faster rendering using polygons versus realistic simulations using voxels.

Voxelisation

Voxelisation is the process of converting a polygon mesh model to a voxel model. One common method of accomplishing this is a precess called "scan conversion". This process converts the polygon model into a signed distance function, also known as a level set. A distance field is an area or volume of space, where each point (typically a discrete point) holds a scalar distance value to the closest point of the surface of an object. A signed distance field extends this with a a signed value representing if the point is inside the volume (positive) or outside the volume (negative). The signed distance field leads to two main types of voxelisation, "solid" and "narrow band". The narrow band method creates voxels according to intersections with the polygon surface. The solid method adds to this by creating voxels that are considered inside the narrow band, or internal to the model. Solid voxel methods as well as some narrow band methods require a surface to be non-manifold (for example a water-tight model). Figure 2.6, depicts narrow band and solid representations of voxel models.

The scan conversion method of et al. [6] uses constructive solid gemometry (CSG), comprised of super-ellipsoid primitives, to create a distance volume. In this method, first a narrow band of voxels is calculated around the surface of each super-ellipsoid. This is calculated by a depth first search through a graph of super-ellipsoids, finding the closest point from a voxel to a super-ellipsoid. If the distance is zero, then the voxel is said to be in the narrow band. Once the narrow band is calculated, distances are propagated to the remaining voxels through a fast marching method to create a solid voxel model.

Another method, presented by Chang et al. [9], only computes a signed distance field that is local to the narrow band distance. Voxels that intersect the polygon model's surface are found using a triangle list method. This method avoids expensive voxel-to-polygon distance checks by subdividing polygons into sub-triangles, which can roughly fit inside the bounding box of a voxel. Fast intersection tests are then calculated between these micro triangles and the voxels, by checking where the three vertices lie in relation to the bounding boxes of nearby voxels. As this process is run, each voxel builds a list of candidate triangles. From the candidate list, the shortest distance to the surface is calculated. This complete process quickly

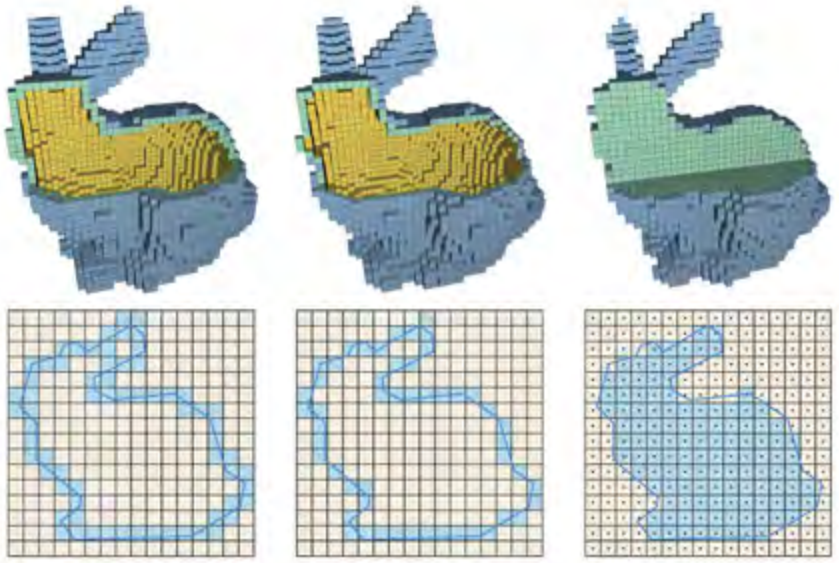


Figure 2.6: *This is an example of different voxelisation representations from Schwatrz et al. [55]. Image "c" shows a solid voxel model, while "a" and "b" show two different types of narrow-band voxelisations.*

builds a local signed distance field around the narrow band.

As voxelisation is a slow process for fine grids and large meshes, parallel accelerated voxelisation algorithms have also been created. Schwatrz et al. [55] demonstrate this concept with their work on GPU-accelerated real-time voxelisation. They present three different parallelised voxelisation methods. Their first method focuses on surface voxelisation. This method finds voxels that overlap with each triangle through intersection tests with the projection of the triangle and voxel's bounding box onto the x-z, x-y and z-y planes. Each triangle is tested, in parallel using individual threads, against voxels within it's bounding box. The second method focuses on solid voxelisation. This solid method is conceptually similar to rasterisation, where the object is rasterised into a multi-sliced frame buffer. In the same way as the second method, all the triangles are parallelised by a single thread each. In the same way as the narrow-band method, each column is tested against a triangles y-z projection. The column's center is projected onto the triangles plane, which is used to calculate the range of voxels in the column that are inside the surface. Lastly their final method uses a sparse-octree based voxelisation approach. A sparse-octree reduces the memory footprint of traditional octrees, by grouping voxels in close proximity that are similar to each other. In this method they use a binary voxel representation, which greatly simplifies the similarity grouping step. Figure 2.7 illustrates a two dimensional slice of a sparse voxel octree. The method creates the octree during voxelisation through a bottom-up approach. First the finest node level-0 (individual voxels) and next coarser level-1 nodes are determined. This is achieved through similar mechanisms to the second solid model-scan method. From these nodes the octree structure is built. Once the structure is set, the inside/outside parity is propagated by scanning through the octree.

3D surface reconstruction

Reconstructing a 3D polygonal surface from a discrete voxel model is useful for a variety of reasons. These range from rendering speed and memory requirements, to ease of use and compatibility with other software. This has lead to many different reconstruction methods and tools.

The best known and most widely adopted technique is the *marching cubes* method by Lorensen et al. [37].

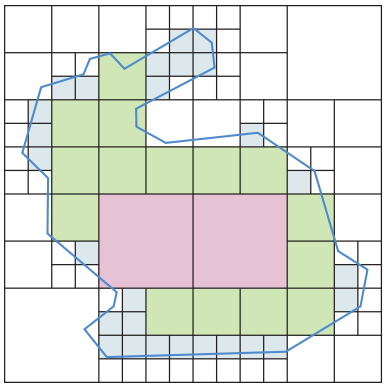


Figure 2.7: *This figure shows a two-dimensional slice of the sparse-octree voxelisation from Schwatrz et al. [55]. Each colour, blue, green and red, shows a different voxel size. This sparse methodology significantly reduces memory footprint of the octree.*

This method creates a polygon surface, according to eight corner vertices of a virtual cube. Each vertex of the virtual cube corresponds to a voxel, which is either active (inside the surface) or inactive (outside the surface). From the cube it can be determined if the surface intersects the cube or not. The surface intersects an edge of the cube if one vertex is active and the other is inactive. This creates 2^8 possible states of the cube (two states for eight vertices), which is reduced to fourteen states after removing duplicates (identical cube states after rotation or mirroring). By referencing a look-up table of these fourteen states, pre-calculated polygon groups can be created inside the virtual cube. The vertices of each polygon are made from the center points of the edges of the virtual cube. Once a cube is completed, the process "marches" to the next cube to be processed, ultimately creating a complete polygon mesh surface.

Marching tetrahedra is an alternative surface reconstruction method introduced by Nielson and Hamann [44]. This method is a modification to the marching cubes algorithm, to reduce ambiguity which results in topological errors. Additional edge intersections are added to marching cubes, by separating the cube into six tetrahedra. These are made by cutting the cube in half three times. The cutting lines are made along the diagonal line of each face pair (opposite faces of the cube). The edges are made where the cutting lines intersect the faces. An additional edge is create from the intersection of all three cutting lines, along the main diagonal of the cube. The tetrahedra reduces the lookup table size, only four points are necessary, while increasing the calculations with six tetrahedra to a cube.

Regularised marching tetrahedra is a further modification, which maintains the aspect ratio of triangles.

2.4 Solid texture synthesis

Texture synthesis is a method for automatically creating synthetic textures through computation [15]. Such an automated approach is important for applications in computer graphics, computer vision and image processing. For example texture generation can streamline a texture artists work-flow by automating parts that are tedious when carried out by hand, such as making textures tileable. A texture is said to be tillable if no seams are visible when copies of the texture are placed adjacent to one another in a tilled fashion (as seen in Figure 2.8). Due to the demand for synthetic textures the field of texture creation is broad, containing a variety of solutions and techniques [64]. A particular area of interest to our work is the procedural generation of volumetric (or solid) textures [51]. These textures are the 3D equivalent of a standard 2D texture or a stack of parallel 2D texture slices, as depicted in Figure 2.9. 3D textures are effectively voxel grids, which make voxel methods also applicable to them.

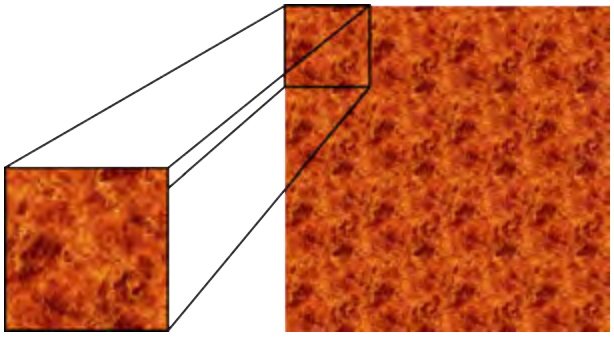


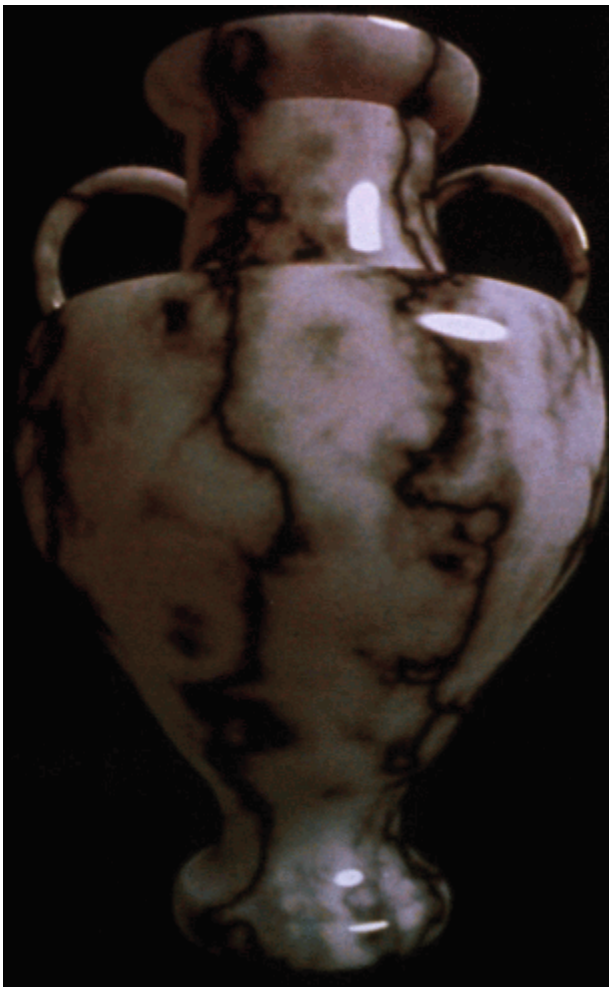
Figure 2.8: *This Figure illustrates a how a tilable texture is seamless, when placed in a tilable fashion.*

Figure 2.9: *This Figure depicts three examples of solid textures created from 2D texture samples [32]. Each 2D sample is shown next to the associated solid texture model. Left: A statue model that appears to be made of wood. Middle: A cube model, where dark brown sections of the texture are rendered transparent, revealing internal structure. Right: A bunny model that has a section sliced off to show it's interior.*

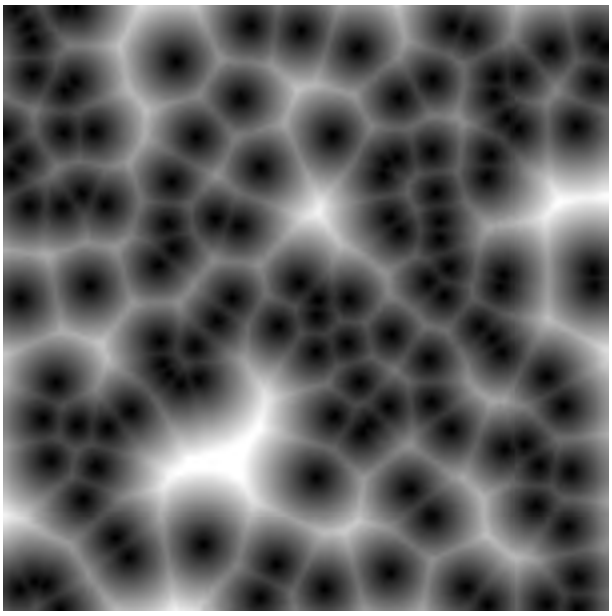
Solid textures are capable of adding far greater detail to 3D models than 2D textures. Where 2D textures are limited to approximating fine surface detail, such as colour and geometric structure, such as normal maps [4], volumetric textures are inherently capable of defining additional internal detail to the 3D model, at the cost of greater memory utilisation. 3D textures can usually be attached to 3D models through a trivial parametrisation. By contrast, mapping a 2D texture to a 3D surface is a complex process with many techniques to choose from [18]. Even the best parametrisation schemes introduce distortion, and the effects on the appearance of a surface models can be very jarring.

2.4.1 Procedural methods

Procedural methods are used extensively to automatically generate 2D and 3D textures. While procedural methods typically focus on generating 2D textures, most are capable of being generalised to 3D texture generation [49]. These methods can potentially provide a number of advantages [14]. By storing the generation procedure instead of the final texture, textures can be stored in a much more compact way. This can reduce storage size from megabytes for the texture data, to kilobytes for the procedure. The produced textures are also scalable, with no fixed boundary or resolution sizes. The methods also offer variation in results, but can suffer at the need for skilled programmers to create or modify them. A common basis for many procedural methods is through the use of noise functions.



(a) *This Figure shows marble textured model of a vase, created by Perlin noise [50].*



(b) *This Figure depicts a representation of white noise created by Worley's noise function* [11].

(c) *This figure shows a textile cone snail* [12]. *The snail's shell contains a surface pattern similar to Rule 30 introduced by Wolfram et al. [46]*

Figure 2.10: *A collection of figures featuring natural and generated texture patterns.*

Procedural Noise functions

Noise functions are one of the most common and successful procedural techniques, often fundamental to other complex techniques [33]. The functions are simple, compact and efficient ways of creating natural looking textures through structured randomisation. For example, Figure 2.10a shows an example of Perlin Noise [50], used to create a marble texture effect. Perlin noise is a function used to synthesise white noise, which returns pseudo-random values between -1 and 1. These values are derived by interpolating random vectors defined on the vertices of an integer lattice. Another method of generating noise is through a cellular noise function, introduced by Worley [66]. This noise function is based on the euclidean distances between a system of randomly placed particles. The values produced are based on the distance of the second closest particle to a selected point. This procedure results in creating noise with a cell-like character, shown in Figure 2.10b.

2.5 Cellular automata

Cellular automata (CA) are mathematical models of systems composed of many simple components working together to produce complex behaviour [46]. CA have been used extensively in procedural generation, notably for the generation of complex models, physical simulations and texturing.

To apply or "run" a CA, one iteratively applies a set of rules to a discrete representation. Applying CA to a regular grid of cells, such as a grid of pixels in an image, can produce intricate patterns after several iterations. At each iteration the rules change the state of a cell according to its own state, as well as its neighbouring cells states. Each of these grid cells in has a finite number of states. The states can vary from simple Boolean values (0/1 or on/off) to a complex set of states, such as a range of colours and transparency levels. CA can also be applied in parallel at each iteration, this is due to the cells state only relying on states from previous iterations.

2.5.1 Texturing cellular automata

Cellular automata are often used to create complex 2D textures. For example, the method proposed by Wolfram et al. [46] creates surprisingly complex structures and patterns on a 2D surface, through the use of simple rule sets, such as their Rule 30 and Rule 110. These rules are limited to simple manipulation of on or off states but can create convincing renditions of patterns subsequently observed in nature, as seen in Figure 2.10c.

3D cellular automata are similarly used to create solid textures. An example of this is shown by Buchanan et al. [8], who demonstrate a 3D CA that can create volumetric wood textures. This approach is capable of adapting to local phenomena such as knots, by simulating an approximation to the biological model.

While CA can be used to generate full 3D textures, many techniques choose to focus on surface detail. One method introduced by Gobron and Chiba [20], simulates crack patterns using surface cellular automata.

[12]Rocchini, *Worley noise: example of Worley noise with a euclidean distance function*, `http://www.webcitation.org/6UB75k20g`, [Online; accessed 17-October-2014]

[12]Photographer: Richard Ling, *A Textile cone snail (Conus textile)*, `http://www.webcitation.org/6TdMvPzaW`, [Online; accessed 10-September-2014]

2.5.2 Simulation cellular automata

In addition to creating textures, CA are able to simulate physical process which can create complex structures. Arata et al. [3] use CA to model free-form shapes, capturing physical deformation in clay models. This provides a realistic model of clay deformation by modelling individual clay elements as voxels physically interacting with each other. CA are also used in a similar method by Tarakanov and Adamatzky [62], which physically simulates clothing. This simulation is achieved by modelling the clothing as a 2D mesh of particles, each particle connected to four other particles in a regular fashion. The particles are placed in an initial state and position, which is then altered into different states during iteration. The position is updated according to these states to form the simulation. These states include, rigid displacement, swinging and free-fall.

Cellular automata are also used to simulate plant growth. Greene et al. [22] use voxel-space automata which are capable of sensing their surrounding environment. This allows the CA to avoid barriers and to model heliotropism (simulated through ray-casting).

2.5.3 Efficient and real-time cellular automata

Cellular automata can offer great benefits for procedural generation due to their representational versatility. However, they can be difficult to implement efficiently in certain situations. This is a direct consequence of the way in which cellular automata work: the rules have to be be applied to every cell in the grid, which is particularly onerous for 3D grids. Ferrando et al. [17] try to mitigate this cost through the use of parallel hardware. First they use a Octree data-structure to divide up the data-set according to spatial locality, similar to the approaches seen in voxel grids 2.3. Then when calculating the rules over the grid, they exploit the parallelism from both multi-core central processing units (CPUs) and many core Graphics Processing Units (GPUs).

Accelerated cellular automata are also capable of real-time simulations, typically within video games. The most common use of CA in games is to simulate environmental effects, such as, fluids and precipitation, explosions, air and fire [61]. Judice et al. [28], propose a real-time fluid modelling and animation framework using Lattice Gas Cellular Automata (LGCA). This framework simulates molecular dynamics through simple rules that move particles across a lattice in two stages. In the first stage rainfall is simulated across a 3D hexagonal lattice in a cubic volume on the CPU. Once the rain drops fall far enough to intersect the terrain height-map, a separate ground water simulation occurs on the GPU.

Another application of CA to create structures is by Johnson et al. [26], who procedurally generate level maps for games. This technique creates level designs in real-time, which show how to create compelling tunnel based maps in an infinite cave game.

2.5.4 Creating cellular automata

Although cellular automata are capable of generating interesting and complex behaviour, finding rules that create the desired outcomes is a non-trivial task [39]. Creating CA to generate a particular output by hand is also laborious. This is, in part, due to the large number of rule sets that have to be devised. Especially, when the desired outcome requires global structure to be maintained across great distances.

Machine learning and cellular automata

One way of creating cellular automata is through the use of machine learning [39]. By using machine learning techniques, CA can be evolved or discovered that are capable of retaining local and global behaviours. For

example, Andre et al. [2] use genetic programming with automatically defined functions to evolve CA. This is achieved by evolving rule sets that closely match specific fitness functions. As an example, they find specific global criteria with binary 2D CA. In a similar vein, Chavoya et al. [10] expand on this by using genetic algorithms to grow 3D CA that are capable of forming into desired shapes. In a further study, Breukelaar et al. [7] show that it can be easier to evolve CA with genetic programming, by using multi-dimensional CA. Interestingly, this result implies that multi-dimensional CA can solve problems faster than one-dimensional CA.

Cellular automata grammars

Another way to create CA is through the use of grammars. Grammars can speed up the creation process significantly by providing a more intuitive interface. This is the approach we have adopted. One way of aiding this rule creation process is to allow the user to specify the desired behaviour through the use of more intuitive grammars.

Stauffer et al. [59] prove that it is possible to find a mapping from L-systems to cellular automata. In their research they find that L-systems can specify self-replicating structures. These self-replicating L-systems can then be represented by equivalent cellular automata in a graphical interpretation. As L-systems are naturally suited to growth modelling, the resulting cellular automata prove a good fit for growth. Alfonseca et al. [1] introduce an inverted approach to Stauffer et al. by creating L-systems from n-dimensional automata. In the study they find equivalent L-systems to 1, 2 and 3 dimensional CA. While both of these approaches show that it is possible to find a mapping between CA and L-systems, they do not explore any automatic or general conversion methods.

An alternative approach is to use shape grammars to derive cellular automata (see Section 2.2.2 for an overview on shape grammars). Spellar et al. [58] show that it is possible to create cellular automata by transcribing shape grammars. This enables a designer to combine the shape grammars ease of intuitive visual creation with the ease of transcription for computation of cellular automata.

2.6 Voxel-space shape grammars

Previous work at the University of Cape Town [13], investigated voxel-space shape grammars as an alternative to conventional mesh-based approaches. A voxel-based approach has two main advantages over mesh-based approaches. First, is the reduction in difficulty of Boolean geometry operations on shapes. Boolean geometry operations allow the user to create composite shapes from two or more shapes [29]. Mesh-based Boolean geometry operations require complex reconstruction techniques, such as stitching mechanisms. In contrast, voxel-based approaches only require simple blending approaches on intersecting voxels. Second, surface detailing on stitched areas or sub-meshes can be difficult to achieve in comparison to a voxel-based approach. This is because it requires careful alignment of each meshes UV-maps to avoid seams and distortion.

2.6.1 Surface detail with 2D cellular automata

In this prior work 2D cellular automata are used to create surface detail in voxel-based models. The 2D CA are limited to surface alterations, via detail tags, and do not modify the active state of the voxels. The cellular automata are manually programmed in python, and do not interact with the shape-grammar system. This has similarities to pixel-shaders present in polygon-based rasterisation libraries. Two examples of such 2D CA are shown in Figure 2.11 In contrast, the system presented in this work both modifies detail and the active state of voxels. This is similar to geometry-shaders in the rasterisation pipeline. .

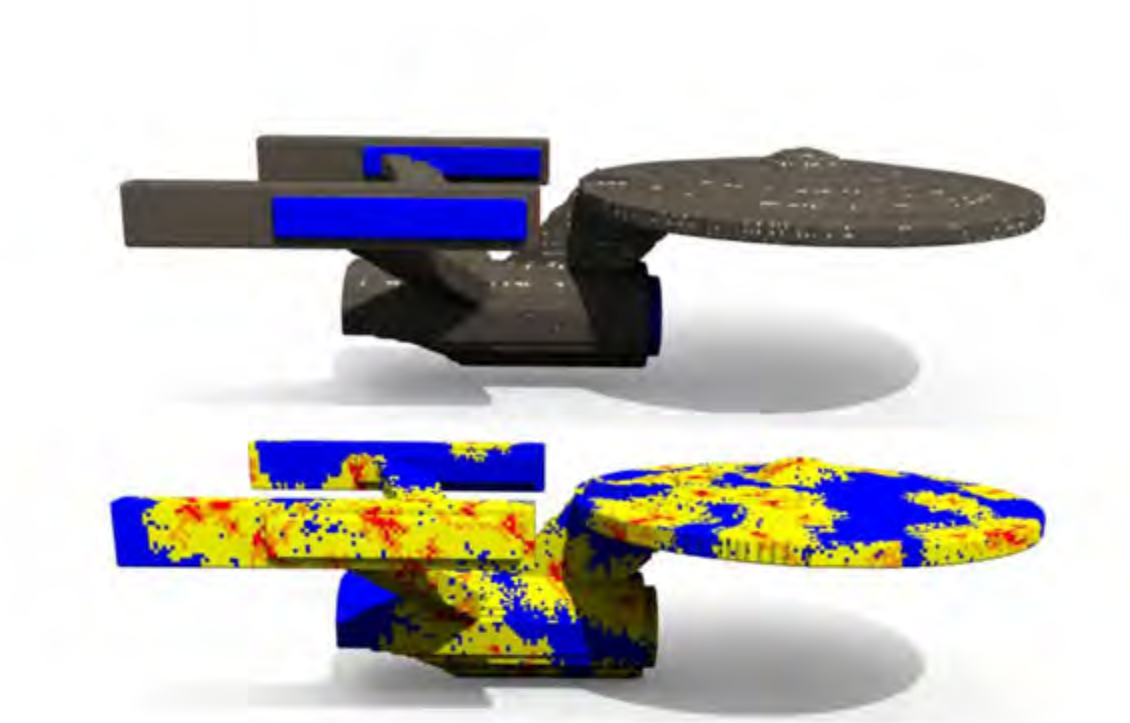


Figure 2.11: *This figure shows two different detailing rule sets applied to a model of the Starship Enterprise. The top image shows a window placement CA, while the bottom shows a random "plasma"-like effect.*

The capability of the CA are critically dependant on the operations and capabilities available to the detail rules. This dependency process can become cumbersome to manage and parse. Additionally, managing ambiguity between rules and rule priority can be problematic. Global parameters, such as the number of detailing passes need to be included with the rule set, which can reduce the overall flexibility of the system. This is alleviated somewhat by the addition of rule-set extensions.

Apart from the difficulties noted above, the 2D CA method could be extended in a number of ways: First, one can include active state modifications, which would allow 3D erosion and structure creation. Second, 3D shape recognition and 3D movement could allow for a smarter, or more context aware CA. Finally, the introduction of probability and parameter variance could help introduce more variety and realism to the resulting detail.

As a proof of concept, the system in this work includes all the functionality listed above. However, some functionality is limited and can be extended in future work.

2.7 Summary

In this chapter we examined the background information relevant to this research. The background includes an introduction to procedural generation and, reviews generation tools and real-time procedural generation. We also discussed the use of procedural grammars, voxel grids and cellular automata in procedural generation. We use many of the techniques described above in our own work. For example, procedural grammars to generate CA, voxel data-structures for efficient storage and procedural generation using voxel-based CA.

In the next chapter we introduce the voxel engine, which serves as the base development platform for this research.

Chapter 3

Voxel Engine

The core goal of this thesis, is to design and implement grammars that produce three-dimensional cellular automata (CA) for the purpose of generating surface detail on 3D models. To test the grammars, a system needs to be created that can apply the CA produced. This makes it vital to create an engine that can construct and manage the voxel-data use by the CA. As described in Section 2.3, voxel grids are capable of describing complex shapes and colourful 3D models. However, managing voxel grids efficiently can be difficult due to their exhaustive enumerations. To be more efficient, data-structures are carefully used to accelerate and manage the voxel data.

In this chapter we discuss the design of the voxel engine. This includes how voxel grids are imported, manipulated and exported. Section 3.1 gives a brief overview of the engine and it's subsections. This is followed by Section 3.2, which provides motivation for the programming libraries we use in the engine. This is followed by a deeper look at the import and export abilities of the engine in Section 3.3 and Section 3.4, respectively.

3.1 Engine overview

The voxel engine design has four important sections. These sections are the; model converter, voxel data-structure, cellular automata executor and cellular automata interpreter. A visual representation of the components is shown in Figure 3.1. Each section is supported by programming libraries, which are discussed in Sections 3.2 and 4.4.1.

The most critical section of the engine is the CA executor. The executor is crucial in testing the CA produced by the grammar. The CA are tested by executing them over different voxel grids. Additionally, the CA interpreter connects the parsed CA grammars to the CA executor. This is achieved by converting XML representations of the grammars into executable programming functions. Chapter 5 further explains these processes in detail.

The third section is the voxel data-structure. This contains and manages voxel-data created by the model converter. As described in Section 2.3.1, there are different approaches to managing voxels. In Sections 3.2 we discuss how voxels are managed in the engine, specifically looking at the OpenVDB library.

The final section is the model converter, which transforms polygon mesh based models into voxel based models. For real-time rendering and fast processing of 3D data, polygon mesh models are primarily used in industry. However, as discussed in Section 2.5, it is simpler to apply CA to voxel grids. This makes the capability of converting to and from polygon or voxel model representations essential. In the engine design this importing/exporting process is bootstrapped by features of the OpenVDB library (see Section 3.2). In

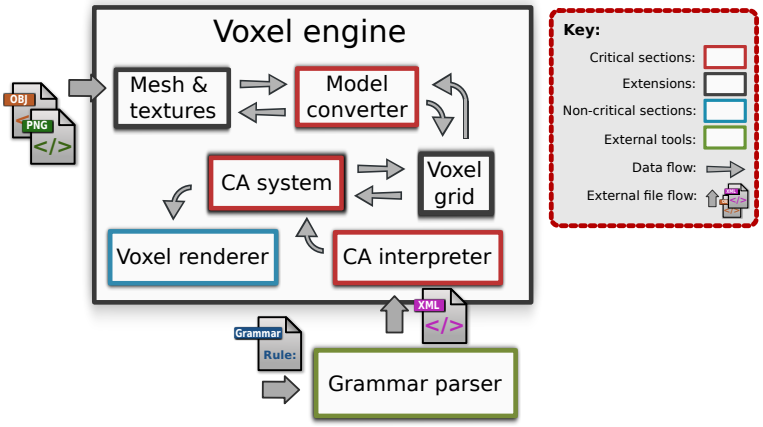


Figure 3.1: *This diagram shows the component structure of the voxel engine. The three main components in the engine are the cellular automata (CA system, CA interpreter and model converter. These components allow the creation and management of the CA as they interact over the voxel grid. The model converter converts polygon-based models into a voxel-based representation. The voxel grid contains the voxel data, which is modified by the cellular automata. The CA interpreter converts the parsed grammar file (XML) and creates an executable CA. Each CA is controlled by the CA system, which facilitates the voxel grid modification. Finally the voxel renderer displays the voxel grid, which helps when debugging the CA.*

the context of the engine, importing refers to converting from polygon to voxel representations. Exporting refers the the reverse process. Sections 3.3 and 3.4 detail our extensions to the OpenVDB import/export features, respectively.

Additionally, section 3.5 describes the voxel debug render. This is a less critical component of the engine, with the primary focus on debugging the CA.

3.1.1 Engine work-flow

The first step of the work-flow is for a user to import polygon-based models into the engine. A model is then chosen to be converted by the engine into a voxel-based form. The model is converted using the OpenVDB library with extensions to transform the texture map information to voxel colours (see Section 3.3 for more detail). A diagram of the interacting sections of the voxel engine is shown in Figure 3.1.

In parallel to model importing, cellular automata grammars can be imported by the engine. This process is an automated process, with multiple stages transparent to the user. First the engine executes the Bison and Flex parser to parse a chosen grammar file (see Section 4.4). Next, an XML file resulting from the parsing process is imported directly by the engine. During the import stage, a programming function tree is built from the XML form. The function tree contains the rules of the CA and is ready to be executed over a voxel model.

With both a voxel-based model and an executable CA loaded into the engine, the CA can now be executed over the voxel grid. CA can be executed once or over many iterations. Different CA can also be run sequentially to one another, enabling complex functionality from interacting CA.

In a final step, the modified voxel models can be exported. Each voxel model is converted back into a polygon form using OpenVDB, with extensions to create texture maps from voxel colours (see Section 3.4).

3.2 Voxel storage and library motivation

The main objective of this thesis is creating cellular automata grammars. To focus on the main objective, it is important to reduce the development time of tangential systems. However, reducing time can be a difficult (or impossible) task when considering the development of integral systems. One of the integral systems in this research is the voxel engine. The goal of the engine is to make voxel manipulation by the CA as seamless as possible. As the CA are so dependant on the engine, it is important to find techniques or libraries that satisfy the requirements. This also needs to be achieved simply, while not detracting from the focus of the research.

Data structures suggested in Section 2.3.1, help to mitigate the effects of storing and accessing voxels in large grids. However, implementing these data-structures requires careful design and attention to detail. Without careful design the contained data could be bloated, volatile and slow to access. To help mitigate the risk of designing and implementing our own data-structure, we decided to look for libraries to boot-strap this process.

We found that while voxel data is ubiquitous in computer graphics, few voxel-based libraries are available. This is mainly due to the use of proprietary software created by development studios. Typically this software is not distributed outside the studio, or only severely limited versions are released. One exception is the recently released library, OpenVDB[1]. OpenVDB is an open-source library for time-varying volumetric data discretized on three-dimensional grids. The foundation of OpenVDB is a novel sparse-volume data-structure [42]. The *"open"* in OpenVDB, describes the open nature of its development, with a strong focus on collaboration. Additionally the *"vdb"* describes the data structure they use, a Volumetric, Dynamic grid similar to a B+ tree (VDB).

Not only does the library provide a state-of-the-art data-structure, it also contains many voxel-based tools. Two of the most important tools are the voxel-to-mesh and mesh-to-volume functions. These are tools that can automatically convert between polygon mesh and voxel model representations, and vice versa. The only component we require that these tools do not cover is retaining and recreating colour information provided. The texture maps in mesh-based models are therefore lost when using the standard tools.

Covering so many features of our requirements in a single library, OpenVDB proves to be a good solution for our research. The following two sections detail the model conversion tools of OpenVDB and the extensions we made for them.

3.3 Mesh to volume

As discussed in Section 2.3.2, many techniques are available to facilitate the conversion from polygon-mesh models to voxel models. In this implementation we use OpenVDB's MeshToVolume tool. While this tool can create a coherent voxel representation, colour information stored in texture maps are not used in the voxelisation process. To address this, we extend the tool to colour each voxel within the grid. The following subsections describe OpenVDB's volume to mesh tool and the voxel colouring extension, respectively.

3.3.1 OpenVdb's mesh to volume implementation

OpenVDB's mesh to volume conversion converts polygon models (quadrilaterals and triangles), into a level set (signed distance field) or unsigned distance field [42]. This is achieved through a scan conversion technique. The implementation requires non-manifold surfaces, as well as requiring a surface model to completely enclose a volume (the model is water-tight).

[1] DreamWorks Animation. *About OpenVDB*, `http://www.webcitation.org/6WmNfiGZk`, [Online; accessed 04-March-2015]

This scan conversion technique is able to convert into a sparse data structure, by partitioning the mesh into non-overlapping tiles. These sub-meshes are constructed such that each tile contains polygons from the original mesh that lie within a distance of $B/2$. Here, B is chosen as the width of the narrow-band. The sub-meshes are then scan converted directly into the LeafNodes of the OpenVDB sparse-octree.

3.3.2 Voxel colouring

To retain and use colour information stored in a polygon-meshes texture-map, we assign colour values to each voxel. This process is commonly refereed to as "voxel colouring". Once a mesh is created using OpenVDB's MeshToVolume method, we traverse through the voxel grid to assign each voxel a colour. This is achieved by finding the closest pixel colour in the texture map according to UV coordinates. The UV coordinates are found by interpolating between the vertices of the closest triangle in the polygon mesh.

First, we need to determine the closest polygon to each individual voxel. Conveniently, OpenVDB's MeshToVolume process stores the closest polygon in the original mesh to each voxel. This is determined when the voxel grid is initially created. To simplify interpolation we also convert all the produced polygons into triangles.

Using the closest triangle, we determine the closest point on the triangles plane to the voxel's position. This will allow us to determine the best UV coordinates to describe the position. This is achieved during the UV interpolation process, described below in section 3.3.2.

UV coordinate interpolation

UV coordinates are used to map points within a polygon to pixel positions on a texture map. The UV coordinates for points within the polygon will need to be interpolated from the vertex UV's. UV coordinates map a value between 1 and 0 for the two axes of the texture, the U and V axis. We limit our scope to the common 2D image, however UVW coordinates could map to volumetric textures. To save memory, the UV coordinates are only stored in the vertices of the polygon.

Before the interpolation can be calculated, the closest point on the triangle to the voxel's position must be found. To find it, the point is projected onto the surface of the triangle's plane. As depicted in Figure 3.2, the the voxel's position (P1) is projected onto the triangles plane. This is found by moving the point along a line, n, orthogonal to the triangle's plane, which also passes through point P1. The point of intersection between the plane and the line (P2), is calculated using Equation 3.1. This equation calculates the distance D, from P1 along line n to the intersection point P2 (also known as the shortest distance from P1 to the plane). In the equation, $\hat{\mathbf{n}}$ is the unit vector normal of the plane, $\mathbf{x_0}$is the voxel position P1 and $\mathbf{x_i}$ is one of the vertices V1-V3.

$$D = \hat{\mathbf{n}} \cdot (\mathbf{x_0} - \mathbf{x_i}) \tag{3.1}$$

To find the UV coordinate of the projected point, we use barycentric coordinates. The barycentric coordinates are used to help interpolate between the UV values at the polygon's vertices (V1-V3 in Figure 3.2). In Equation 3.2, the point $\mathbf{P}$ is made up from a ratio of three components, $a_i\mathbf{P_i}$. Here $\mathbf{P_i}$ are the three vertices of the triangle, and the coefficients are defined so that $a_0 + a_1 + a_2 = 1$. If any of the coefficients are $0 > a_i > 1$, then the point lies outside the triangle. Similarly, if $0 = a_i$ then the point lies on the side of a triangle, and if $1 = a_i$ the point lies on vertex $\mathbf{P_i}$.

$$\mathbf{P} = a_0\mathbf{P_0} + a_1\mathbf{P_1} + a_2\mathbf{P_2} \tag{3.2}$$

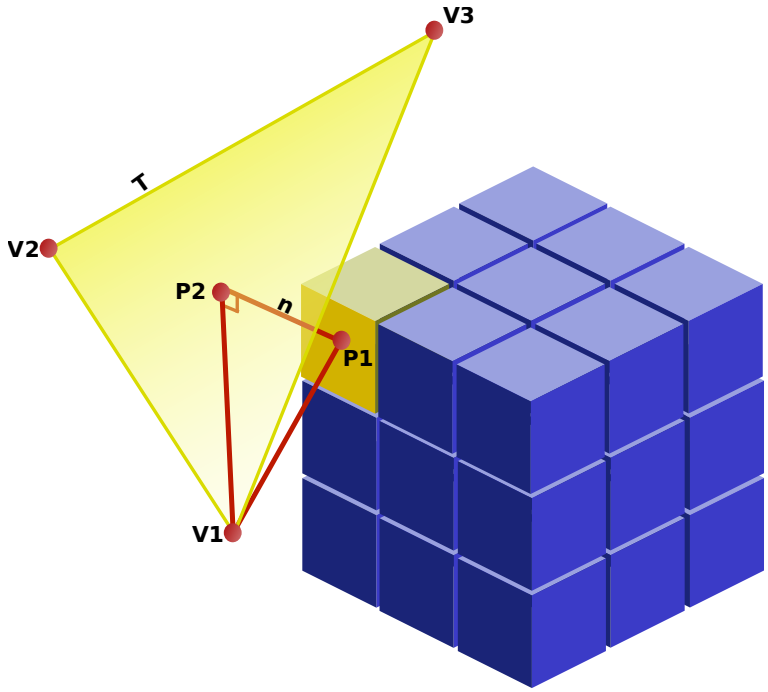


Figure 3.2: *This figure depicts the projection of an arbitrary point P1 onto the triangle T. The point is moved along line n, which is orthogonal to T and passes through P1. The distance to move from P1 to the intersection point P2 (the projected point), is calculated using Equation 3.1.*

The a_i cooefficients are found by the magnitude of the vector from each vertex to the point. For example, a_o is the magnitude of the vector from point $\mathbf{P}$ to point $\mathbf{P_1}$. These coefficients ratios are homogenous, and can therefore be used to solve Equation 3.3.

$$\mathbf{UV} = a_0\mathbf{UV_0} + a_1\mathbf{UV_1} + a_2\mathbf{UV_2} \tag{3.3}$$

In the rare case where $0 > a_i > 1$, and the point lies outside of the triangle, the point needs to be constrained to the triangles area. This is achieved by projecting the point onto one of the triangles edges. This process is similar to the plane projection, where the edge normal is orthogonal to triangles normal. Once projected onto the edge the barycentric equations can be solved.

3.4 Volume to mesh

Once the cellular automata have modified a grid, it is useful to convert from the voxel representation back into a polygon mesh. Traditional tools and pipelines for work on meshes, as well as fast rendering speeds make the conversion a practical choice. To facilitate the conversion OpenVDB has a simple tool called VolumeToMesh. Similar to the MeshToVolume tool 3.3, the colour information is not stored during this transformation. We therefore extend the tool by using a simple texture generation technique to store the colour information of the output mesh.

In the following subsections, we discuss OpenVDB's VolumeToMesh implementation and the texture generation extension we use, respectively.

3.4.1 OpenVdb's volume to mesh implementation

OpenVDB use a modification of a technique called dual marching cubes, which is an adaptive mesh reconstruction technique. The original dual marching cubes method is an extension of both the marching cubes method by Lourensen and Cline [37], and the dual contouring method by Losasso et. al. [27]. The advantages if of the dual marching cubes method, is the ability to retain sharp edges, sparser tessellation than previous methods and is adaptable to octree-based data-structures without the need to patch tiles together.

Similar to the dual contouring method, the dual marching cubes method creates a mesh that is topologically dual to the mesh created by marching cubes (or similar primal grid methods). In a dual mehs, each vertex in the DMC or DC methods corresponds to a face in the MC mesh and vice versa. DMC differs from DC by aligning the created vertex according to the signed distance field (also known as the implicit value $f(x, y, z)$) and not the features of the surface. This is achieved by not only storing the position of the created vertex, but the estimated value w of the signed distance at the point.

To determine the vertex position that approximates the feature in a cell c, they use a quadratic error function QEF(similar to the function proposed by Garland and Heckbert [19]). To generate the QEFs, tangent planes to $f(x, y, z)$ are computed over grid of points sampled within c. The QEF is defined by Equation 3.4, where $T_i(x, y, z)$ is the tangent (see Equation 3.5) and $\nabla f(x_i, y_i, z_i)$ is the gradient at sample point i.

$$E(w, x, y, z) = \sum_i \frac{(w - T_i(x, y, z))^2}{1 + |\nabla f(x_i, y_i, z_i)|^2} \tag{3.4}$$

$$T_i(x, y, z) = \nabla f(x_i, y_i, z_i) \cdot ((x, y, z) - (x_i, y_i, z_i)) \tag{3.5}$$

3.4.2 Reconstructed mesh parametrization and texturing

To create a texture for the mesh reconstructed by OpenVDB's VolumeToMesh implementation, we essentially reverse the process seen in Section 3.3.2. First, we parametrize the mesh by creating a new texture and assigning UV coordinates to the new vertices. Once a texture and UVs are present, we can colour the texture according to the voxels closest to each pixels world space coordinates.

Mesh parametrization

The method we chose for parametrization is a simple method, limited by scope, which is capable of relatively realistic texture rendering. However, it is quite naive and should be replaced by a more complete parametrization technique. The method maps each polygon to a triangle in UV space, which are packed in a compactly and uniformally.

As a first step in the approach we simplify each polygon into triangle sets. Next each triangle's vertex UV's are mapped to triangles in uniform triangle strip rows across the texture. Due to the relatively uniform nature of the reconstructed meshes polygons, and the size of each polygon to voxel size (almost completely in a 1-1 ratio), we can achieve relatively realistic textures from single colour triangles. This approach is however flawed for the larger polygons created from the adaptive mesh reconstruction techniques (which we found to be very rare in high resolution grids).

This approach could be replaced by any mesh parametrization technique (also known as UV unwrapping). A smarter choice, such as the technique by Lvy et al. [35], would assign UV areas in the texture which retain the relative shape and size of the original polygons. These techniques create texture atlases, which better utilize the texture space by providing higher pixel detail in large polygons and lower fidelity for small

polygons. The atlases are also easier to modify by a texture artist by packing the polygons according to their original distance and orientation to each other.

Pixel colouring from voxel colours

To find the best colours for each pixel in the new texture, the closest voxel to each pixel's world position is found. The world position is once again found by using the barycentric coordinate Equations 3.2 and 3.3. For each triangle, every pixel within its vertex UVs are found through a scan-line process. Each pixel's world space point is found by solving Equation 3.3, and then using the homogeneous coefficients a_i in Equation 3.2.

The closest active voxel is then found through a choice of two methods, a data-structure search and a grid ray-cast. The first search method is a closest voxel search accelerated by using the sparse-octree data-structure. This method is faster than the ray-casting method, however it occasionally introduces aliasing issues. Aliasing occurs around the polygon, where subsurface voxels are preferred over voxels lying above the polygon surface. This introduces artefacts with CA that only modify surface level voxels. The second is an accelerated ray-casting method using OpenVDB's digital differential analyser (DDA). Each ray-cast is created using the normal for each polygon. From a point 4-voxel units in the normal direction from the world position, a ray is cast in the direction opposite to the normal. This will find the closest surface coordinate, while introducing "fuzzy" edge artefacts from choosing polygons that are not necessarily the closest and increased computation. The artefacts are only noticeable at low resolutions, so this method is preferred for accuracy.

The colour of the closest voxel is used to colour the pixel. As an alternative a trilinear interpolation of the colours from closest voxel and its neighbours may be used, at the cost of further computation.

3.5 Voxel rendering

Once a voxel model is created it is important to be able to view the grid. This is useful when debugging running CA, to see how their rules interact and how they behave unexpectedly. A typical approach is to render the grid using standard graphics API's, such as OpenGL[2] or Direct3D[3].

This enables the use of traditional, and fast, raster rendering techniques. However, rendering voxels through rasterisation can be unintuitive to set up and difficult to optimise. As rendering is not the focus of the research, we sought to implement a simple technique that would satisfy our rendering performance needs. As our needs are to only debug cellular automata, we typically only need to render low resolution voxel grids. With small grid sizes, a naive rendering approach can be sufficient and later extended or replaced.

The rendering technique we choose is a simple geometry instancing based technique. In this technique multiple copies of the same mesh are rendered in a scene at once. While each mesh instance has the same geometry information, they may each have different properties, such as, colour or animation pose. For our purposes, we instance a simple cube mesh with a low polygon count. Each cube has it's own colour and world position information.

Before each frame is rendered, the position and colour of each instance is sent as packed textures to the GPU, with a single copy of the cube mesh. This is achieved using OpenGL texture buffer objects (TBOs). Each instance can then access colour and position information based on it's instance ID. The ID's are are provided by the graphics API, which facilitate indexing memory on the GPU.

[2]Khronos Group. *OpenGL, The Industry's Foundation for High Performance Graphics*, `http://www.webcitation.org/6X4PXoQfQ`, [Online; accessed 16-March-2015]

[3]Microsoft. *Getting Started with Direct3D*, `http://www.webcitation.org/6X4QLQBsZ`, [Online; accessed 16-March-2015]

Together all these properties create simple voxel rendering, which is realistic and fast enough for debugging purposes.

3.6 Summary

In this chapter we introduce the voxel engine. The voxel engine acts as both the user interface for the system, as well as the core voxel management system. The foundation of the voxel engine is built upon the OpenVDB library to speed up development time. In addition to the tools provided by OpenVDB, a number of extensions are introduced.

The first of these extensions is the voxel colouring mechanism extending OpenVDB's mesh to volume tool. This extends the model type conversion tool by setting the voxels colour and tangent-space unit vectors from the polygon-meshes texture maps. This is achieved through coordinate projections and barycentric coordinates. Similarly, we extend the volume to mesh tool to create texture maps from the voxel colouring. The texture maps are coloured by ray casting into the volume from the approximate world position of each pixel coordinate. The voxel engine also renders the voxel grid through an instance-based rasterisation method, which allows users to easily debug the CA they create.

In the next chapter we look at the cellular automata grammar, which we use to generate 3D surface detail.

Chapter 4

Cellular automata grammar

Creating cellular automata (CA) can be a conceptually difficult and laborious task. This is especially difficult to conceptualise when using more than two dimensions. In this work we focus on generating surface detail on 3D models. This influences the design of our grammar to make the process of programming 3D CA easier.

In this chapter we discuss the motivation behind our grammar 4.3. This is followed by a section on the grammars design 4.3, which includes a break down of the grammmar's rule structure and syntax. Next, we categorize the grammar according to similar grammars and grammar theory 4.2. Finally, we discuss the implementation on the parser for our grammar 4.4, including the tools used and the abstract syntax tree produced.

4.1 Grammar motivation

There are many ways of simplifying the process of creating CA, as introduced in section 2.5. Some of these methods include; manual creation, traditional programming languages, visual interfaces, grammars and machine learning techniques. Our focus on grammar creation is motivated by our target audience. By using grammars we aim to provide technical artists with a simple written tool, which emphasises rule creation over programming. While a visual interface could be written to produce or bypass the grammar, this is out of scope for this project and we leave this for future work.

The grammar should:

- Simplify the creation of 3D CA.
- Provide a less verbose interface than traditional programming languages, without losing flexibility.
- Provide a tool-kit to create complex CA, capable of contextual awareness.
- Reduce compilation times.

These goals are important to increase productivity when prototyping grammar rules. We will briefly describe the main features we choose to combat the above requirements.

First is the concept of evaluation and modification functions. These are two distinct function types, which often correspond to one another. "Evaluators" analyse the attributes of voxels and their neighbourhoods, returning a result if the function finds a match or not. "Modifiers" change the attributes of voxels when evaluators are found. Separating symbols into these two categories makes the rule structure more coherent,

with the additional benefit of being able to overload function names depending on the section context. An example of two corresponding functions would be the colour evaluator and modifier, both of which have the similar look and syntax, $colour(r, g, b)$. This simplifies function names, which improves grammar readability and usage.

Next is neighbourhood navigation. This tool follows symbol-based movement paradigms, similar to those introduce in L-systems [53]. We extend the symbol movement in two ways. First is by allowing 3D movement through the standard x, y and z symbols, while indicating direction with the + or minus − symbols. These can be used together to represent a grid movement of $< 1, -1, 1 >$, as $+x - y + z$. Second, we add the notion of scope similar to curly-bracer scope in programming languages such as C++ or Java. This is an optional way of expressing navigation with more clarity than a string of movement symbols.

Finally, we include functions and global variables. The variables range from predefined colours to voxel states. Global variables work in similar ways to those in programming languages such as C++ and Python. Global variables are useful when information needs to be shared between grammars. For example, states that are declared may be used by each rule in the grammar or across grammars.

As well as the global goals mentioned, each of the sections and symbols within the grammar have important design decisions. Later in this chapter, these decisions are described in the introductions to the section-s/symbols.

4.2 Grammar categorization

The grammar we outline below takes inspiration from components of many other grammars. In particular we borrow from three types of formal grammars; L-systems, shape grammars and 2D cellular automata grammars. To determine which category of the formal grammar hierarchy our grammar lies within, we compare the grammar categorization to other grammars. This includes a discussion of the similarities of our grammar to others, and finally a brief discussion of the way in which 3D CA grammar relates to the grammar hierarchy.

4.2.1 3D CA grammar

The grammar presented in this work (a 3D CA grammar) has many similarities to L-systems. For example, productions in this grammar are also applied in parallel. The grammar also rewrites symbols, which in the context of this work represents the rewriting of voxels.

The simplest form of the grammar can be context-free, with deterministic and non-deterministic (stochastic) extensions. This is very similar to OL-systems, and can likewise be categorised in the Chomsky hierarchy. Additionally, the grammar can be context-sensitive, which once again fits into the Chomsky hierarchy similar to IL-systems.

The 3D CA grammar rewrites voxels in a similar way to shape grammars, where a voxel (or voxel neighbourhood) with certain parameters can be matched and rewritten. However, in the 3D CA grammar a matched voxel or neighbourhood can cause any surrounding voxel to be rewritten. This is analogous to a matched symbol rewriting symbols that are not necessarily within the context of the matched symbol. This neighbour rewriting technique is unique to the 3D CA grammar, which to our knowledge is not present in other procedural grammars.

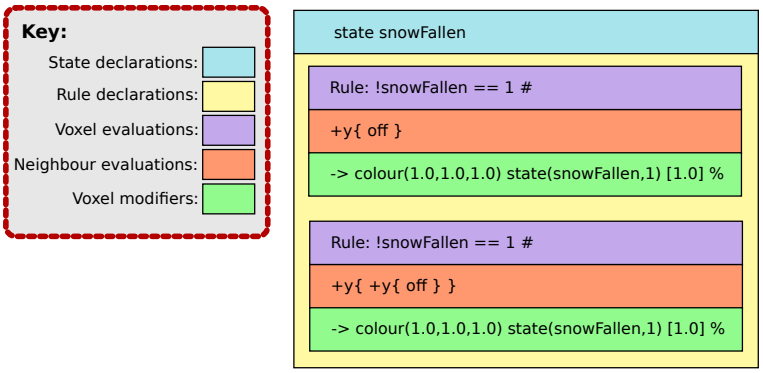


Figure 4.1: *This diagram depicts the main sections of the grammar. The first section contains the state and variable declarations. The second holds the grammars rule list. Each rule is further seperated into three sections. The first of these, contains parameter evaluators targeting only the index voxel. The third section shows evaluators within the index voxels local neighbourhood. Finally the last section consists of modifiers, which will change the parameters of the voxel.*

4.3 Grammar specification

The structure of our grammar is split into two sections, the CA sections and the section syntax. The section structure is the overall description of large containing sections, while the section syntax describes the token structure within each sections.

The following subsections describes the syntax as follows. First section 4.3.1 describes the breakdown of the CA into rule sections and their subsections. Section 4.3.2 goes a level deeper, describing the individual evaluation nodes within the rules. This is followed by a look at section 4.3.3, which shows how neighbourhood navigation nodes affect evaluation nodes. Finally, section 4.3.5, describes the voxel modifier nodes in the rules and how they set voxel attributes.

4.3.1 Rule structure

The structure of the rule set is defined in two sections. The first section holds user-defined variables, such as parameters and states. The second section is where each rule is defined.

The user-defined variables are available to the entire rule set of a grammar. For ease of use, the definitions are similar to member variable declarations in object oriented programming. Defining the variables in the beginning of the grammar makes user definitions easier to parse and clearly states what variables are used to readers.

Each of the rules are applied to every voxel in the grid during an iteration. For ease of use the rules are split into separate sections, evaluators and modifiers. The sectioning is designed to allow symbol overloading between evaluators and modifiers, while maintaining clarity which symbol is a modifier or evaluator. Without the sectioning, different symbol names would be required for each type, which would make the grammar more cumbersome to write.

The evaluator and modifier symbols are parameter-based, which mimic the structure of programming language functions. The evaluators determine whether the voxel is a match to the rule, or not. The evaluators find a match, by checking the variables of the voxel and its neighbours. Once a successful match is found, voxel modifiers change the attributes of the voxels. An overview diagram of the grammar section structure is depicted in Figure 4.1. This highlights sections across a simple snowfall grammar.

The following subsections describe the overall grammar sections in detail.

States and global variables

In the first section the user may define custom states assigned to voxels. These custom states allow the grammars to be more flexible, without the need for additional functionality to be added (additional symbols). The state values are stored in the voxel engine, which means states can be accessed by different CA. This allows cross-functionality and CA chaining during execution.

Each voxel may contain its own states, which users are able to use in a variety of ways. This includes: identifying voxels, storing custom voxel parameters and flagging voxels for further processing. Each voxel may acquire states, but by default has no state. The voxel's state is defined by a value. This is used to indicate degree. For simplicity, boolean values are simulated by limiting the state to an integer value of 0.0 or 1.0.

Each rule in the grammar may use the declared state. Declaring a state is similar in purpose to variable declarations in a programming language. First the declaration expresses the identifier "state", followed by the name of the state. The state "snowFallen" in Figure 4.1 shows an example of a declaration. Here, the state "snowFallen" identifies voxels that have changed to a white snow colour.

Global variables are declared in a similar way. In contrast to sates, global variables are not attached to voxels. Instead, global variables are attached to voxel models. Global variables are useful for storing globally constant information. For example, a variable could store the position of a light-source.

Both global variables and states are shared across all grammars for each model. This behaviour is determined by the voxel engine at runtime. Grammars each declare global variables with a particular name, where each name is associated with a data-structure in the voxel engine. The data-structure is created on demand, from the first grammar to use that variable name. Subsequent grammars that declare the same name use the existing data-structure.

Rule description

The rule section contains the production rules in order of priority (the first rule to match is applied). The rule is split into three different sections. First is the index voxel evaluator, second is the neighbourhood evaluator and last is the index voxel modification.

The first section is prefaced with the string "*Rule* :", the *rule-start symbol*. This clearly indicates the start of the rule, which can be easily determined by users unfamiliar to the grammar specification. The start symbol is followed by the non-terminal evaluators that evaluate aspects of a particular voxel. The evaluators are only concerned with the centred selected voxel and its construction. This selected voxel is also referred to as the "index" voxel in the grammar. This is used to indicate that the voxel will be the reference point for all neighbouring voxels in the rule. The end of the section is marked with the *index symbol*, represented as a "#", which is chosen for its visual similarity to a central voxel surrounded by neighbouring voxels in a grid.

The second section is where the evaluation of neighbourhood non-terminals are situated. The distinction between this section and the index voxel section is chosen for clarity. The neighbourhood section enables additional checks within the surrounding voxel neighbourhood of the index voxel. Locations within the neighbourhood are indicated through the use of movement non-terminals. They describe a direction of movement relative to the index voxel. By checking surrounding voxels the grammar allows for the creation of context-sensitive CA. The section ends when the modification symbol "→" (as described in Appendix A.1) has been used, which splits the left and right hand sides of grammar rules (or grammar productions). This is identical to the left/right hand syntax of other procedural grammars.

The third and final section, the modification area, contains modifier symbols. Symbol overloading between modifiers and evaluators reduces the memorisation needed by users, while the sectioning still allows visual clarity. The modifier symbols are potentially terminal, depending on the surrounding rules in the grammar. The symbols set the index voxel according to its specific properties of a voxel, such as colour or state. Following the modifier symbols, the end symbol "%" represents the end of the rule (and section). The index voxels are set once by each grammar rule during an iteration (assuming the rule passes evaluation and probability).

4.3.2 Voxel evaluators

Voxel evaluators are non-terminal symbols in a rule. They check aspects of a voxel, to match and filter them. This matching process determines whether or not a voxel will be modified. When an evaluator finds a match, it acts like a boolean function returning "true". A miss-match will likewise return a value of "false". The results from consecutive evaluators are strung together with boolean operators. The most used of these symbols is the "and" symbol (similar to the "&" symbol in programming languages, such as c++). White-space between two evaluators is parsed as an "and" symbol by default, unless explicitly stated otherwise. This is to improve clarity and conciseness in the grammar, because most rules rely on "anding" evaluators together.

Below we describe a few types of evaluators.

Active state

Every voxel can be either active or inactive (on or off). This grammar feature allows the creation of 3D structures or deformations, which would limit the grammar to 2D surface CA if left out. If the voxel is inactive then it has no states or variables (including colour), essentially unseen and invisible to the world. If a surface voxel were to become inactive, an indentation or gap (one voxel unit in volume) would be created at its position. The active state is predominantly used to guide the CA away from voxels that can be ignored.

The active or inactive evaluators are simplified to the words "on" and "off", respectively. This is to leverage the simplicity of common conversational language. An example of this is seen in the CA below 4.5, where any voxel that is on is turned off.

$$\begin{array}{l}\text{Rule: on \#}\\ \rightarrow\ \text{off \%}\end{array} \tag{4.1}$$

Colour evaluation

Each active voxel contains a $vec3$, which represents a colour. The colour evaluator forms the core method of the texturing CA, by finding colours in a local neighbourhood which match certain criteria. This enables the CA to exhibit colour-context awareness, which is is critical for surface detail.

A voxel's colour is represented by the three standard colour channels, red R, green G and blue B. Each channel is described by a floating point value between 0.0 and 1.0. This method follows the standard colour cube model, in which each channel represents an axis in a three-dimensional domain.

Colours can be exactly matched through the symbol $colour(r, g, b)$, where r, g and b are the primitives matched against a voxel's colour. The symbol also allows for individual tolerance values for each colour channel, $colour(r, g, b, tR, tG, tB)$. In this case a value is matched with a tolerance value (tR, tG or tB)

above and below each channel colour (r,g or b, respectively). For example, with values of $x = 0.1$ and $r = 0.5$, any red component between 0.4 and 0.6 would match. Similarly, evaluators for checking each colour channel are available. Individual tolerance evaluators for channels red R, green G and blue B are $colourR(r, x)$, green $colourG(g, y)$ and blue $colourB(b, z)$, respectively.

In addition to the exact match symbol, standard boolean operators may be used. These include the <,>,<=,>=,== operators. A description for the use of each of these operators is included in Appendix **??**. The boolean operators allow greater control over colour evaluation than the exact or tolerance evaluators. The example CA below4.2, finds a voxel with a colour of near black, and then removes the voxel from the scene (it is made inactive).

$$\begin{array}{l}\text{Rule: colour(0,0,0, 0.1,0.1,0.1) \#} \\ \rightarrow \text{ off \%}\end{array} \tag{4.2}$$

Besides the RGB colour cube evaluator checks, checsk are available for the HSV space colour cube. A voxel's HSV colour is similarly represented by the three channels, hue H, saturation S and value V. The HSV colour space provides a different way of comparing colours. For example it can be easier to match colours close in hue, without having the saturation or value impact the result. In the example $colourHSV(h, s, v, tH, tS, tV)$, h, s, v represents the channel values, while the tolerance values for each component are available by tH, tS, tV.

Another colour method is the ability to evaluate the colour of a neighbourhood voxel in contrast to the colour of the index voxel. This is useful for when similarity within neighbourhoods is required (or inversely, specifically not required). The difference functionality is provided by the operator $colourDiff(x, y, z, tR, tG, tB)$, where x, y, z corresponds to the neighbourhood position and tR, tG, tB is the colour tolerance. An HSV equivalent is provided by the symbol $colourDiffHSV(x, y, z)$.

Boolean "Not" symbol

When the boolean *not* symbol "!" is present before an evaluator, then the result of the evaluator is negated. This is a critical feature for the grammar to reflect standard boolean-based logic. For example, if the "not" precedes the exact match colour evaluator, the combination will match any colour that is not exactly equal. This situation is illustrated in the CA 4.3, where any voxel colour that is not pure black will match and be turned pure black.

$$\begin{array}{l}\text{Rule: !colour(0,0,0) \#} \\ \rightarrow \text{ colour(0,0,0) \%}\end{array} \tag{4.3}$$

Position evaluation

Every voxel has a position in space. The position corresponds to an integer value on the x,y and z axis, where voxels occupy a 1x1x1 unit cube volume. Similar to the colour evaluators, position evaluators can match exact positions within a tolerance, or by boolean comparators. The CA feature below, removes any voxel with and x position equal to five (resulting in a plane of empty space). This could also be useful to constrain CA to portions of the model. For example, not allowing growth outside set bounds. While not being a critical feature, we find this useful to create context-aware CA.

$$\begin{array}{l}\text{Rule: posX(5) \#} \\ \rightarrow \text{ off \%}\end{array} \tag{4.4}$$

Custom states

Each voxel may have a value associated with a custom state (as described in section 4.3.1). Custom states allow the creation of additional functionality to the grammar, without the need to change the specification. Additionally, states are store within the voxel engine, which are globally accessible to grammars. This allows cross-grammar functionality and CA chaining.

Voxel evaluators can be applied to these state values. As with colour evaluators, state evaluators can match by exact value, with a tolerance or by boolean comparators. This allows users to easily check for state values above, below or equal to a certain state. The example CA below 4.5, first defines a state "s", then removes all the voxels with a state equal to 0.

$$\begin{aligned} &\text{state s} \\ &\text{Rule: s == 1 \#} \\ &\rightarrow \text{ off \%} \end{aligned} \tag{4.5}$$

Is Selected

"Is selected", is a special case state which does not need to be declared. This is a boolean value only, which determines if a voxel is has been selected or not. This state is tied to the selection mechanism in the voxel engine, but is also usable by CA grammars. Selection is useful when only certain portions of the model need to be altered by a CA. For example, in the CA below 4.7 all the selected voxels are made inactive. Selection is achieved manually using the selection tool (described in section 5.2.3), automatically by the CA or a hybrid approach by using both. For example, a hybrid selection could "grow" manually selected areas through the use of a selection CA.

$$\begin{aligned} &\text{Rule: selected \#} \\ &\rightarrow \text{ off \%} \end{aligned} \tag{4.6}$$

4.3.3 Neighbourhood navigation

A voxel neighbourhood is a set of voxels in close proximity to a given voxel. Figure 4.2 shows a simple voxel neighbourhood in the shape of a cube. Traversing the neighbourhood is important when a CA is context sensitive. A context sensitive CA needs to be able to check states of voxels surrounding the queried voxel to find a context match. Neighbourhood navigation is handled in the grammar by using translation symbols, similar to those used in L-systems. The navigation symbols and neighbourhood evaluators are placed separately from the index voxel evaluators, as seen in Section 4.3.1. Each symbol shifts a lookup index according to a chosen axis. The index moves up and down along each axis by intervals of a single voxel. As an example, when moving along the y-axis of the grid, the symbols "$+y$" and "$-y$" are used for movement in the positive and negative directions, respectively. The symbols can also be stacked together to navigate through the neighbourhood by placing them consecutively. So the string "$+y\ +y\ +y$", will move the index three places along the positive y direction from the centre (index voxel).

In addition to the navigation symbols, curly braces, "{" and "}", can be used to show the scope of a symbol without having to negate the movement with another symbol. For example, "$+y\{\ \}$" has the same effect as

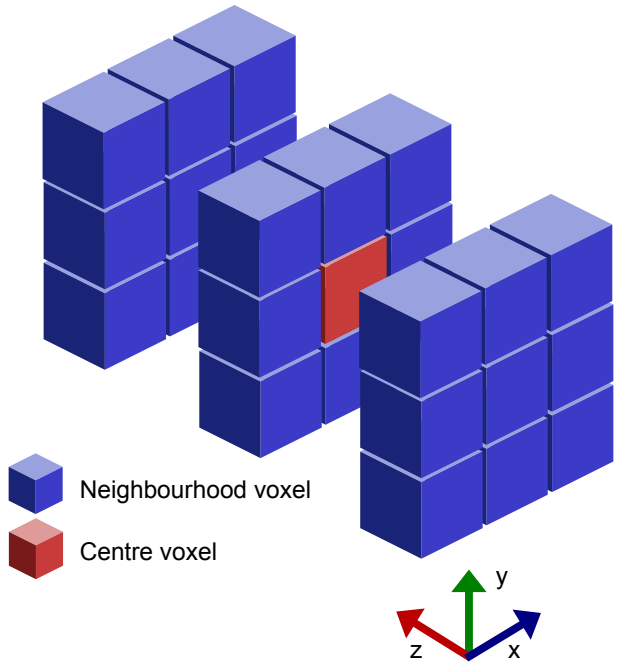


Figure 4.2: *This diagram depicts a 3x3x3 neighbourhood of voxels surrounding a voxel. The index voxel is marked in red, while neighbouring voxels are marked in blue. The space between voxels along the z-axis is exaggerated to highlight where the index voxel lies.*

"$+y$ $-y$". The use of curly braces is to leverage scope as commonly applied in programming languages. The example CA below, checks the if voxel above (one positive unit along the y-axis) the index voxel is on.

$$\begin{array}{l} \text{Rule:} \quad \# \\ +\text{y on -y} \\ \rightarrow \quad \% \end{array} \tag{4.7}$$

Navigation symbols are also available in the voxel modification section. Without this functionality only traditional CAs, where only the queried voxel is modified, would be possible. A problem with adding navigation is the modification precedence and parallel processing. A race condition can occur when a rule is modified from two different index positions in parallel. Depending on which index voxel is evaluated first can influence the final result. Additionally, a deadlock can occur when two threads attempt to write to the same location at the same time. Deadlocks are avoided trough the use of mutexes, which restrict modification to a single thread at a time. This can result in a lower overall speed-up from parallelisation when the modification section is the slowest operation, resulting in sequential execution. However, this is typically only true in a very simple CA with few evaluators, where a parallel speed-up is not as crucial. Race conditions rarely effect the outcome the overall CA, as each rule only one modification section. The only case where a single modification section of the rule could create different results, would be from the addition of probability. Otherwise, the modification is only duplicated. However, if probability is added then the unpredictable result is probably preferable.

4.3.4 Shape guided neighbourhood navigation

While the axis-aligned neighbourhood navigation symbols are useful, it can be cumbersome to create neighbourhood queries that span a great distance. To simplify this we additionally create shape-guided neighbourhood evaluators. These evaluators search a the neighbourhood in pre-defined volumes, such as cubes, spheres and rectangles. These function as short-cuts to searching large neighbourhood volumes, which are defined according to user set parameters.

As well as volume size paramaters, an additional parameter guides the CA to which boolean operator is applied the results for all the voxels in the volume. In the example $cube(or, 3, off)$, the logical "or" operation is be applied to find a single inactive voxel in a cubic volume of dimensions $3x3x3$. Additionally, these evaluators are stackable with other neighbourhood queries. For example, $+x + xcube(or, 3, off)$ will apply the same operation seen before but shifted by two units in the positive-x direction.

For efficiency sometimes a more efficient but less accurate shape evaluator is needed. These shapes types are provided in three different ways, full, cross and shell. As the names describe these enhance initial shapes evaluators by using either a complete, cross-section or outer shell approximation of the shape area. For example the operator "sphere(shell,3,off)" would find any inactive voxels which intersect the surface of a sphere shape with a radius of 3.

4.3.5 Voxel modifiers

Voxel modifiers are potentially terminal symbols in the modification section of each rule. Termination depends on how rules interact with each other. For example, a rule is terminal if voxels are set to a state that will never match other rules. However, any non-terminal rule that matches the state can break this behaviour. CA can be both non-terminal and terminal, so both behaviours are desirable in the grammar.

In contrast to evaluators, modifiers do not return a boolean value. Instead, they change the centre voxel within a neighbourhood (also referred to as the index voxel). The modifier will only change the states of a voxel if all the evaluators have passed. Including, the index voxel and its neighbour states. This is consistent with how traditional CA's work.

Colour modification

Colour modifiers change a voxel's colour, as a counterpart to the colour evaluators. Colour modifiers can set individual colour channels, red "R", green "G" and blue "B". Colours can be set to an exact value, similar to exact matching by colour evaluators. This is represented by the symbol "$colour(r, g, b)$", where r, g and b are floating point values for each colour channel. In a similar fashion to the tolerance colour evaluators, colours can also be set to a rough colour value, "$colour(r, g, b, r2, g2, b2)$". This is essentially chooses a colour between the colours r, g, b and $r2, g2, b2$. This is useful in situations where a random colour must be chosen, which would be difficult with the standard modifier. For example, colouring a model red with an amount of random noise applied to it.

Similar to the colour evaluator, a colour can be modified using HSV coordinates. This is provided by a the symbol $colourHSV(h, s, v, h2, s2, v2)$, identical in form to the equivalent evaluator. This is especially useful in situations where the random colours should conform in some way, such as choosing pastel colours with a similar saturation.

Standard mathematical operators may also be used to alter a voxel's colour. The operators include standard mathematical operators "+", "−", "∗", "/". These allow for greater flexibility in colour modification, which is also able to use the previous voxel colour. For example, in the expression "$colour(+, 0.1, 0.0, 0.1)$", a

floating point value of 0.1 will be added to the original red and blue colour channels. In another CA shown below 4.8, the red channel's intensity is reduced while the other channels are unchanged.

$$\begin{aligned} &\text{Rule: } \# \\ &\rightarrow \text{ colour(*,0.1,1.0,1.0) \%} \end{aligned} \tag{4.8}$$

In addition to the traditional operators, we also define a percentage operator "%" (not to be confused with the mod operator present in programming languages). This operator creates a colour from a percentage of two colours. This is defined by overloading the second colour in the colour symbols to define the percentage. For example, the symbol "$colour(\%, r, g, b, 0.1, 0.1, 0.1)$" creates a colour that is 90% the original colour and 10%.

State and selection modification

Before the voxels states can be successfully evaluated, they need to obtain a state. State modifiers are capable of giving voxels states, by creating and changing them. To change a state, the symbol $state(s, n)$ is used. Here s is the state name, while n is the new state value. If the voxel does not have a state, it is added to the state data-structure. The state data-structure is analogous to the voxel data-structure, where only the voxels that have a state are active (with a set value). Each state has its own data-structure to reduce search times. The "selection" state is a special state modified through the same symbol, where the state name is "selection". The CA example below 4.9, finds voxels that are selected and have a state "s" of value 0. Voxels that match this criteria, have their "s" state modified to a value of 1.

$$\begin{aligned} &\text{state s} \\ &\text{Rule: selected \&\& s == 0 \#} \\ &\rightarrow \text{ state(s,1) \%} \end{aligned} \tag{4.9}$$

Probability

To introduce stochastic behaviour to CA, we add probability values to modification areas. This is similar to the approaches seen in other grammars like L-systems and shape grammars. The probability value determines whether modifiers apply to the voxel or not. The probability is represented by a value between 0 and 1. For example, a value of 0.25 will succeed one in every four times (a probability of $\frac{1}{4}$). By default the probability for each modification area is 1.0. The probability example below 4.10, makes voxels inactive with chance of $\frac{1}{2}$.

$$\begin{aligned} &\text{Rule: } \# \\ &\rightarrow \text{ off [0.5] \%} \end{aligned} \tag{4.10}$$

4.4 Parser implementation

Once a grammar has been written in text form, it needs to be interpreted into an executable cellular automata. This is achieved in two steps. First the grammar is parsed by a parser defined using the Bison and Flex libraries. In our implementation, we choose to create a stand-alone parser. This parser is a separate

executable to the voxel engine. This is chosen to to limit the need for library dependencies in the engine and to avoid unnecessary code complexity.

The parsing process creates an abstract syntax tree describing the grammar. This syntax tree is then converted into an intermediate language. The intermediate language we choose to use XML. The choice was made due to the availability of libraries that support XML importing and exporting. Finally the XML can then be read by the voxel engine and turned into executable instructions.

4.4.1 Flex and Bison

Flex and Bison [34] are tools developed to generate compilers. Since creation, they also now commonly used to interpret structured input. Flex is used to create lexical analysers and Bison creates syntax analysers. Together, they are capable creating a parser that can interpret text to create an *abstract syntax tree*(AST). An AST, is an abstract tree-like description of the syntactic structure of text (or programming source code). The parser can use the AST form of text to interpret into other forms . These forms include; programming languages, binary object code and mark-up languages such as XML[1] or JSON[2] (see Section 4.4.2 for more information).

Flex

Lexical Analysis, also known as a scanner or lexer, separates input text into chunks called tokens. The tokens produced from text are separated into different labels by the lexer. The label types are defined in the lexer through regular expressions. *Regular expressions*, or regex, are sequences of characters that form search patterns. The search patterns are used to match strings of characters within text. The matches can then be labelled as different tokens. As a simple example, a set of label expressions could separate tokens into numbers and words. A more complex example could label subjects, objects and adjectives in a sentence. Equation 4.11 shows a lexer for a basic calculator.

$$
\begin{aligned}
&\text{//Floating point numbers and intergers are found using regularexpressions.}\\
&[0-9]+\backslash.\ [0-9]+\ \ \{\ \text{return FLOAT;}\ \}\\
&[0-9]+\ \ \{\ \text{return INT;}\ \}\\
&\\
&\text{//Mathematical operator symbols are found using matching character letters}\\
&\text{“}+''\ \ \{\ \text{return PLUS;}\ \}\\
&\text{“}-''\ \ \{\ \text{return MINUS;}\ \}\\
&\text{“}*''\ \ \{\ \text{return TIMES;}\ \}\\
&\text{“}\backslash''\ \ \{\ \text{return DIVIDE;}\ \}
\end{aligned}
\tag{4.11}
$$

Bison

Once the lexer has finished scanning the text, the syntax analyser can begin. The Bison analyser takes tokens and groups them into valid “sentences” defined in a grammar. Each “sentence” is a string of tokens that match syntax specified by the grammar. The analyser matches the sentence tokens against a tree of syntax expressions (or expression parse tree). A *syntax expression*, checks if a sequence of tokens match a

[1] W3Schools.com. *Introduction to XML*, `http://www.webcitation.org/6WJFDYVDC`, [Online; accessed 13-February-2015]

[2] ECMA-404 The JSON Data Interchange Standard. *Introducing JSON*, `http://www.webcitation.org/6WJEocKJu`, [Online; accessed 13-February-2015]

particular sequence. This sequence defines the syntax of the grammar. Each expression can also contain other expressions, similar to function calls in a programming language. The links between expressions and contained expressions, makes up the parse tree.

Syntax matches can be calculated by traversing through the expression tree in a depth-first search. The search follows the nodes in the tree toward the leaf nodes. When the leaf nodes are checked, then their results are propagated back to their parents. The propagation continues from these nodes, until the root has obtained results from all of it's children. Finally, if the root expression returns a match, a sentence is said to be syntactically valid.

A Bison example of this is show in Equation 4.12. This example shows two simple expressions. The first expression, called *statement*, is defined by two tokens "NAME" and "=", followed by an expression. *Statement* will return true if the tokens match and the inner-expression returns true. The second expression, name *expression*, describes the inner expression called by *statement*. It has two sets of token sequences, each containing three tokens. Matching either of these will make *expression* return true. The phrase "variable = 1 + 2", would be syntactically correct in this grammar. The phrase "variable = variable + 2", would be syntactically incorrect.

```
statement:
        NAME ' = ' expression
expression:
        NUMBER ' + ' NUMBER
        | NUMBER ' − ' NUMBER
```
(4.12)

While an expression can be syntactically valid, it can also be semantically invalid. Consider the statement presented by Chomsky [12], "Colourless green ideas sleep furiously". This statement is syntactically correct in the English language, while being semantically incorrect. An object cannot be both green and colourless, and ideas cannot sleep or have colour. To combat this, additionally sanity checks need to be added. A discussion on this is available in section 4.4.2

4.4.2 Abstract syntax tree

An abstract syntax tree (AST) is abstract representation of the syntax of structured text. An AST represents the syntax of a processed grammar in the form of a tree. This tree representation makes it easy for the parser to convert the grammar from one form to another. Typically ASTs are used to convert the text source code of a programming language into another. For example, a program written in C can be converted to assembly, then the assembly can be converted to object code (eg. machine code).

In our solution, we convert the grammar text into an AST, followed by a conversion to XML (see Section 4.4.3 for more information on the XML). The AST is an abstract representation of the grammar structure defined in Section 4.3. As the parser analyses the syntax of the input grammar, AST nodes are created for each detected expression. A visual description of the AST tree is shown in Figure 4.3.

At the beginning of processing, the root node of the AST tree is created. This contains a list of rule nodes, each of which contain a single processed rule. Next, the rule node is subdivided into three child nodes. These nodes describe the index voxel evaluation, neighbourhood voxel evaluation and index voxel modification sections. The *index evaluation section node* contains a single child node. This child node is an *evaluation node* node containing evaluation information (as defined in Section 4.3.5). The child node is also the first node in a linked list of evaluation nodes. The linked list is achieved by having each node containing a link to the next node in the list. The motivation for having a linked list of nodes is due to the way the evaluator expression is parsed in Bison. This is shown through a pseudocode example in equation 3.

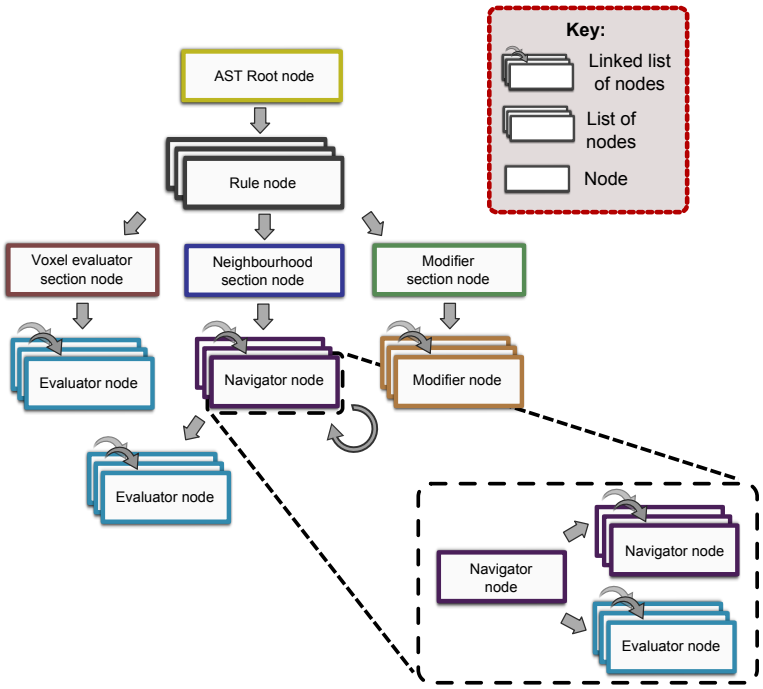


Figure 4.3: *This figure depicts the node heirarchy of the abstract syntax tree (AST).*

Algorithm 2 *Bison pseudo code to parse an evaluator*

```
1:  evaluator:
2:      //if an evaluator token "eval_node" is found, return the node
3:      eval_node evaluator
4:      {
5:         //create a new node
6:          Node newNode;
7:           //set the new node's child node to the result from the next expression
8:          newNode.childNode = evaluator;
9:         //set this node as it's parents child
10:         return newNode;
11:     }
12:     |   //else, no more evaluators are available in the section
13:     {
14:         //return a null node, this serves as the end of the list
15:         return null;
16:     }
```

The *modification section node* contains a single child node. The child node acts as the root node of a linked list of modification nodes, similar to the evaluation linked list.

4.4.3 XML conversion

To convert the abstract syntax tree (AST) into a usable format, we export the format into XML. The primary reason for this is to use XML as an intermediary language between the parser and the voxel engine. The primary difference between the XML structure and the AST structure, is the node linked-lists. The links in the linked-lists are broken, and instead are parented as children to a single parent tag. This is done to match the executable rule data structure presented in Section 5.1.

Algorithm 3 *A basic XML scheme*

```
1: <rulelist>
2:     <rule>
3:         <center>
4:         </center>
5:         <outside>
6:             <pos_y>
7:                 <off>
8:                 </off>
9:             </pos_y>
10:        </outside>
11:        <result>
12:            <colour r="1" g="1" b="1" tx="0" ty="0" tz="0" o="=" >
13:            </colour>
14:            <state name="snowFallen" s="1" >
15:                </state>
16:            <probability p="1" >
17:            </probability>
18:        </result>
19:    </rule>
20: </rulelist>
```

4.5 Summary

This chapter has presented our grammar to generate surface detailing voxel-based cellular automata. The grammar describes a CA, that is defined by a collection of rules, or productions. Each rule is applied by the voxel engine to chosen voxels during each iteration of the CA execution. Each rule is broken into three sections, the index voxel evaluation, neighbourhood voxel evaluation and voxel modification sections. The two evaluation sections look at the state of voxels surrounding the current voxel that the CA is executing on (the index voxel). By contrast, the modification section modifies the voxels within the neighbourhood if every evaluation check passes. The rule sections are populated with symbols which define which states to evaluate or modify. For example, the "colour()" symbol can both evaluate or modify a voxels colour, depending on which section it is found in. The grammar is then parsed into an abstract syntax tree, which is represented by XML. The XML can be loaded by the voxel engine and executed as a function tree, or CA.

In the next chapter we investigate the tools we provide when executing the CA.

Chapter 5

Cellular automata execution

The final section in the overall system is the cellular automata (CA) executor. Once a grammar is parsed (see Chapter 4) and a voxel model is loaded (see Chapter 3), a CA can be executed on the model. The CA executor is a section of the voxel engine, which controls how and when CA are applied to voxel models. First the XML form of the parsed grammar is transformed into an executable rule data-structure. This rule data-structure is the executable CA. Next the executable CA can be applied to the voxel model in a number of iterations. Section 5.1 and section 5.2 describe the rule data-structure and rule execution, respectively.

5.1 Rule data structure creation

The parser described in Section 4.4, parses and creates XML representations of input grammars. The XML files can now be loaded by the voxel engine and converted into an executable form. The executable form is structurally similar to the abstract syntax tree seen in section 4.4.2. The tree-like data-structure is created, with rule nodes branching off of a root cellular automata rule node. Each rule node branches to three section nodes: index voxel evaluations, neighbourhood evaluations and index voxel modifiers. Each of these sections are first introduced in the rule description section 4.3.1.

5.1.1 Index voxel evaluation

The index voxel evaluation section contains all the nodes that evaluate properties of an indexed voxel. An indexed voxel is one of the voxels currently being evaluated during a search through every voxel. The indexed voxel's properties are evaluated according to each child node attached to the sections node.

Evaluation nodes

Each evaluation node is defined by inheriting from an abstract "eval node" class type. This class has an "evaluate" abstract method which must be overloaded by each child type. This abstract method returns a result as a boolean value, as seen in section 4.3.2. The function receives a voxel coordinate as its only parameter by default. The coordinate is used by the "evaluate" function to determine which voxel to check. When an evaluator finds a voxel match or mismatch it returns a result of "true" or "false", respectively.

From the abstract class definition, each child type must then overload the "evaluate" function with its own functionality. For example, the colour evaluator stores colour and threshold parameters, and tests if the voxel's colour matches the range.

Evaluation node results

When a rule evaluation is requested, a depth-fist-search tree traversal begins. During the traversal each node in the tree is evaluated and results are propagated backwards. However, this is not a typical homogeneous tree traversal, instead each parent node will dictate how or when its child results are determined.

For example, the results from the children of the index voxel evaluation node are strung together by boolean logic operations. By default the "and" (or "&") operator is used to form a final result from the children. When applying the "and" operation, the first mismatch found will return a negative result for the whole group. This is done for efficiency, where at worst case every evaluator is checked and at best the first mismatched evaluator is checked.

In addition to the section nodes having evaluator node children, each evaluation node may have evaluator child nodes. When the children are evaluated is dependant on the type of parent evaluator node. For example, the "not" (or "!") boolean operator could be considered to be a special evaluator node with one child. When a not node is evaluated, it evaluates its children first, then returns the negated result.

Similarly when an "or" operator node lies between two evaluators the evaluators on the left and right of the node are added to the "or" node as two separate sub-tree's. The "or" operation can then be applied to the results from both sub-tree. For consistency in the tree construction, standard operator precedence is applied in reverse (lower precedence near the root of the tree). The reverse construction allows the depth-first-search to find the operators in order of standard precedence. For example, the "and" operator is evaluated before the "or" operator.

5.1.2 Neighbourhood voxel evaluation

The neighbourhood voxel section is very similar to the index voxel evaluation section, with a few modifications. Like the index voxel section, the neighbourhood section is built up of evaluation nodes 5.1.1. However, it also includes neighbourhood navigation nodes, which alter the voxel coordinate passed between evaluator nodes.

Our neighbourhood navigation system is introduced in section 4.3.3. As described in the section, neighbourhood areas within the grammar are nestable and have scope. This directly translates to neighbourhood nodes that can contain evaluation nodes and other neighbourhood nodes as children. At each new neighbourhood node the coordinate passed to its children are modified by the nodes value. For example, the node representing "$+x$" will pass a coordinate moved one unit in the negative x direction from the coordinate it received. If the node received $(0, 0, 0)$, it would then pass $(1, 0, 0)$ to its children.

As an example of neighbourhood node nesting, the rule "$+y\{\ +y\{colour(white)\}\}\ -x\{\ +y\{colour(white)\}\}$" will result in the rule tree shown in Figure 5.1. This CA neighbourhood section will check if the voxel two units above, as well as diagonal, to the index voxel is white. At each voxel node the shifted coordinate values from a starting coordinate of $(0, 0, 0)$ are shown.

5.1.3 Index voxel modifier

In contrast to the index voxel and neighbourhood evaluation sections, the index modifier section modifies the index voxels. We limit voxel modification to the index voxel only (no neighbourhood modifications),

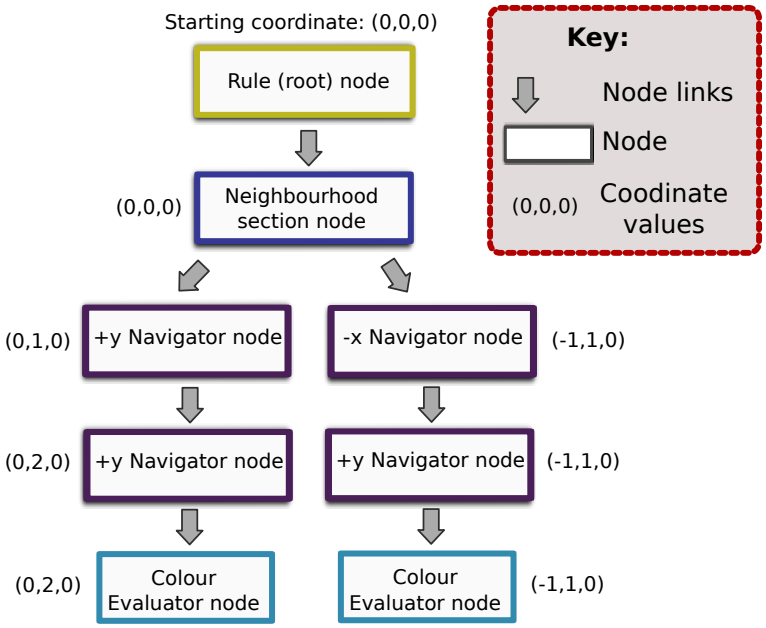


Figure 5.1: *This diagram depicts the rule tree structure resulting from the neighbourhood evaluation section "$+y\{\ +y\{colour(white)\}\}\ \ -x\{\ +y\{colour(white)\}\}$". Additionally, the resulting voxel coordinate is shown at each stage from an original coordinate value of* $(0, 0, 0)$.

to be consistent with traditional cellular automata. This limitation simplifies the CA parallelisation process, but can lead to verbose CA grammars. A discussion on CA parallelisation issues can be found in section 5.2.1.

Modifier undo framework

For debugging purposes, we create a rule undo stack. This undo stack, allows us to apply and reverse rules instantly without having to recompute the results. This is useful when debugging CA, as you can visually step through each rule as it is applied over the grid.

The stack we use is built upon Qt's undo framework[1]. The framework automatically handles the undo state management, which consists of a stack of commands which have undo and redo functions. Each command stores information about how an object is modified, and how to return it to its original state. In the command, only the resulting modification step is stored, not how the result is found.

In our system moving up or down the command stack allows a user to undo and redo changes made by the CA. The method saves memory by only storing the modified voxels, in contrast to storing the entire grid after each CA iteration. Additionally, the CA does not need to be recomputed when moving along the stack. The commands are created according to modifiers that are applied after a successful evaluation. The commands only need to store the voxel information being modified, as well as the modification parameters. For example, a colour modifier will store the original or modified colour state and the modification reverse modification procedures.

The stack is only modified when a CA is run. The grid state that the CA is run on depends on the stack index. If the index is at the top of the stack, then the CA is run on the last grid state and grid state from

[1]The Qt Company. *Overview of Qt's Undo Framework*, `http://www.webcitation.org/6Z8NceQaa`, [Online; accessed 8-June-2015]

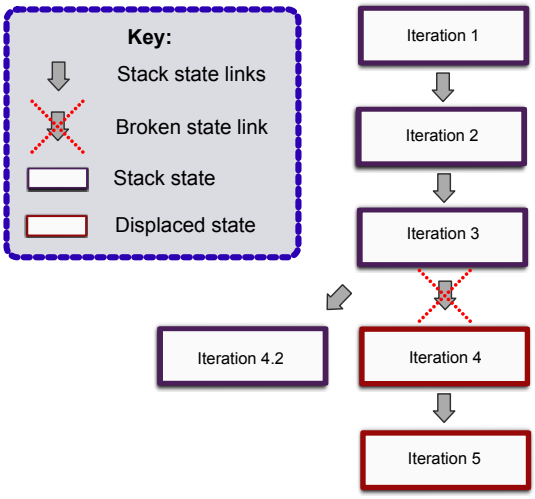


Figure 5.2: *This diagram shows how stack states diverge when a new cellular automata is run. Here a CA is run on grid state "Iteration 3", creating state "Iteration 4.2". The states "Iteration 4" and "Iteration 5" are now displaced from the stack and removed.*

the new CA is pushed onto the top of the stack. If the index is below the top of the stack then new CA state displaces the stack states above the index. The displacement is depicted in Figure 5.2.

5.2 Rule execution and additional parameters

To add to the grammar and navigation capabilities discussed earlier in this chapter, we also add a number of useful tools to enhance cellular automata execution. The first enhancement is computing CA in parallel to save time. The second is orienting the voxel neighbourhoods according to the surface normals. Finally we introduce our voxel selection mechanism.

5.2.1 Parallel computing

Running cellular automata over large grids can make execution times extremely slow. This is predominantly caused by applying the CA to every active or selected voxel in the grid. To speed the process up, the CA can be applied to voxels in parallel.

Applying cellular automata in parallel is typically an easy process. In our implementation, each CA and contained rules are run sequentially. When applying a rule to the grid, a thread is spawned for each voxel (we identify these voxels as index voxels). The threads are then applied to index voxels in parallel to one another. The number of simultaneous threads running is limited to the number of processing cores available. This is to reduce the overhead of thread management.

When running a cellular automata over a grid, each voxel is visited exactly once per rule. From this we can see that when only the index voxels are modified, the problem is embarrassingly parallel. This is evident as each CA thread may execute without worrying if another thread is accessing the same data. In contrast, neighbourhood modifications can lead to race conditions. This is caused by threads modifying the same voxel from different neighbourhoods.

While our implementation does not attempt to eliminate race conditions, we avoid deadlocks through the use of mutexes.

5.2.2 Tangent-space neighbourhood orientation

In addition to the neighbourhood navigations provided by the grammar, we use a neighbourhood orientation technique. This neighbourhood orientation method modifies the navigation rules in the models tangent-space. In tangent-space, instead of the navigation rules following the x, y, z axes, they follow the tangent-space axes instead. These axes are referred to as the normal, bi-tangent and tangent unit vectors or the u, v and w unit vectors. For example at the bottom of an object, the navigation command $+y$ would normally move in the direction $< 0, 1, 0 >$, but could instead move along a normal direction of $< 0, -1, 0 >$. The red and blue arrows show the normal and tangent unit vectors for points on a cross section of the sphere, respectively.

Unit vector construction

The unit vectors are constructed from the vertex normal and UV coordinates for each polygon. From these, each voxel is assigned their own set of unit vectors using barycentric coordinate interpolation. The interpolation is calculated during the voxel colouring process 3.3.2. From the vertex normals, we calculate the bi tangent and tangent vectors for each vertex. Each vertex can have infinite possible orientations for tangent-space. Additionally each vertex can have its own orientation, which does not align with its neighbour vertices.

To maintain a consistent orientation, we use calculate the unit vectors by the direction of each model's UV coordinates. This is a common technique used in normal mapping, known as Lengyel's method [2]. From the method, the u and v axes (tangent and bi-tangent unit vectors) will follow the increasing U and V coordinates, respectively. To calculate the UV direction, we use equation 5.1. In the equation, the vector representing the change in position, $\vec{q_i}$, can be represented as a combination of components from two orthogonal directions, $\hat{u}$ and $\hat{v}$. By equating the change in position and the direction components, we can solve for the $\hat{u}$ and $\hat{v}$ unit vectors. From two vertices we can calculate a value for $\vec{q_i}$, and using the change in UV coordinate values we can find the components of the unit vectors.

$$\vec{q_i} = \Delta U_i \times \hat{u} + \Delta V_i \times \hat{v} \tag{5.1}$$

By using three vertices and substitution we can produce equations for the $\hat{u}$ 5.3 and $\hat{u}$ 5.4 components, respectively. In both equations the value r 5.2 is a common factor. This method however, produces inconsistencies along the texture seams. These can lead to two adjacent faces having tangent space orientations that are rotated at an angle of π to one-another. During oriented neighbourhood navigation this can result in mirroring traversals near the seams. To remove or minimise this effect, a different tangent space calculation could be used or an additional smoothing step can be adopted. We use the later, where the orientation at vertices shared by multiple polygons are averaged between each polygons' orientation.

$$r = \frac{1}{\Delta U_1 \times \Delta V_2 - \Delta V_1 \times \Delta U_2} \tag{5.2}$$

$$\vec{u} = r \times (\vec{q_1} \times \Delta V_2 - \vec{q_2} \times \Delta V_1) \tag{5.3}$$

$$\vec{v} = r \times (\vec{q_2} \times \Delta U_1 - \vec{q_1} \times \Delta U_2) \tag{5.4}$$

[2] Lengyel, Eric. *Computing Tangent Space Basis Vectors for an Arbitrary Mesh*, `http://www.webcitation.org/6ZSP0VLP2`, [Online; accessed 16-June-2015]

Orientation variability

A common problem with tangent-space orientation is that occasionally the orientation can be oriented incorrectly to the aliased surface. This can result in surface skimming without connecting, which can result in a premature termination point for the CA. To account for this we add variability to the tangent-space orientation. This is achieved by navigating in the original orientation, as well as a rotated orientation. We rotate the orientation by a user defined amount around the tangent direction for the bi-tangent navigation, and the bi-tangent direction for the tangent navigation.

While this can help connect with the surface in a near tangent miss, it can result in false-positives. For example, the standard orientation as well as the rotated orientation can return true. This can be minimised by limiting the signed distance field (SDF) value of a connection (limiting it to an are local to the surface), or by constructing a CA that will only propagate further if its neighbourhood matches a certain requirement (such as no voxels above it).

5.2.3 Voxel selection

Voxel selection is a tool we use to apply CA to only selected portions of a model. The selection tool works by casting a ray from a screen-space point into the scene. When the ray collides with an active voxel, the active voxel and a spherical neighbourhood around it are marked as selected. The radius of the selection sphere can be defined by the user for fine to coarse control.

Casting a ray in the scene

First, two world positions are calculated from a 2D screen-space point. The screen-space point is provided by the graphics API as a coordinate with axis values between 0 and 1, where $(0, 0)$ is the bottom-right hand point of the view window and $(1, 1)$ is the top-right. The world positions correspond to points on the camera's near plane (conceptually similar to the cameras "screen") and the far plane (the furthest viewable plane away from the camera). The "view frustum" created between the near and far plane, is calculated and oriented according to the camera's direction and perspective in the scene. The near and far plane are placed at a distance z away from the camera. From the z value of near and far planes we can obtain two 3D coordinates from the screen-space point, the near and far points respectively.

The world positions of the near and far points are calculated by multiplying the screen coordinate (including z-value) by the inverse of the view-projection matrix. The standard view-projection matrix is used to rasterise a 2D pixel image from the view frustum. We can now cast a ray between the near and far points. The ray cast logic is provided by an accelerated ray marching OpenVDB tool. This ray marching tool, will march rays from the near point to intersecting active voxels progressively. This is achieved through a digital differential analyser method (DDA).

5.3 Summary

In this chapter we describe our method of executing CA on voxel models. The CA is represented by an executable function tree, which the voxel engine creates from the XML file created by the parser. Each symbol in the grammar has a corresponding function created in the engine. These functions execute the parameters defined by the grammar, including the use of probability and boolean logic operators. A number of optional techniques are available when executing the CA through the engine. The first of these is parallel CPU optimisations, which greatly reduce execution time. Due to the execution speed-up, the optimisations are enabled by default. However, the number of parallel execution thread can be modified in the engine,

which is useful in situations such as performance testing. Second is the neighbourhood orientation extension. This extension enables users to track neighbourhood navigation symbols according to surfaces tangent-space, rather than the standard global space. This can greatly reduce the number of rules in the input grammar, by making surface detailing CA more rotationally robust. Finally, the voxel selection tool allows manual picking and painting of voxels that the CA can execute over. This can be useful when a user only wants to execute the CA on a portion of the model.

The following chapter shows the results of the CA system we have presented. In the chapter we review the system by reproducing CA from other work, and analysing the systems performance.

Chapter 6

Results

This chapter describes testing of and experimentation with the presented grammar system. To evaluate the system we apply two main types of testing: test-case and performance analysis.

To analyse how versatile and capable the grammar system is, we create a number of cellular automata (CA) test-cases. The test cases are chosen to test CA characteristics based on our own requirements and those present in other work. We then create these cellular automata using the grammar system. By using these standard test cases we demonstrate the versatility of the grammar, while also ensuring that using the grammar does not hinder the creation of complex CA.

As the grammar system is only a proof of concept, we do not include tests that look at the ease of writing grammars in comparison to manual programming or a more visual creation process. Furthermore, a realism comparison between the result meshes and meshes in other work is not conducted. Both of these testing criteria are out of the scope of this research, however they should be investigated in future work.

Next, performance tests are run to determine how resource intensive and scalable the overall system is. We analyse the running time of the mesh-voxel model conversions, CA execution and grammar parsing subsystems. Additionally, we calculate the memory usage of the system, including the resulting mesh model and grid.

6.1 Model preparation

To test the cellular automata, we sourced example models that we could modify. The selection criteria for these models (listed below) follow from the constraints imposed by our system and the OpenVDB library. Finding models that fit all the criteria can be difficult as artists and studios adhere to their own standards, which can often lead to conflicting requirements. Many of these missing or conflicting requirements can be fixed by using modelling tools and packages such as Blender[1] and Meshlab.

Our model criteria are as follows:

- Each vertex must have associated UV-coordinates.
- Each vertex must contain normal vector information.
- Each mesh must be a closed 2-manifold surface (water-tight).

[1] Blender.org. *Blender*, `http://www.webcitation.org/6aAHxPicj`, [Online; accessed 20-July-2015]

The UV-coordinates are used to calculate the tangent and bi-tangent unit vectors for each vertex. Next, the vertex normals are used by the cellular automata and OpenGL shader programs. Finally, the model meshes must be water-tight for OpenVDB's voxelisation tool. Additionally, it must be noted that while each mesh must have UV coordinates, they do not need to have associated texture or vertex colour information. In the case that no texture image or vertex colours are found, the system assumes each vertex is coloured by the specular, diffuse and ambient colours defined by the model's material (if no material information is found then the model is rendered as a white surface by default).

6.1.1 UV-coordinate and normal vector requirements

As our system needs UV-coordinates to calculate tangent and bi-tangent vectors, we assumed that each mesh contained UV-coordinates at each vertex. If a model did not have UV-coordinates, then we unwrapped the mesh in Blender. The unwrapping process creates a 2D representation of a 3D surface, similar to how a 3D box is created from a flat 2D sheet of cardboard, only in reverse. This can be achieved by creating cutting-lines across the surface, which can be used to unwrap the 3D surface onto a 2D plane. There are a variety of unwrapping tools available in Blender, and we carefully chose the tool according to the shape of each surface. For example we used the sphere-map based unwrap tool for simple surfaces, or a texture atlas unwrapping tool for more complex surfaces.

Similarly, Blender contains tools to calculate vertex normals of the surface. The standard tool calculates normals according to the polygons orientation, while the normal smoothing tool interpolates the normals to reduce flat-shading artefacts.

6.1.2 Manifold mesh requirements

Our system assumed that each model was composed of one or more manifold meshes. However, we often found model files which do not fit this assumption. Typically we found meshes that had holes and non-manifold polygons or edges (such as a disconnected quad). We fixed these holes by either using Meshlab's hole filling algorithms, or manually filling the holes with Blender's edge-loop face filling tool. Next we used Blender's non-manifold edge select tool to either remove or enclose non-manifold faces and edges. For example, a non-manifold polygon may be fixed by a small extrusion from the polygons face. While this method typically worked, we also needed manual inspection and modelling.

Another common problem was when a manifold surface is split into multiple meshes. A quick solution to this, is to combine these meshes in Blender and remove any duplicate vertices, edges and faces.

6.2 Test-case evaluation

To test the grammar system we evaluate a number of test cases. These test cases are chosen to highlight core features of our grammar, and also show that the grammar does not hinder the creation of CA. To show how capable the grammar system is, we reconstruct a variety of CA found in other work. These test-cases are varied in order to demonstrate the versatility of the grammar, including colour sensitive, eroding and structure creating CA.

The following rendered images present the results of the CA we created, with the images showing the progress of the cellular automata at different stages. Additionally, the grammars for each CA are available in Appendix C. The images are rendered using the OpenGL debug software, including diffuse and specular shading from GLSL shader programs. The quality and resolution of these images could be improved in future work by rendering the final mesh with a global illumination rendering program, such as the one included in the Blender modelling package.

The cellular automata are run on a variety of input models to limit bias. The models are also chosen to represent the versatility of the model converter, including shapes that have an irregular topology. For example we use models which have a genus greater than 0, which contain one or more topological holes. All of the models used in the results come from public domain repositories, such as Stanford's 3D scanning repository[2] and Keenan Crane's model repository[3].

6.2.1 Acid CA

To represent the erosion capabilities of our system, we create a red acid effect. This effect is shown in three images within Figure 6.1, showing the original mesh (6.1a), midway progress (6.1b) and final output mesh (6.1c). The acid effect CA is run on Crane's "Yeah Right" model at a grid resolution of [827×751×1440] voxels. The model is notable for having a genus of 131 and is notoriously difficult to parameterise. As the results show, our system has no trouble converting the mesh and colouring it.

Before the acid CA is run, a fast seed placement CA places a few red "seed" points across the surface of the CA. Next the CA spreads the red colour volumetrically across the mesh according to a set probability. Once a red voxel is surrounded by red voxels to a radius of 3 voxel units, the acid effect works by making the surrounded voxel inactive. The result after a number of iterations is a Swiss cheese-like effect with red rings around the eroded areas.

6.2.2 Moss CA

In contrast to the erosion evident in the acid CA, the system is also capable of creation-based CA. To illustrate this we use a moss growing CA on Stanfords "Dragon" model at a grid resolution of [441×733×992] voxels. The effect of the CA is shown in a progression of four images in Figure 6.2. The CA simulates the natural moss growth and colouring seen on historical outdoor statues. Before the moss growth CA is run, a placement CA seeds moss randomly across the surface. The growth CA then grows the moss laterally outwards across the surface of the model, with a maximum height away from the surface in the direction of the surface normal. This restricts the growth to a thin layer of moss over the surface. The new moss surface is coloured green with a small variance in hue, and larger variances in saturation and value.

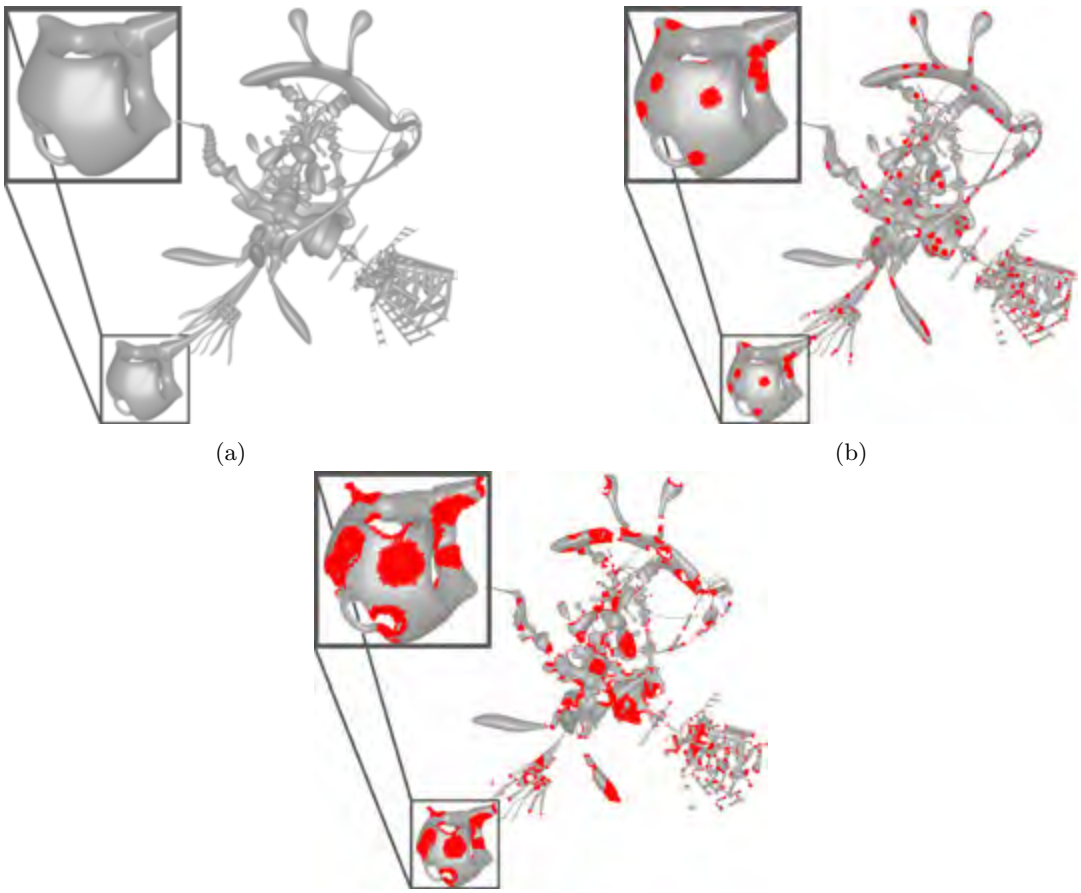
6.2.3 Camouflage and fracture CA

To show a more complex use case with changing state and colour properties, we introduce camouflage and fracture CA. To start off, the camouflage CA is applied to Crane's "Bob" model of a rubber duck with genus 1 at a grid resolution of [930×488×733] voxels, as seen in Figure 6.3. First the camouflage seed points are randomly spread across the surface of the model, each seed having a state pointing to one of the pre-defined camouflage colours. Next the camouflage colours are spread out in a sphere from the seed points, setting each voxel without a camouflage state to the state and colour of the seed point. These set voxels now act as seed points and the process continues until almost every voxel is set to a camouflage state.

Two notable artefacts can be seen in the example shown in Figure 6.3c. The first, is where camouflage seeds of the same colour are placed too close to each other. This can cause some of the resulting camouflage patches to be noticeably larger than others. The second, is a similar situation where two different colour seeds are placed in very close proximity to one-another. The sphere check of these close seeds can cause a dithering-like effect of the two colours, where one colour is more dominant than the other. This could

[2]Stanford Computer Graphics Laboratory. *The Stanford 3D Scanning Repository*, `http://www.webcitation.org/6b4WwqNmA`, [Online; accessed 26-August-2015]

[3]Keenan Crane. *Keenan's 3D Model Repository*, `http://www.webcitation.org/6b4WwFZQz`, [Online; accessed 26-August-2015]



(a) (b)

(c)

Figure 6.1: *These series of figures show the various stages of an acid CA executed on Keenan Crane's "Yeah Right" model. (a) the original mesh. (b) the output mesh after an initial seed CA pass, followed by 10 iterations of the acid CA. (c) the final mesh after another 10 iterations of the acid CA*

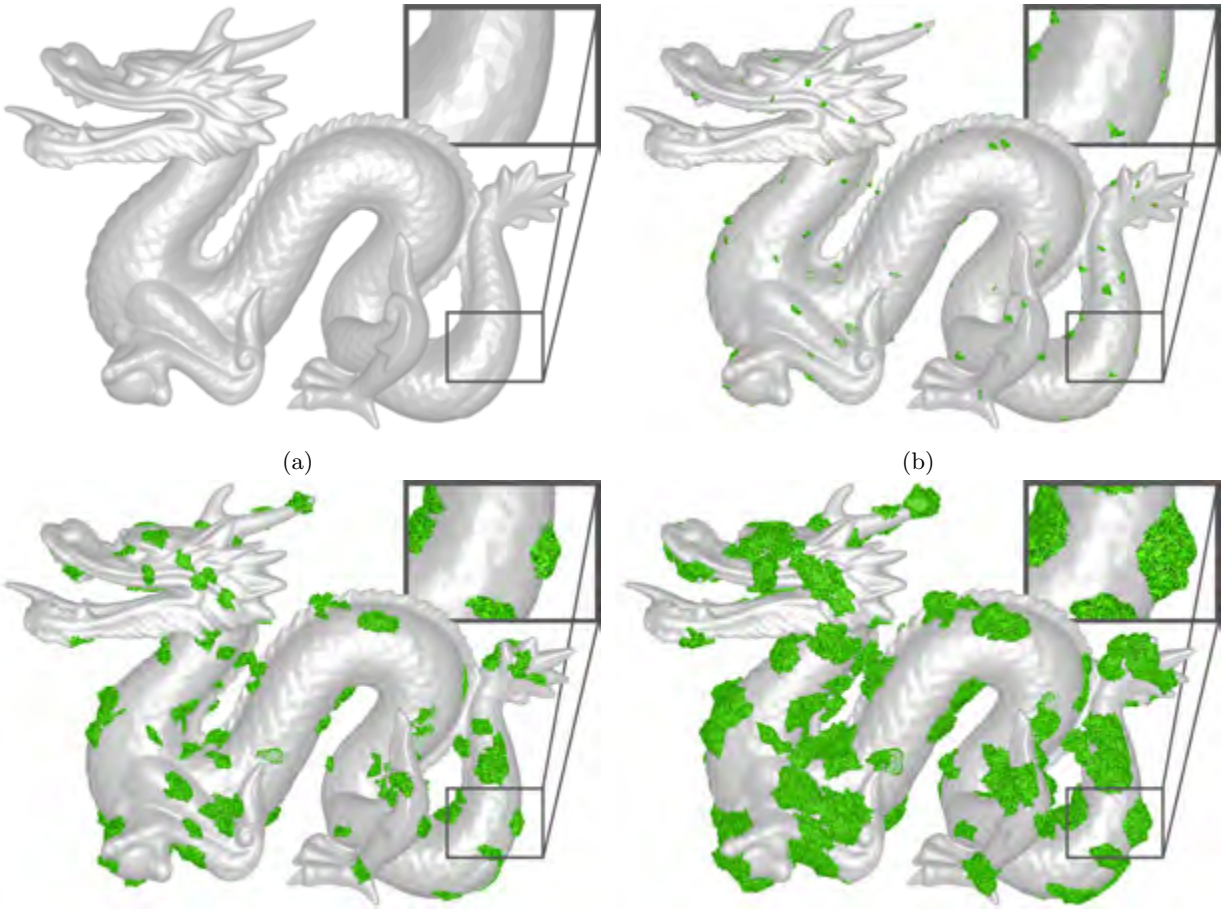
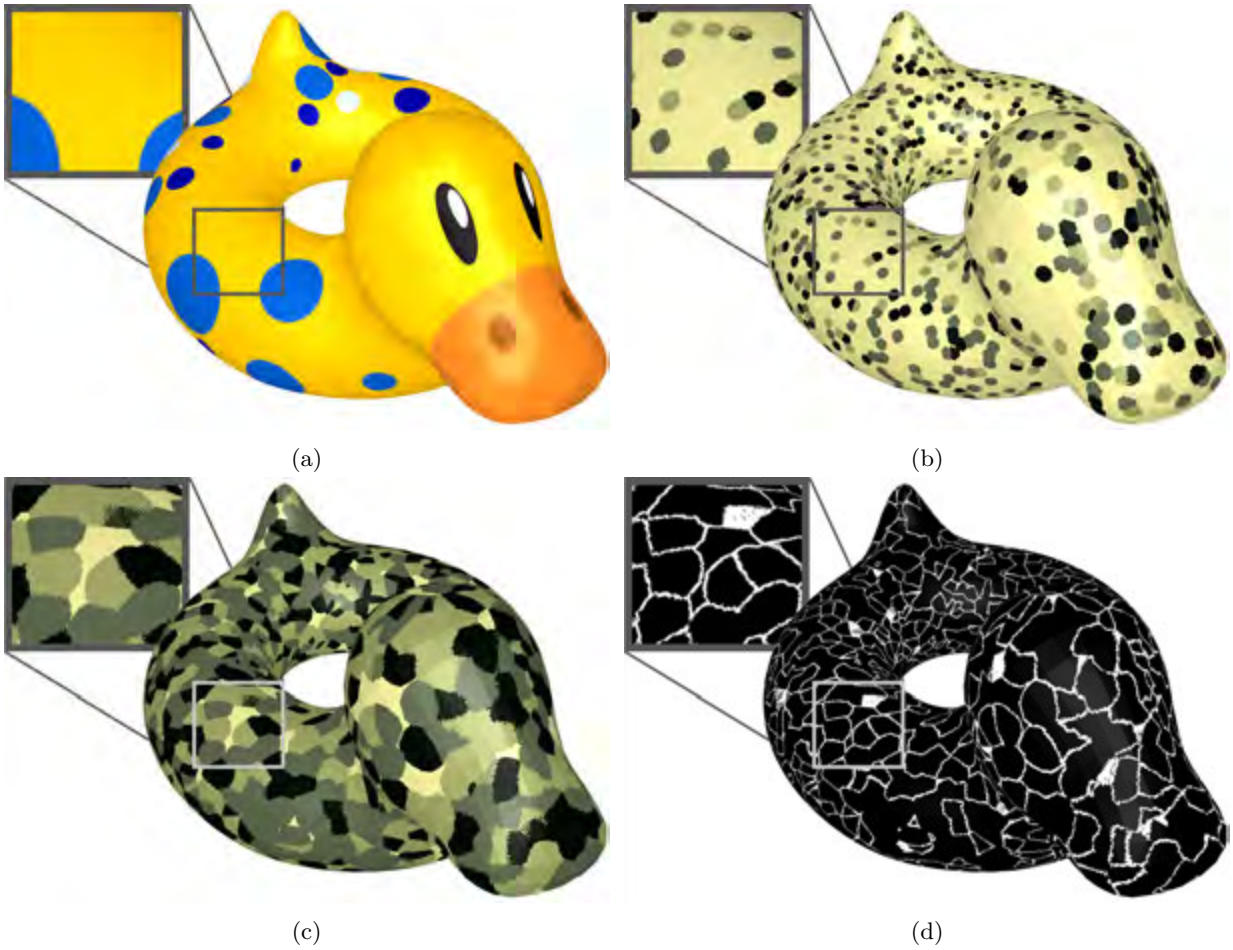


Figure 6.2: *These series of figures show the various stages of a moss CA on Stanford's "Dragon" model. (a) original mesh, (b) a seed pass and 20 iterations, (c) after 40 iterations, (d) after 80 iterations.*

be mitigated by limiting the proximity between seed points or removing seed-points surrounded by another patch.



(a) (b)

(c) (d)

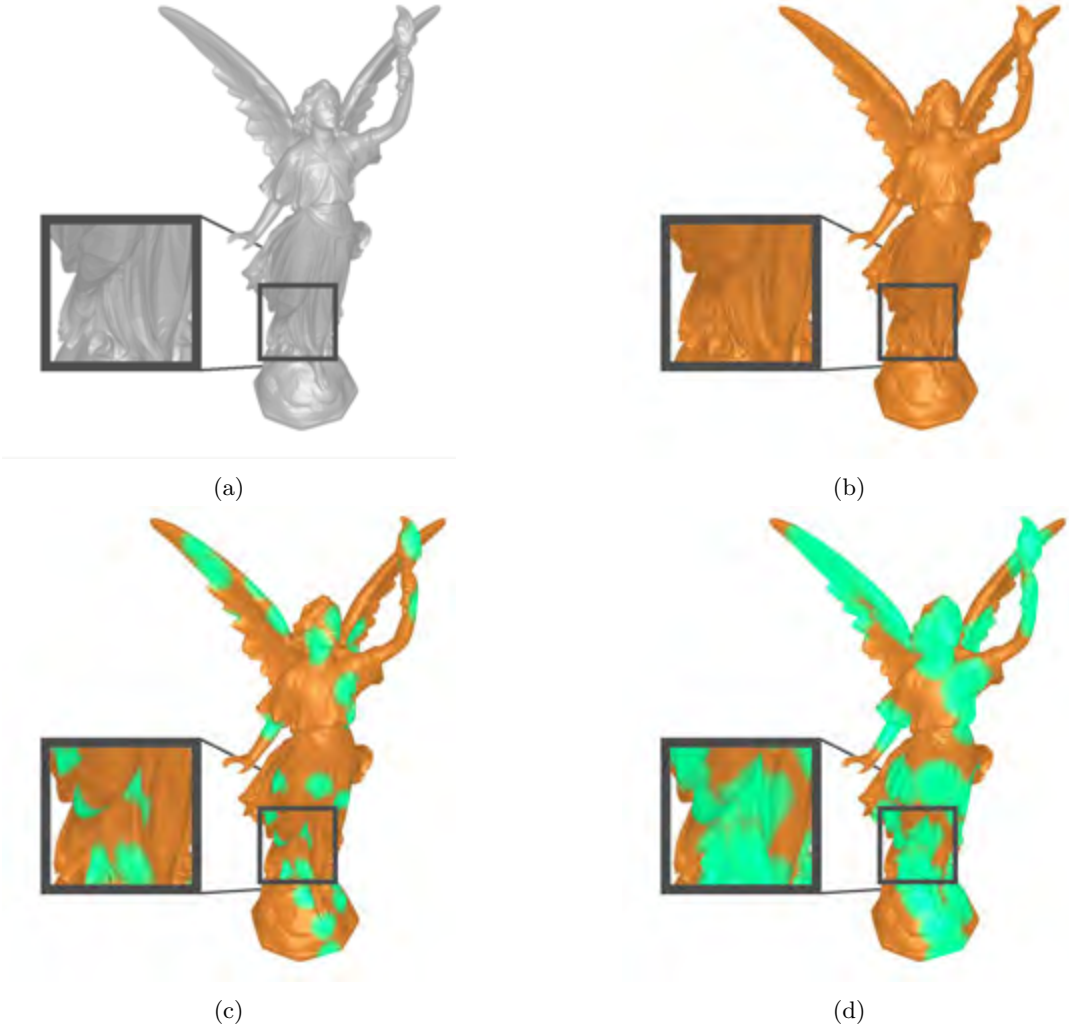
Figure 6.3: *These series of figures show the various stages of a camouflage and crack CA on Keenan Crane's "Bob" model. (a) original mesh, (b) after seed pass and 15 iterations, (c) after 30 iterations, (d) after a fracture CA pass.*

The fracture effect is caused by running a find colour difference check after the camouflage CA. The CA colours the index voxel white if the colour of a neighbourhood voxel is different to the index voxel by a tolerance amount. Otherwise, the index voxel is coloured black. Fracture-like lines are created along the boundaries of the camouflage patches, where the colour of the surface changes significantly. The colour difference CA is versatile, and can also be used to visualise colour changes for other CA results and models.

6.2.4 Patina CA

Another CA we have created simulates the oxidation process on copper, creating a layer of patina similar in colour to the Statue of Liberty in New York. To illustrate this we execute the copper CA on Stanford's "Lucy" model at a grid resolution of [517×1041×464] voxels, as shown in Figure 6.4. Before the patina CA is run, the model is coloured a copper colour with a small amount of speckled variation (random saturation

and value). At the same time, small spots of patina are seeded randomly across the surface of the model. The patina CA works by slowly seeping a patina colour into the surface voxels. The seeping is simulated by moving the colour of the copper voxels toward the patina colour by a small percentage at each iteration. Similar to the Acid CA, the patina spreads across the surface from seed patina voxels.



(a) (b) (c) (d)

Figure 6.4: *These series of figures show the various stages of a patina on copper CA on Stanford's "Lucy" model. (a) the original mesh, (b) initial copper colouring and patina seed CA passes, (c) after 60 iterations, (d) after 120 iterations.*

6.2.5 Plasma CA

The plasma CA works in a very similar way to the patina CA, with the difference of adding a set colour at each iteration instead of moving a starting colour towards a target colour. To demonstrate the effect we use Blender's "Suzanne" model at a grid resolution of [918×663×579] voxels, shown in Figure 6.5. Before the plasma CA is run, a CA colours the model blue followed by placing random yellow plasma seed voxels across the model's surface. Next, the plasma CA works by making voxels in the neighbourhood of plasma voxels randomly turn into yellow plasma voxels too. After each iteration the plasma CA "ages" all of the

currently active plasma voxels towards a red colour. This ageing process results in sunburst-like colouring in the plasma areas.

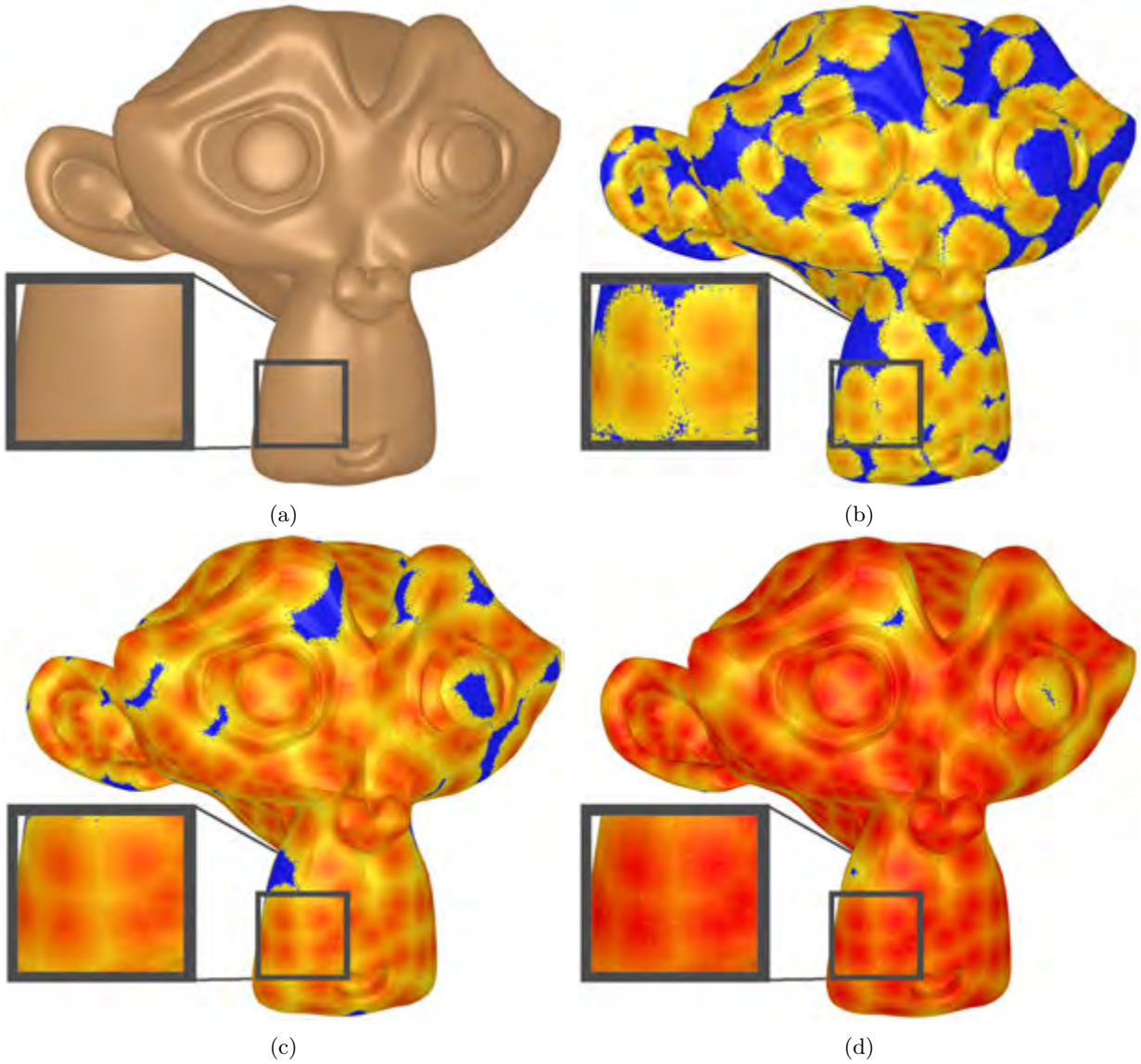
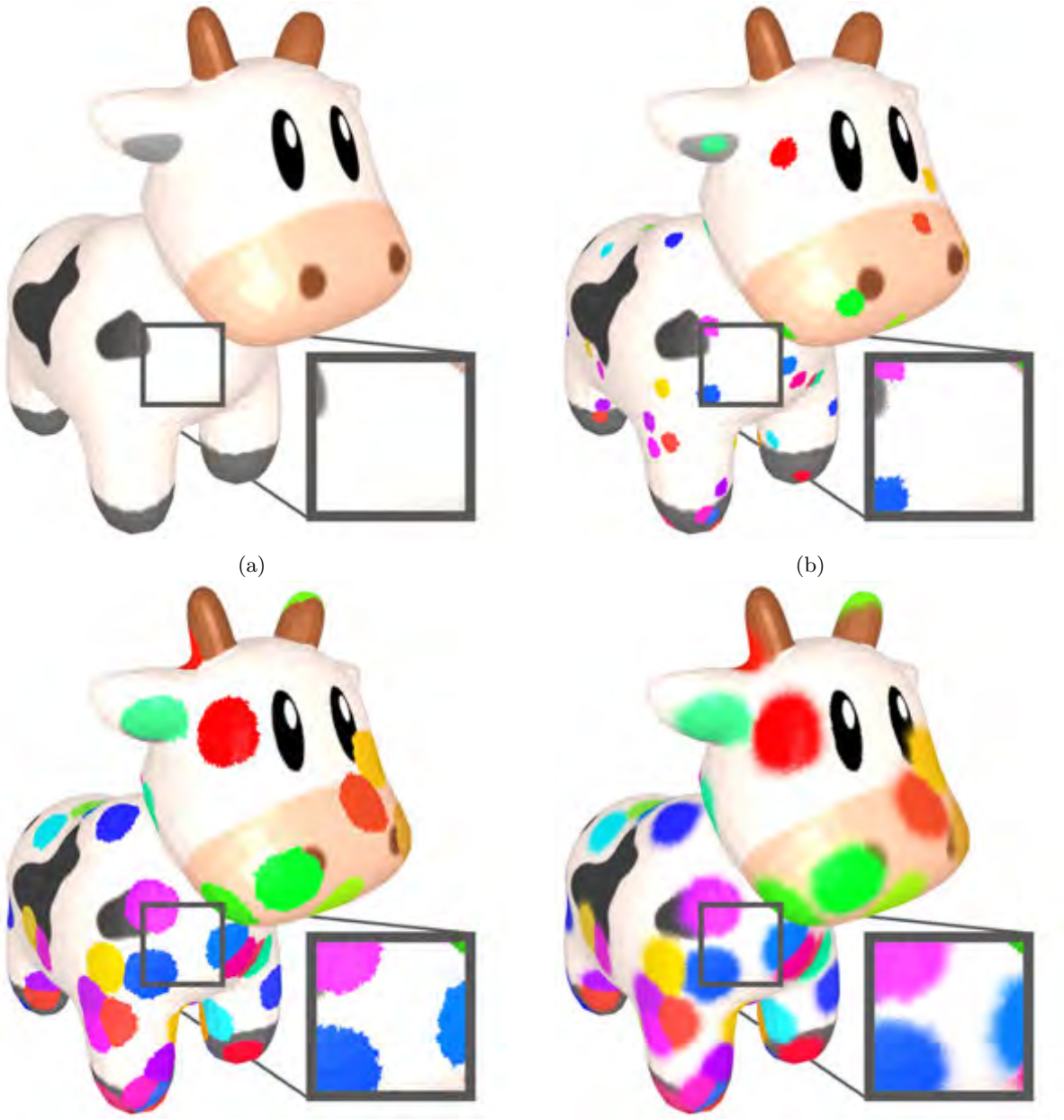


Figure 6.5: *These series of figures show the various stages of a plasma CA on Blender's "Suzanne" model. (a) the original mesh, (b) after initial blue colouring, plasma seed and 10 iterations (c) after 20 iterations, (d) after 35 iterations.*

6.2.6 Water colour CA

The water colour CA is similar to the other colour spreading, with the difference of a random colour for each seed point and the addition of a blur effect. To demonstrate the CA we use Crane's "Spot" model at a grid resolution of [379×678×689] voxels, shown in Figure 6.6. To start, a random placement of pastel-colours are chosen as seed points, chosen with the "colourHSV()" symbol, and placed randomly across the surface

of the model. Next, the seeded colour areas are grown across the surface with a set probability of growth. Finally, a blur effect is grown outwards from the seeded areas. The blur is caused by recolouring voxels with a percentage of colour from neighbourhood voxels. This results in a colour that is an average of the colours within a neighbourhood. The blur is create using the grammars "colourCopy(%)" symbol, which creates a new colour from a percentage of two colours.



(a) (b)

(c) (d)

Figure 6.6: *These series of figures show the various stages of a water colour CA on Crane's "Spot" model. (a) the original mesh, (b) after a pastel colour seed placement and 15 iterations (c) after 35 iterations, (d) after 21 iterations of blurring.*

6.2.7 Borg CA

The borg CA is designed to detail a standard cube model into a Borg cube (seen in the movie Star Trek: First Contact [4]). This is simulated through random surface colouring, followed by a green circuitry effect that spreads across the surface.

The random surface colouring CA colours the surface a speckled grey colour using the "colourHSV(+)" symbol. Only the variance is altered from a base black colour. Next, a seed CA places circuit voxels that circuit lines can grow from. The circuitry CA grows circuit lines outwards from the seed points (circuit voxels). The growth spreads according to the surface-oriented "x" and "z" directions. A circuit line can spread to a new voxel if it does not encounter circuit voxels perpendicular to the movement (the opposing "x" or "z" direction). The current spread direction is determined by active voxels surrounding the current seed voxel. The spreading favours the current direction, however branching off can occur at a much lower probability. Once the circuitry has spread across the surface a circuit line expansion CA is executed. This spreads the circuitry effect to any voxel within the neighbourhood of a circuit voxel. This expansion makes the circuit lines more distinct for high-resolution models.

Another optional CA can erode the model along the circuit lines, creating circuit canals. The CA removes circuit voxels and converts voxels below them (the negative "y" surface oriented direction) into circuit voxels. Multiple iterations of this CA can make deeper canals. However, the canals are difficult to see in the final render.

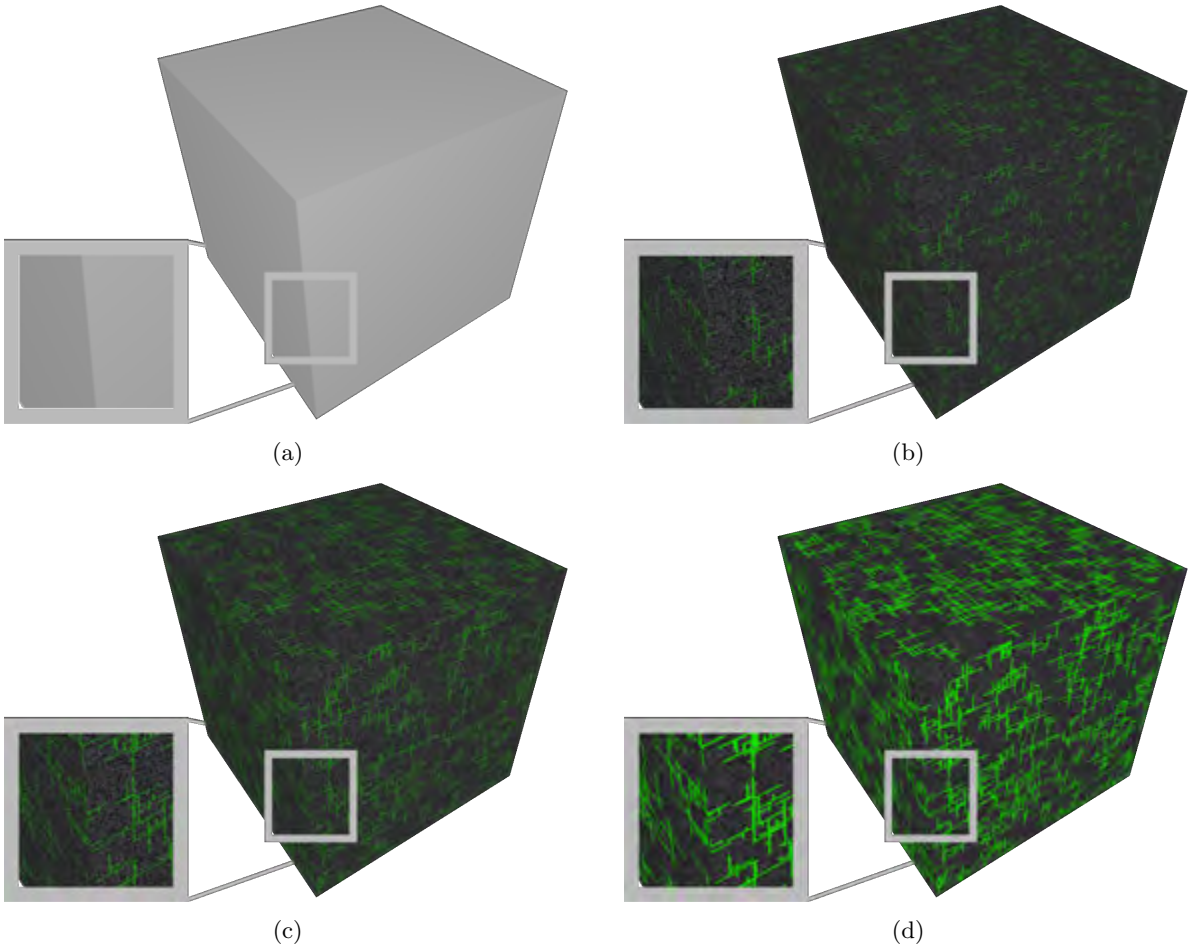
6.2.8 Icicle and snow CA

The icicle CA is a growth CA, similar to the moss CA. The icicles grow straight downwards according to the negative global "y" direction. This simulates a stalactite-like growth created from water droplets that are affected by gravity. Additionally, a snowfall CA is executed on the model to complete the winter effect.

To demonstrate the CA we use Crane's "Pittsburgh Bridge" model. The arched back of the human-form bridge provides a perfect spot for icicle growth. First, a CA places seed icicle voxels in inactive spots randomly below an active voxel. This is further limited to voxels at above a specified height on the "y" axis. Next, the seeded icicle grows downwards at a set probability. At the same time, the icicle grows outwards, in the global "x" and "z" directions, to empty positions below active voxels at a lower probability to the downwards growth. This results in the voxel forming into an upside-down cone-like shape.

Once the icicles have formed to a desired length, a snow CA is executed over the model. This snow CA randomly chooses voxels, with inactive voxel positions above it, to acquire a "snowfallen" state. Using the "colourHSV(+)" symbol, the voxels with a "snowfallen" state are moved towards a white colour by decreasing the saturation of the voxel. Additionally, active voxels surrounding "snowfallen" voxels without a "snowfallen" state randomly acquire the state. This spreads out the snow across the model to create snow build-up effect.

[4] CBS Studios Inc, *Star Trek, Borg Cube*, `http://www.webcitation.org/6d8XJ1fLS`. [Online; accessed 15-October-2014]



(a) (b) (c) (d)

Figure 6.7: *These series of figures show the various stages of a borg cube CA on a cube model. (a) the original mesh, (b) after a random seed CA and 20 iterations of the circuitry CA, (c) after another 40 iterations, (d) 3 iterations of circuit line thickening CA.*

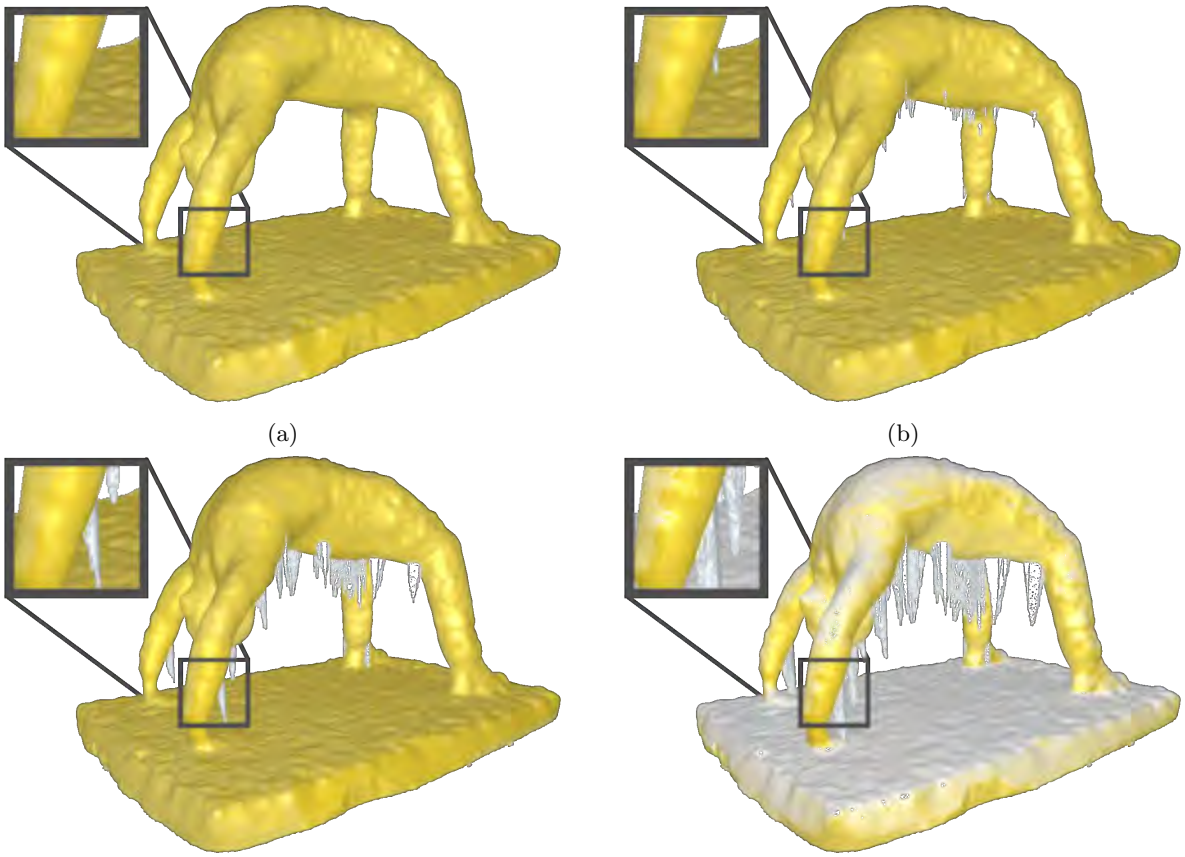


Figure 6.8: *These series of figures show the various stages of an icicle and snow CA on Crane's "Pittsburgh" model. (a) the original mesh, (b) after a random seed icicle placement CA and 20 iterations (c) after 40 iterations, (d) after another 20 iterations and 6 iterations of the snow CA.*

6.3 Performance testing

To measure the performance of the system we look at the three most computation and memory intensive processing stages: volume to mesh, cellular automata execution, and mesh to volume. To ensure our testing is representative of real-world use cases, we measure the performance of these stages across a variety of grid sizes and simultaneous processing threads. The performance tests are broken into two categories. The first is execution time and speed-up due to parallelisation. The second is the random access memory(RAM) usage before, during and after execution. Additionally, we look at the total execution time for each of the CA tested in Section 6.2.

For the performance tests we need to keep a few variables fixed to eliminate unwanted influence in the tests. The first of these constants is to use a single model, a crate model. This model is chosen because it acts as a worst-case scenario for the sparse-voxel octree, where the highest level nodes are completely filled. However, if the narrow band is limited (not fully volumetric) then the data-structure still manages to fill the internal areas sparsely. For consistency we also keep the grid's narrow-band at a constant width of 5 voxels. Finally, we use a stripped-down version of the snow CA C.8 for each of the tests. It does not modify the structure of the grid (the number of voxels) and has no probability. This ensures that the CA is deterministic, and constant for each test.

Next we create two types of tests. In the first, we test three different grid resolutions, 64^3, 128^3, 256^3, and 512^3, while we keep the number of threads constant (at 8 threads). The second test switches the constant, measuring different thread counts against a constant grid resolution of 128^3. The number of threads increase from a single threaded solution, to 2,4 and 8 threads.

To mitigate the effects of other processes running on the computer, such as operating system operations, we repeat each of the tests. We measure each performance statistic 15 times and average the resulting data.

6.3.1 Equipment

These tests are run on a single desktop PC, running the Ubuntu operating system. The specifications for the PC are as follows: a Quad Core Intel i5-3570 CPU at 3.40GHz (without hyper-threading), an NVIDIA GeForce GTX 660 Ti and 12GB of 1333MHz DDR3 RAM. The testing was run by first measuring the time taken to execute the sequential method. This was followed with the parallel OpenMP solution with a variable number of threads. The QTime object provided by qt was used to track the elapsed time for each test case.

To measure the memory usage of the system we regularly read from the Linux file "/proc/[PID]/status", where "[PID]" is the process ID of the executable. The operating system captures memory usage details of processes in this file. However, the results can be inaccurate and slow to update.

6.3.2 CA execution time

Figure 6.9 shows the total execution time for each CA tested in Section 6.2. The performance of each CA is impacted by a number of factors including, the number of rules in each CA, the number of iterations the CA is run and the computational expense of each rule. This is evident from the borg CA result, which is run for the most iterations, as well as having the most rules out of all the CA C.2.

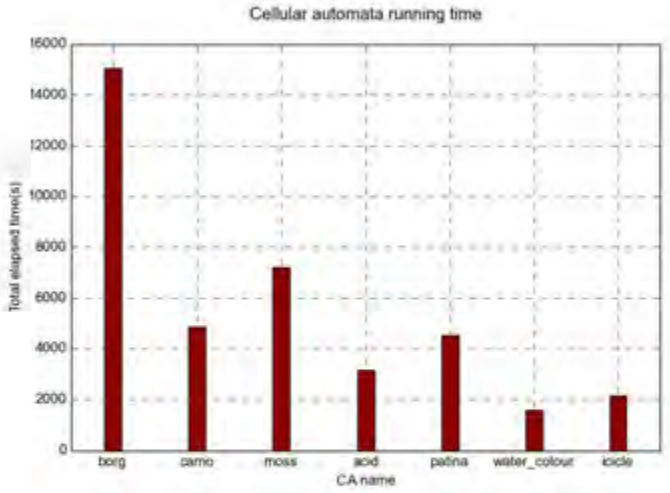


Figure 6.9: *The total exection time of each CA in seconds.*

6.3.3 Processing performance with variable grid resolution

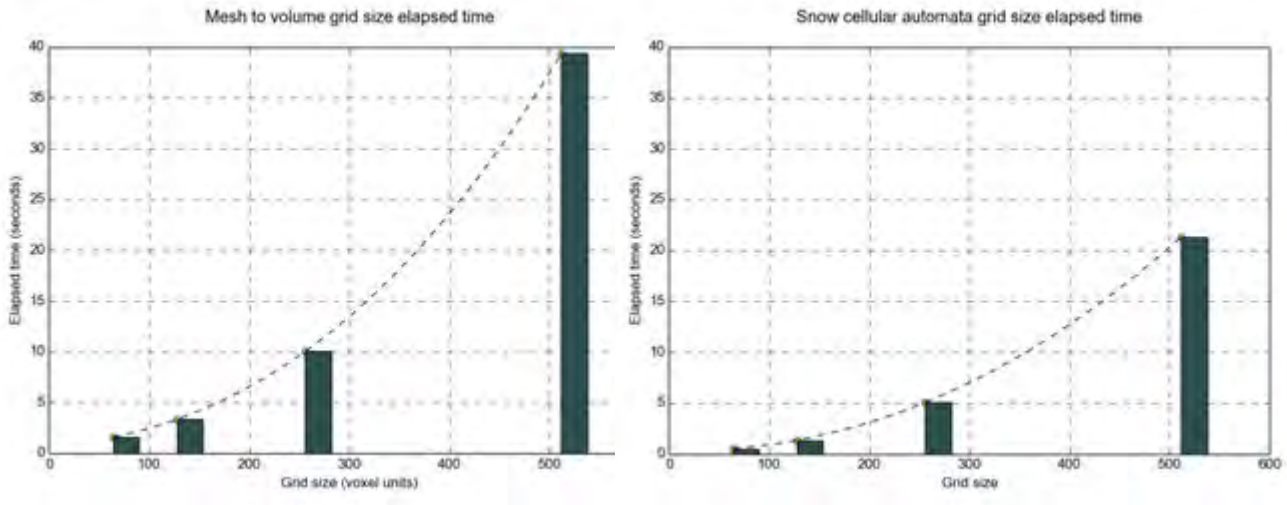
Figure6.10, shows the impact of different grid sizes on the total elapsed processing time. From the graphs we can see that the total elapsed time increase as $O(n^3)$. However, the cellular automata and volume to mesh sections finish roughly twice as fast as the mesh to volume. This is likely due to the mesh to volume OpenVDB tool, which is relatively slower. Another cause could be the grid navigation complexity of each process, where each method uses a different method of grid access. The mesh to volume process uses data-structure accelerated voxel look-ups, the snow CA contains neighbourhood queries and the volume to mesh process ray-casts through the volume.

6.3.4 Processing performance with variable thread count

Next, we examine the processing performance of different numbers of parallel threads at a constant grid size of 128^3. Figure 6.11 shows the elapsed time for each process for different thread counts. To sequentially access the OpenVDB data-structure we use the data-structure iterators provided by the library. The parallel implementation uses Intel's Threading Building Blocks library (TBB), which is also used by OpenVDB in their parallel implementations. However, due to implementation constraints we could not change the maximum number of threads to graph parallel performance. This is due to OpenVDB initialising the thread synchronizer before we can access it. For this reason we also implement a parallel implementation using the OpenMP threading library. The volume to mesh section is the only section where a TBB implementation was not used. This is because we are parallelising iteration over the result mesh polygons, not the voxel data-structure.

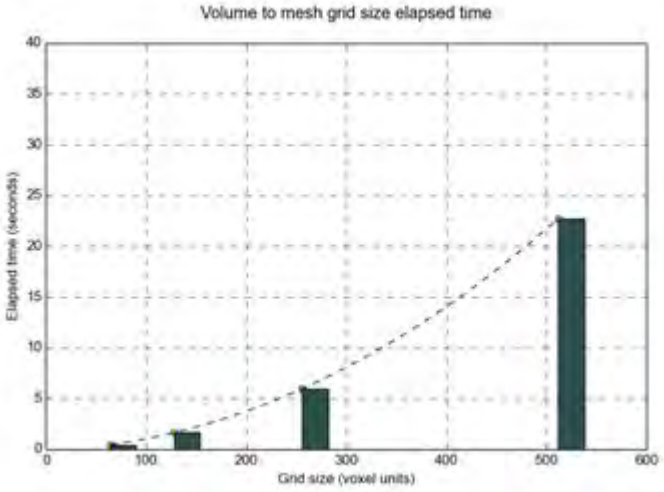
From the graph we can see that the total processing time decreases substantially as the thread count increases, then increases again as the thread count becomes greater than 4. As the testing machine only has 4 cores available, the overhead cost of thread synchronisation could explain the slow down when adding more threads. It is also evident that the TBB implementation completes faster than the OpenMP version with the same amount of threads. Additionally, the OpenMP version of the cellular automata section slows down again after adding more than two threads, and the TBB version is significantly faster for 4 threads.

Another way to visualise this data is by plotting the speed-up and efficiency of the parallelisation, as shown in Figures 6.12 and 6.13, respectively. The speed-up S_n is defined as the proportion of elapsed time for a



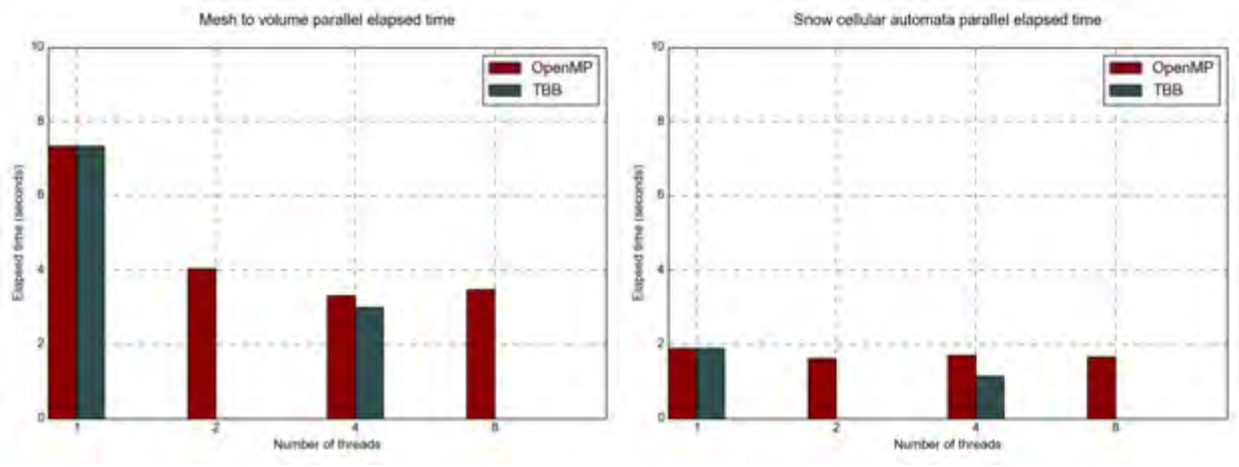
(a)

(b)

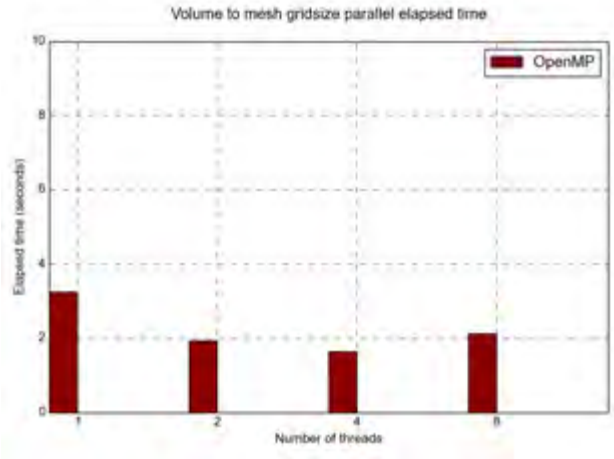


(c)

Figure 6.10: *The total elapsed time of processes at different grid resolutions: (a) mesh to volume, (b) snow CA and (c) volume to mesh. The dashed line shows a cubic regression from the data samples.*

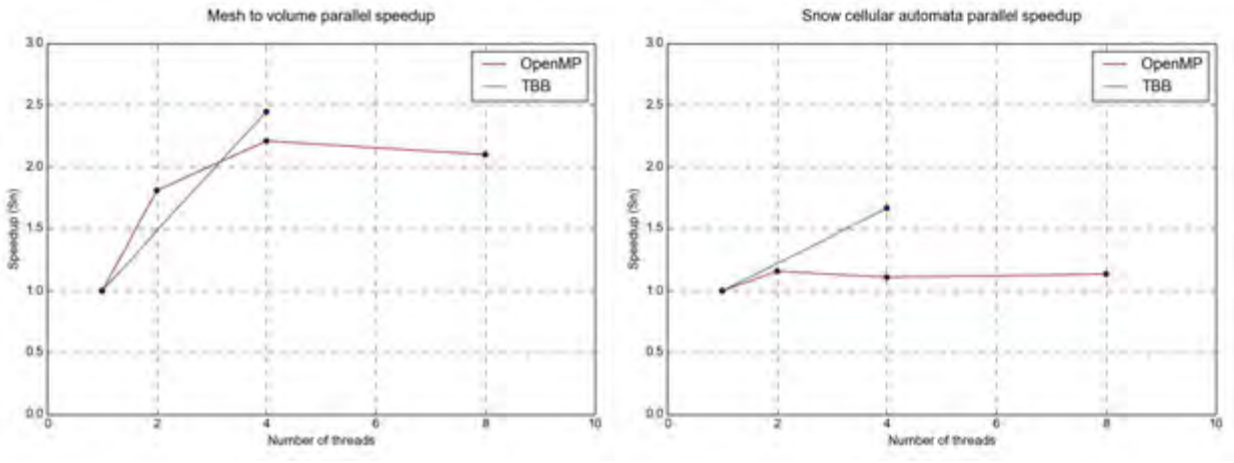


(a) (b)

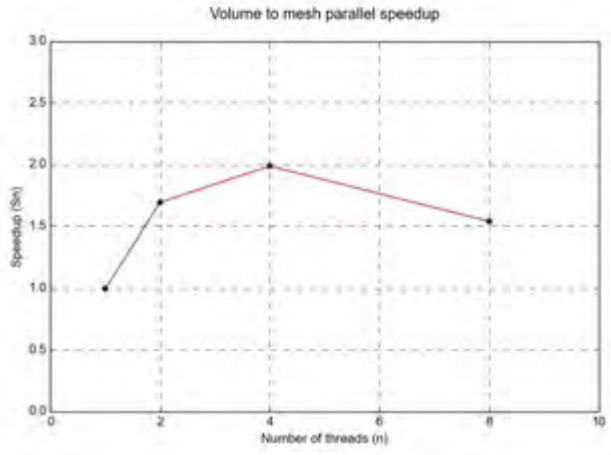


(c)

Figure 6.11: *The total elapsed time of processes at different numbers of parallel threads: (a) mesh to volume, (b) snow CA and (c) volume to mesh.*



(a) (b)



(c)

Figure 6.12: *The relative speed-up of processes at different numbers of parallel threads: (a) mesh to volume, (b) snow CA and (c) volume to mesh.*

thread count of 1 relative to a new count n, $S_n = \frac{t_1}{t_n}$. The speed-up seen in Figure 6.12 follows Amdahl's law, where the overall speed-up from parallelisation is limited by the sequential sections of the process. Additionally, it should be noted that the OpenVDB meshToVolume, and volumeToMesh tools execute at a fixed thread count equal to the number of cores available. This would reduce the overall speed-up seen from the parallelisation of our extensions because the sections execute in the same amount of time regardless of thread count. The thread efficiency seen in Figure 6.13 also reflects this, with a clear linear decline in efficiency as the thread count increases.

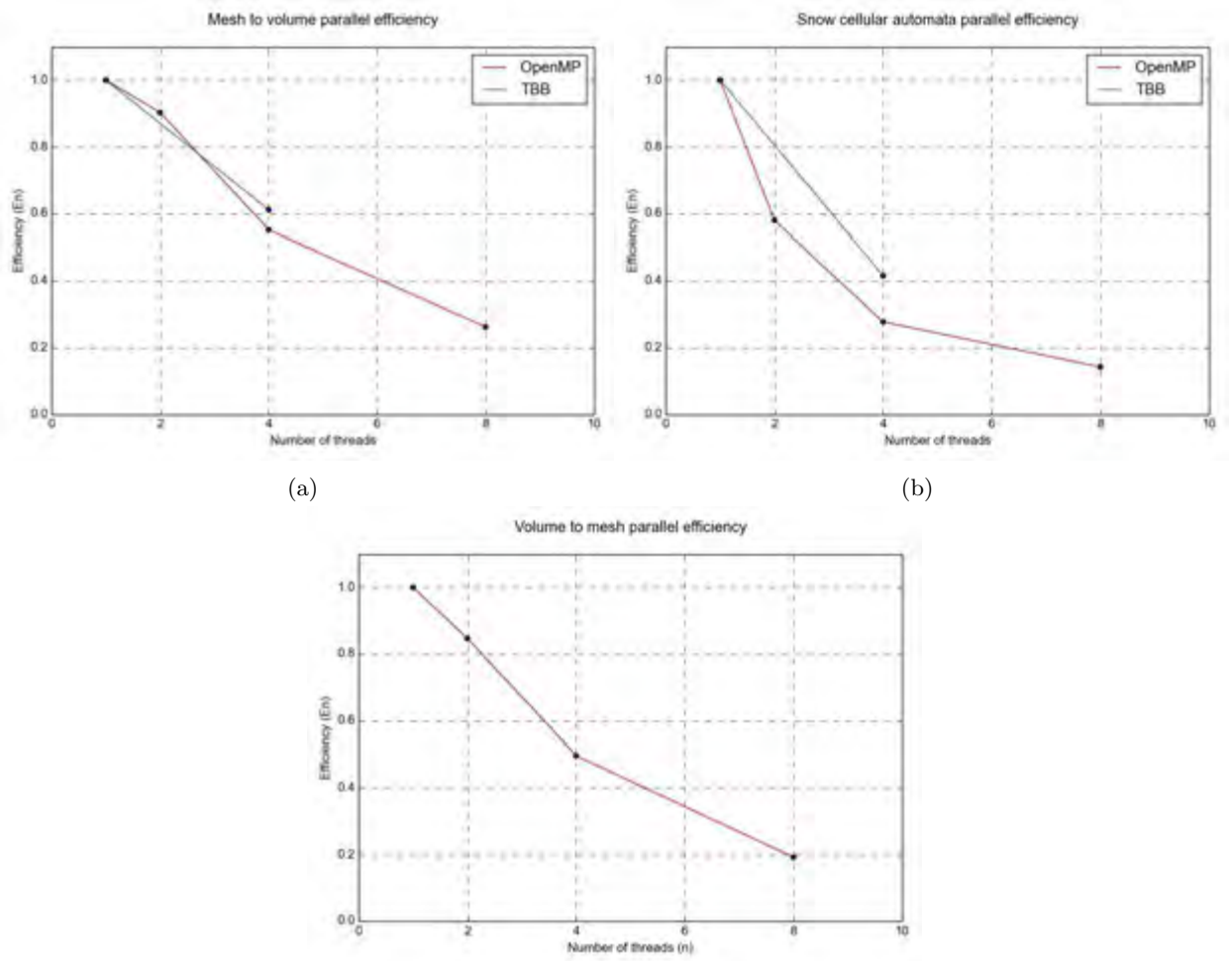


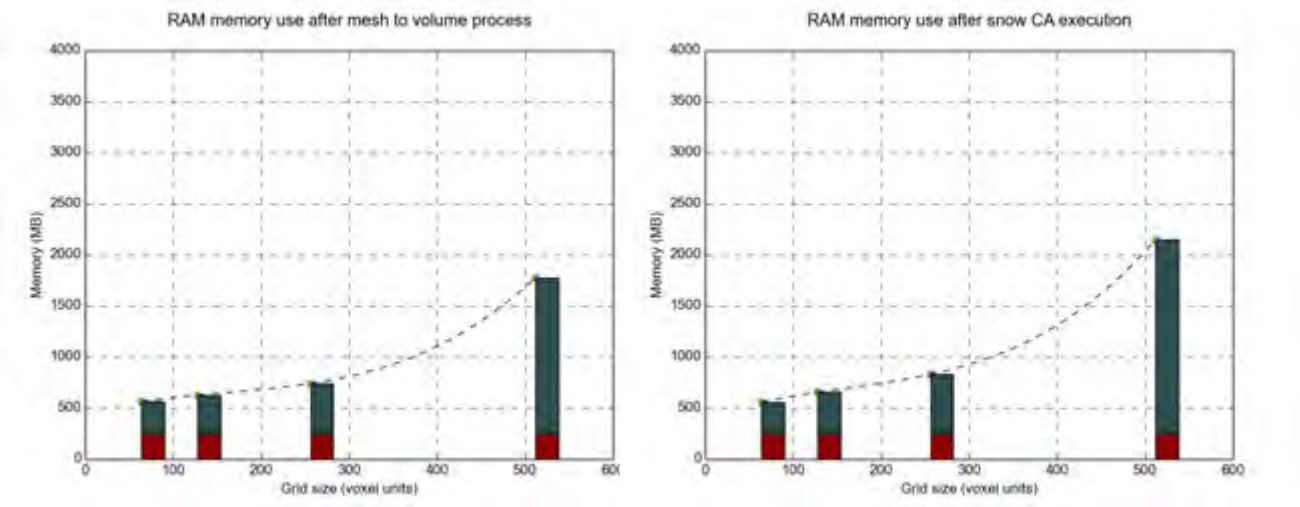
Figure 6.13: *The thread efficiency of processes at different numbers of parallel threads: (a) mesh to volume, (b) snow CA and (c) volume to mesh.*

From the speed-up graphs, we can see that the OpenMP parallelisation of the snow CA is far worse than the TBB version. However, even the TBB implementation does not quite reach a 2 times speedup with 4 threads. This could be caused by the thread synchronisation during the voxel modification area, which could be improved. The synchronisation of the OpenMP version is a naive implementation, where every thread tries execute at each voxel unless blocked by another thread that obtained access to the voxel first. In contrast the parallelisation of the mesh to volume section is more effective. The speed-up increases more consistently, before flattening out (and slightly decreasing) after the thread count exceeds the number of cores. The efficiency also reflects this result showing a slower decrease in efficiency than the other processes.

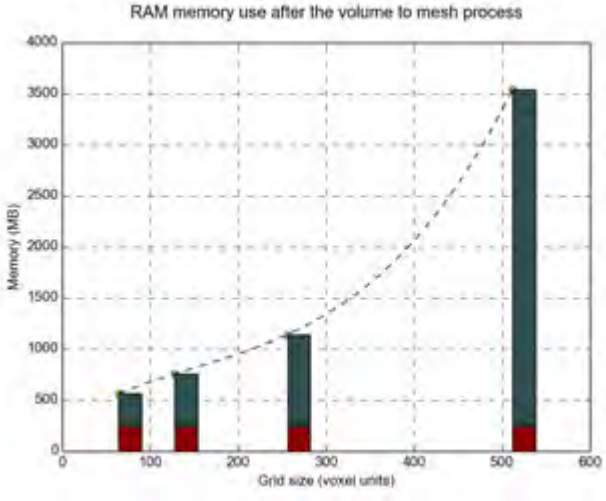
However, the volume to mesh does not benefit quite as well as the mesh to volume process. The speed-up is roughly two-thirds as effective, and falls far more drastically after the thread count exceeds the number of cores.

6.3.5 Memory usage with variable grid resolution

Finally we take a look at the memory usage of the system. The only changing variable that impacts the memory use is the size of the grid. The graphs in Figures 6.14, plot the memory usage for the mesh to volume, snow CA and volume to mesh processes, respectively. The red sections at the base of each bar show the average base memory used by the system before any of the sections have run. From the graphs we can see that the memory usage grows as $O(n^3)$ with the size of the grid, which is naturally expected. It should be noted that both the CA process and volume to mesh processes depend on the mesh to volume section (they need a voxel grid to execute). Thus the memory usage for the two sections include the memory used by the mesh to volume section. From the graphs we can see that the memory use after the mesh to volume process grows significantly in RAM. The snow CA has a little deviation, where the number of states used does not change. Other CA could slowly increase the amount of states used, at worst creating state grids with an active voxel count equalling the colour grid. Lastly, the volume to mesh process adds roughly two-thirds of the memory of the mesh to volume. As we used a relatively thin fixed narrow band (thin relative to high resolutions), the output memory use is very similar to the grid. An output mesh from a fully volumetric grid would use significantly less RAM than the grid. It should be noted that these readings were taken with the snow CA still held in memory.



(a) (b)



(c)

Figure 6.14: *These three figures show memory use of processes at different grid sizes. The processes are as follows: (a) mesh to volume , (b) snow CA and (c) volume to mesh. The dashed line shows a cubic regression from the data samples.*

6.4 Summary

In this chapter we present the results from CA test-case creation/recreation and performance tests. The CA test-cases show how versatile the grammar system is, and how accommodating the voxel engine can be for different model types. By recreating a wide variety of CA present in other work, we show that our grammar system does not hinder the CA creation process. Additionally, the wide variety of model types used, including those of a complex genus, show that the engine can accommodate many model types. However, manual preparation is often needed, such as the elimination of surface inconsistencies and the availability of texture maps.

From the performance tests we found that this voxel-based technique can result in large memory usage and execution times. This makes the techniques more suited to pre-production usage and less applicable to real-time applications. However, with further optimisation and GPU acceleration, this technique could be used in real-time applications on future hardware.

Chapter 7

Conclusion

The goal of this thesis was to develop a system capable of creating and executing 3D cellular automata (CA), to generate surface detail in 3D models. We have successfully shown that it is possible to create surface-detailing CA through a multi-stage approach. The system created can be split into two separate sections, a grammar system to create the CA and a voxel engine to manage the CA and 3D models the CA are executed over.

The grammar system is used to generate 3D cellular automata as an alternative to manual programming. The grammar we use is based on existing CA grammars with extensions inspired by other procedural grammars. Our first extension is 3D neighbourhood cell navigation, which extends the two-dimensional navigation present in other work. This simplifies neighbourhood queries by providing a nested-symbol movement or indexed navigation, depending on what the programmer requires. Additionally, many features such as probability and positional matching are added, as inspired by L-systems and shape grammars.

The CA grammars are created from a collection of rules, which are separated into three sections. The first two sections are built up from symbols called evaluators. The evaluator symbols test whether voxels are a match to criteria specified in the evaluator. If an evaluator does not pass a match then the rule ceases execution for the evaluated voxel (depending on the boolean operators between evaluators). The first section is known as the index-voxel section. When a CA is executed, it is executed once for every voxel, each of which is known as an index voxel. The second section contains evaluators which evaluate only on neighbourhood voxels. These are the voxels in the cells immediately surrounding the index voxel. Movement from index voxels to neighbourhood voxels is achieved through symbols known as navigation symbols. The third section contains modifier symbols. In contrast to evaluators, modifiers change the parameters of the index and neighbourhood voxels. The modifiers will only execute on a voxel if all evaluators have passed, and a correct probability value is found (the default is set to always match).

The voxel engine is divided into four subsections, the model converter, voxel data-structure, cellular automata executor and cellular automata interpreter. The CA execute over voxel data held in the voxel data-structure. This is created by extending the sparse-voxel octree in the OpenVDB library. The model converter creates voxel data by converting polygon-meshes into a voxel representation. This conversion, and corresponding conversions back into a polygon mesh, are facilitated by tools in the OpenVDB library. However, these tools do not retain the colour stored in mesh texture maps. To overcome this limitation we provide extensions. The extensions create coloured voxels and new texture maps from the voxel data.

Before the CA can execute over the voxel data-structure it needs to be converted into an executable form. A grammar parser converts the CA grammars into an XML document of an abstract syntax tree (AST) representation. These XML documents are then read by the voxel grammar and converted into a tree of executable functions that closely resemble the AST. This executable function tree can then apply to

each voxel as a CA. In addition to the navigation symbols in the grammar, neighbourhood rotations were introduced. The rotations orientate the neighbourhood navigation according to the surface normal of the models. This provides a way to create CA that are contextually aware of the surface.

7.1 Findings

Using the system mentioned above, we recreated a number of surface detailing CA present in other work. The CA were created using the CA grammar system, which proved not to hinder the creation process. We measure this success from the large variety of CA recreated. These include acid, moss, camouflage, fracture, patina, plasma, water-colour, borg, and icicle CA. Each of these CA test the system with different behaviours. Three of the most prominent behaviours include: colour modification, erosion and growth.

The first behaviour, which all the CA share, is surface colour manipulation. Colour is manipulated based on a number of parameters including, custom states, current colour and global position. The plasma, camouflage, water colour and patina CA expand on this, using neighbour-hood voxel information. In these CA, the colour is modified according to how colour has expanded across a neighbourhood, as well as according other neighbourhood information. In a similar way to colour expansion, the acid and borg CA erode the surface by removing voxels based on the neighbourhood structure, state and colour. Inversely, the moss and icicle CA create new voxels around areas of interest or "seed" points. The combination of these three behaviours, as well as others, result in surface modifications to models that would not be possible with traditional texture maps. The system tools in this work, such as tangent-space CA execution, proved useful in obtaining these results. In future work we would like to expand this tool-set, as well as introduce new tools such as a cellular network.

In addition to the test-cases above, performance tests were captured. Performance tests show how practical the system is by giving an overview on how resource intensive and time consuming it can be. From execution time we saw that the borg CA is the most time consuming. This is due to both the number of rules in the CA, as well as the number of iterations the CA is executed for. Next, we look at the impact of grid-size on execution time. From the results we saw that the execution time expands with a complexity of $O(n^3)$, as expected in a 3D grid. We also determine that this execution time can be sped up significantly through process thread parallelisation. While our parallelisation results are promising, further parallel optimisation and GPU-based acceleration could be investigate in future work. Finally, by looking at the memory usage at different grid sizes, we see that the system can use a significant amount of memory during execution. While this is a constraint for current hardware, this method can still prove useful for large shared-memory systems and future hardware.

7.1.1 Research questions revisited

The success of our research can be determined by the extent to which the research questions we presented in Section 1.3 have been addressed.

Can voxel based CA be used to create surface detail in 3D models?

From our results we see that using a simple grammar system does not hinder the creation of CA. This is evident from the large range and variety of CA present in the results chapter. A variety of models were used, as well as CA automata types, including colour modifying, structure creating and surface eroding CA.

Can a grammar-based system be used to recreate CA present in other work?

In the results chapter we chose a number of CA to reproduce from other work. While we cannot exhaustively test the variety of CA present, we chose to reproduce enough of a variety that it is reasonable to extrapolate

the capabilities of our system. We feel that the system is versatile enough to create most 3D CA in other work.

Does our system produce surface detail that is difficult or impossible to produce with mesh-based models?

The structure-creating and surface-eroding CA present in our system are very difficult to reproduce with normal texture maps. This is especially evident in the acid and moss CA, where concave structures often result after execution.

It was also shown that with current hardware, voxel-based CA can be used effectively to create surface detail. However, due to the computation and memory requirements of the system, this voxel-based approach is more suited to off-line work than real-time processing. Real-time processing could be used for detail in 3D models, however, the low resolution limitations would constrain the adoption of this method. However, with the rise in popularity of low resolution pixel and voxel art as a stylistic choice, this could still find a target market among artists and independent game developers.

7.2 Limitations and future work

There are many areas for future work in our grammar-based 3D cellular automata system. Many of these areas arise from the limitations of our system. While the grammar we developed has been shown not to hinder the creation of 3D CA, future work is required to determine if grammars are simpler or faster than manual programming. In addition, a visual approach to generating the grammars (or AST) could make the process more intuitive and accessible to non-technical developers.

To draw conclusions on whether the system can assist human graphic designers, further tests should be conducted. These tests should include user testing, which compares the results of this system to similar solutions and how fast or effective the system is as a development tool to others. Additionally, usability tests should be conducted between visual, grammar and standard programming methods of developing CA.

While the range of CA tests have shown how versatile the grammar system is, further testing is needed to compare the results of the system to other solutions. This includes the quality and range of produced detail, as well as performance tests in terms of execution time and memory usage.

The neighbourhood orientation system could be expanded into a more agile and robust system. The orientation system is particularly dependent on the initial normal, tangent and bi-tangent frames created from the input texture maps. Additionally, when a new voxel is added to the dataset it receives tangent space vectors from its neighbourhood. For large scale structural changes this method can introduce significant errors. A system similar to the cellular network system presented by Gobron et al. [21] could provide improved adaptability and robustness.

Another limitation is the creation of texture maps for the systems output mesh. Our näive pixel-per-polygon technique could be replaced by a more sophisticated polygon-interpolated texture atlas method, such as the least squares conformal maps technique at Lèvy etal. [35].

As mentioned above, due to performance and memory constraints the system is not suited to real-time processing. However, other work has proven that CA are naturally suited to acceleration with graphics hardware. Future work could investigate the GPU acceleration of 3D cellular automata, and what resolution is required for real-time applications on current hardware. It should also be noted that for the resolutions used in our testing section, it would be difficult to fit the whole grid into GPU memory. However, this could be overcome with the next generation of hardware or by splitting the CA execution over different sections of the grid in multiple GPU passes.

Finally, the output mesh provided by OpenVDB is at a relatively high resolution. The resolution provided is generally acceptable for offline processing methods, such as those used in movie special effects. However, the mesh would need a mesh simplification pass before it can be used in real-time applications on current hardware. This could be achieved through modelling packages such as Blender[1] or Meshlab[2].

[1] Blender.org. *Blender*, `http://www.webcitation.org/6aAHxPicj`, [Online; accessed 20-July-2015]
[2] Visual Computing Lab. *Meshlab*, `http://www.webcitation.org/6ckDbhOVP`, [Online; accessed 02-November-2015]

Appendices

Appendix A

Cellular automata grammar symbol definitions

A.1 Rule structure symbols

$\rightarrow$

This symbol defines the left and right hand side of the grammar rule, or production. Before the symbol lies the "head", which contains evaluator symbols. After the symbol lies the "body", which contains modifier symbols.

\#

The "hash" operator describes the end of the index voxel evaluator section (the left of the symbol), and the start of the neighbourhood evaluator symbols (the right).

%

The operator, "!", enables the use of the standard "not" bitwise operator.

[0.5]

A number in-between the square brackets "[]" describe a probability. The probability is a floating point number that lies between 0.0 and 1. The square brackets may only appear after all of the modifiers in a production, and this indicates a probability that the entire production will pass.

$a(0.5)$

A string-symbol followed by parentheses, indicates a modifier/evaluator symbols. Numbers in-between parentheses describe the parameters of a symbol.

&

This symbol represents "and" operations, which enable multiple evaluators to be used simultaneously. If any of the evaluators are found to be false then the production rule (or logic set) is false. This symbol appears only on the left hand side of the production rule.

|

Similar to the "and" operator, this "or" operator enables multiple state evaluators to be used simultaneously. If any of the evaluators are found to be true then the production rule (or logic set) is true.

A.2 Complementary evaluation and setter symbols

norm(x, y, z, tx, ty, tz)

Evaluates/sets the normal value of the voxel, with or without tolerance or variation.

sdf(x, tolerance)

Evaluates/sets the sdf value of the voxel.

colour(r, g, b, tx, ty, tb)

Evaluates the colour value of the voxel or sets the colour value to a particular colour, with or without tolerance or variation.

colourR(r, tolerance)

Evaluates/sets the red colour channel of the voxel, with or without tolerance or variation.

colourG(g, tolerance)

Evaluates/sets the green colour channel of the voxel, with or without tolerance or variation.

colourB(b, tolerance)

Evaluates/sets the blue colour channel of the voxel, with or without tolerance or variation.

colourHSV(h, s, v, th, ts, tv)

Evaluates the colour value of the voxel or sets the colour value to a particular colour, with or without tolerance or variation.

colourH(h, tolerance)

Evaluates/sets the hue of a voxel's colour, with or without tolerance or variation.

colourS(s, tolerance)

Evaluates/sets the saturation of a voxel's colour, with or without tolerance or variation.

colourV(v, tolerance)

Evaluates/sets the variation of a voxel's colour, with or without tolerance or variation.

intensity(i, tolerance)

Evaluates the intensity of the colour value of the voxel and sets the colour to a given grey-scale colour.

on **and** *off*

Evaluates the on/off state of the voxel and sets the state.

A.3 Evaluation only symbols

State only symbols

A.4 Relative neighbourhood movement symbols

-x$\{ab\}$

Movement in the negative x axis direction.

+x$\{ab\}$

Movement in the positive x axis direction.

-y$\{ab\}$

Movement in the negative y axis direction.

+y$\{ab\}$

Movement in the positive y axis direction.

-z$\{ab\}$

Movement in the negative z axis direction.

+z$\{ab\}$

Movement in the positive z axis direction.

cube(**length**)

The cube symbol describes a cubic volume to be evaluated according to the length of the cube and its rotation.

sphere(**radius**)

The sphere symbol describes a spherical volume to be evaluated according to the radius of the sphere and its rotation.

Appendix B

Software

The software used to develop this project is availiable at the following link:
`https://bitbucket.org/creative630/proceduralvoxels`

The software is released under an open-source license, which is available in the code respository above.

Appendix C

Cellular automata grammars

C.1 Acid CA

Listing C.1: Acid seed placement grammar

```
state redColour

Rule: #
+y{off}
-> state(redColour,1) colour(1,0,0) [0.0003];
```

Listing C.2: Acid propagation grammar

```
state redColour
state turningoff

checkRed:
+x{redColour == 1}
-x{redColour == 1}
+z{redColour == 1}
-z{redColour == 1}
+y{redColour == 1}
-y{redColour == 1}
;
checkRedOr:
+x{redColour == 1 || off}
+x{+x{redColour == 1 || off}}
-x{redColour == 1 || off}
-x{-x{redColour == 1 || off}}
+z{redColour == 1 || off}
+z{+z{redColour == 1 || off}}
-z{redColour == 1 || off}
-z{-z{redColour == 1 || off}}
+y{redColour == 1 || off}
+y{+y{redColour == 1 || off}}
-y{redColour == 1 || off}
-y{-y{redColour == 1 || off}}
```

```
;

Rule: !redColour == 1 #
+x{redColour == 1}
-> state(redColour,1) colour(1,0,0) [0.5];

Rule: !redColour == 1 #
-x{redColour == 1}
-> state(redColour,1) colour(1,0,0) [0.5];

Rule: !redColour == 1 #
+z{redColour == 1}
-> state(redColour,1) colour(1,0,0) [0.5];

Rule: !redColour == 1 #
-z{redColour == 1}
-> state(redColour,1) colour(1,0,0) [0.5];

Rule: !redColour == 1 #
+y{redColour == 1}
-> state(redColour,1) colour(1,0,0) [0.5];

Rule: !redColour == 1 #
-y{redColour == 1}
-> state(redColour,1) colour(1,0,0) [0.5];


Rule: redColour == 1 #
sphere(full,and,3, redColour == 1)
-> state(turningoff,1);

Rule: turningoff==1 #
-> off;
```

C.2 Borg CA

Listing C.3: Random dark colouring

```
Rule: #
-> colourHSV(0.6,0,0.0,  0.6,0.1,0.4);
```

Listing C.4: Green grid lines seed placement grammar

```
up: +y{off} ;
Rule: !state == 1 #
up()
-> colour(0.1,0.8,0) state(state,1) [0.0005];
```

Listing C.5: Green grid lines propagation grammar

```
up: +y{off} ;
upup: +y{up()} ;
```

```
checkz:
+z{!state == 2}
-z{!state == 2}
;

checkx:
+x{!state == 2}
-x{!state == 2}
;

Rule: !state == 2 #
up()
+x{ state==1 !state == 2}
-> colour(0.1,0.8,0) state(2) [0.7];

Rule: !state == 2 #
up()
-x{ state==1 !state == 2}
-> colour(0.1,0.8,0) state(2) [0.7];

Rule: !state == 2 #
up()
+z{ state==1 !state == 2}
-> colour(0.1,0.8,0) state(2) [0.7];

Rule: !state == 2 #
up()
-z{ state==1 !state == 2}
-> colour(0.1,0.8,0) state(2) [0.7];

Rule: state==1 #
-> colour(0.1,0.8,0) state(2) [0.6];

Rule: !state == 2 #
up()
+x{ state == 2}
checkz()
-> colour(0.1,0.8,0) state(2) [0.001];

Rule: !state == 2 #
up()
-x{ state == 2}
checkz()
-> colour(0.1,0.8,0) state(2) [0.001];

Rule: !state == 2 #
up()
+z{ state == 2}
checkx()
-> colour(0.1,0.8,0) state(2) [0.001];
```

```
Rule: !state == 2 #
up()
-z{ state == 2}
checkx()
-> colour(0.1,0.8,0) state(2) [0.001];


Rule: !state == 2 #
up()
+x{ state == 2}
+x{+x{ state == 2}}
+x{
checkz()
}
-> colour(0.1,0.8,0) state(2) [0.7];

Rule: !state == 2 #
up()
-x{ state == 2}
-x{-x{ state == 2}}
-x{
checkz()
}
-> colour(0.1,0.8,0) state(2) [0.7];

Rule: !state == 2 #
up()
+z{ state == 2}
+z{+z{ state == 2}}
+z{
checkx()
}
-> colour(0.1,0.8,0) state(2) [0.7];

Rule: !state == 2 #
up()
-z{ state == 2}
-z{-z{ state == 2}}
-z{
checkx()
}
-> colour(0.1,0.8,0) state(2) [0.7];
```

Listing C.6: Green line thickening

```
state erosion

Rule: ! state == 1 ! state == 2#
+x{state == 1}
-> state(1) colourCopy(1,0,0);

Rule: ! state == 1 ! state == 2#
```

```
-x{state == 1}
-> state(1) colourCopy(-1,0,0);


Rule: ! state == 1 ! state == 2#
+z{state == 1}
-> state(1) colourCopy(0,0,1);

Rule: ! state == 1 ! state == 2#
-z{state == 1}
-> state(1) colourCopy(0,0,-1);


Rule: ! state == 1 ! state == 2#
+x{state == 2}
-> state(2) colourCopy(1,0,0);

Rule: ! state == 1 ! state == 2#
-x{state == 2}
-> state(2) colourCopy(-1,0,0);


Rule: ! state == 1 ! state == 2#
+z{state == 2}
-> state(2) colourCopy(0,0,1);

Rule: ! state == 1 ! state == 2#
-z{state == 2}
-> state(2) colourCopy(0,0,-1);
```

Listing C.7: Green line trench erosion

```
state erosion

Rule: ! state == 1 ! state == 2#
+y{state == 1}
-> colourCopy(0,1,0) state(erosion, 1);

Rule: ! state == 1 ! state == 2#
+y{state == 2}
-> colourCopy(0,1,0) state(erosion, 1);

Rule: state == 1 #
-> off;

Rule: state == 2 #
-> off;


Rule: !erosion == 1#
+y{erosion == 1}
-> colourCopy(0,1,0) state(1);
```

```
Rule: erosion == 1 #
-> off;
```

C.3 Camouflage CA

Listing C.8: Camouflage seed placement grammar

```
state cammo

up: +y{off} ;
checkCammo:
!cammo == 1 !cammo == 2 !cammo == 3
;

Rule: #
-> colour(0.83203125,0.81640625,0.5390625);

Rule: checkCammo() #
up()
-> colour(0.55859375,0.578125,0.359375) state(cammo,1) [0.0005];

Rule: checkCammo() #
up()
-> colour(0.32421875,0.36328125,0.26953125) state(cammo,2) [0.0005];

Rule: checkCammo() #
up()
-> colour(0.01171875,0.04296875,0.0078125) state(cammo,3) [0.0005];
```

Listing C.9: Camouflage propagation grammar

```
state cammo

checkCammo:
!cammo == 1 !cammo == 2 !cammo == 3
;

Rule: checkCammo() #
sphere(full,or, 3, cammo == 1)
-> state(cammo,1) colour(0.55859375,0.578125,0.359375) ;

Rule: checkCammo() #

sphere(full,or, 3, cammo == 2)

-> state(cammo,2) colour(0.32421875,0.36328125,0.26953125);

Rule: checkCammo() #

sphere(full,or, 3, cammo == 3)
```

```
-> state(cammo,3) colour(0.01171875,0.04296875,0.0078125);
```

C.4 Fracture CA

Listing C.10: Fracture colouring grammar

```
state crack

Rule: #
+x{on}
!colourDiffHSV(1,0,0, 0.01,0.1,0.1)
-> state(crack,1);

Rule: #
-x{on}
!colourDiffHSV(-1,0,0, 0.01,0.1,0.1)
-> state(crack,1);

Rule: #
+z{on}
!colourDiffHSV(0,0,1, 0.01,0.1,0.1)
-> state(crack,1);

Rule: #
-z{on}
!colourDiffHSV(0,0,-1, 0.01,0.1,0.1)
-> state(crack,1);

Rule: #
+y{on}
!colourDiffHSV(0,1,0, 0.01,0.1,0.1)
-> state(crack,1);

Rule: #
-y{on}
!colourDiffHSV(0,-1,0, 0.01,0.1,0.1)
-> state(crack,1);


Rule: crack==1 #
-> colour(1,1,1);

Rule: ! crack==1 #
-> colour(0,0,0);
```

C.5 Moss CA

Listing C.11: Moss seed placement grammar

```
state moss
```

```
Rule: !moss==1 #
+y{off}
-> +y{colourHSV(0.29,0.9,0.2  ,0.31,1,0.4) state(moss,1) } [0.0001];
```

Listing C.12: Moss propagation grammar

```
state moss

Rule: moss == 1#
+x{
off
-y{on}
}
-> +x{colourHSV(0.25,0.9,0.2  ,0.32,1,1) state(moss,1)} [0.4];

Rule: moss == 1#
-x{
off
-y{on}
}
-> -x{colourHSV(0.25,0.9,0.2  ,0.32,1,1) state(moss,1)} [0.4];

Rule: moss == 1#
+z{
off
-y{on}
}
-> +z{colourHSV(0.25,0.9,0.2  ,0.32,1,1) state(moss,1)} [0.4];

Rule: moss == 1#
-z{
off
-y{on}
}
-> -z{colourHSV(0.25,0.9,0.2  ,0.32,1,1) state(moss,1)} [0.4];

Rule: moss == 1#
-y{off}
-> -y{colourHSV(0.25,0.9,0.2  ,0.32,1,1) state(moss,1)} [0.4];

Rule: moss == 1#
+y{off}
-y{-y{-y{-y{!moss == 1}}}}
-> +y{colourHSV(0.25,0.9,0.2  ,0.32,1,1) state(moss,1)} [0.01];
```

C.6 Patina CA

Listing C.13: Speckled copper colouring and patina seed placement grammar

```
state patina

up: +y{off} ;
```

```
Rule: #
-> colour(0.71875,0.44921875,0.19921875);


Rule: #
-> colourHSV(+,0,0,0    ,0,0.1,0.1);


Rule:#
up()
-> state(patina,1) colour(%, 0.0,1,0.6, 0.1,0.1,0.1) [0.00005];
```

Listing C.14: Patina propagation grammar

```
state plasma

up: +y{off} ;

Rule: !plasma == 1#
sphere(full,or, 2, plasma == 1)
-> state(plasma,1) colour(0.8,0.8,0.1) [0.5];

Rule: plasma == 1#
-> colour(-,0.00,0.02,0.005);
```

C.7 Plasma CA

Listing C.15: Plasma seed placement grammar

```
state plasma

up: +y{off} ;
checkState:
!plasma == 1
;

Rule: #
-> colour(0.1,0.1,0.8);


Rule: checkState() #
up()
-> state(plasma,1) colour(0.8,0.8,0.1) [0.0005];
```

Listing C.16: Plasma propagation and ageing grammar

```
state plasma

up: +y{off} ;

Rule: !plasma == 1#
sphere(full,or, 2, plasma == 1)
-> state(plasma,1) colour(0.8,0.8,0.1) [0.5];
```

```
Rule: plasma == 1#
-> colour(-,0.00,0.02,0.005);
```

C.8 Snow CA

Listing C.17: Snowfall grammar

```
state snowFallen
Rule: #
        +y{off}
        +y{+y{off}}
        +y{+y{+y{off}}}
        +y{+y{+y{+y{off}}}}
-> colour(1.0,1.0,1.0) state(snowFallen,1);
```

C.9 Stalactite CA

Listing C.18: Stalactite seed placement grammar

```
state stalactite

Rule: stalactite == 1 posY > 0#
-y{off}
-> -y{colourHSV(0.53,0.0,0.9, 0.59,0.1,1.0) state(stalactite,1)} [0.9];

Rule: stalactite == 1 posY > 0#
-x{off +y{ on }}
-> -x{colourHSV(0.53,0.0,0.9, 0.59,0.1,1.0) state(stalactite,1)} [0.1];

Rule: stalactite == 1 posY > 0#
+x{off +y{ on }}
-> +x{colourHSV(0.53,0.0,0.9, 0.59,0.1,1.0) state(stalactite,1)} [0.1];

Rule: stalactite == 1 posY > 0#
+z{off +y{ on }}
-> +z{colourHSV(0.53,0.0,0.9, 0.59,0.1,1.0) state(stalactite,1)} [0.1];

Rule: stalactite == 1 posY > 0#
-z{off +y{ on }}
-> -z{colourHSV(0.53,0.0,0.9, 0.59,0.1,1.0) state(stalactite,1)} [0.1];
```

Listing C.19: Stalactite grammar

```
state stalactite

Rule: !stalactite == 1 posY > 0#
-y{off}
-> -y{colourHSV(0.53,0.0,0.9, 0.59,0.1,1.0) state(stalactite,1) } [0.0005];
```

C.10 Water colour CA

Listing C.20: Water colour seed placement grammar

```
state seed

Rule: !seed == 1 #
+y{off}
-> state(seed,1)
colourHSV(0.0,0.7,0.9 , 1.0,1.0,1.0) [0.0001];
```

Listing C.21: Water colour propagation grammar

```
state seed

Rule: !seed==1 #
+x{seed==1 }
-> state(seed,1) colourCopy(1,0,0) [0.5];

Rule: !seed==1 #
-x{seed==1 }
-> state(seed,1) colourCopy(-1,0,0) [0.5];

Rule: !seed==1 #
+z{seed==1 }
-> state(seed,1) colourCopy(0,0,1) [0.5];

Rule: !seed==1 #
-z{seed==1 }
-> state(seed,1) colourCopy(0,0,-1) [0.5];

Rule: !seed==1 #
+y{seed==1 }
-> state(seed,1) colourCopy(0,1,0) [0.5];

Rule: !seed==1 #
-y{seed==1 }
-> state(seed,1) colourCopy(0,-1,0) [0.5];

/*
Rule: !seed==1 #
cube(full, or, 1, seed==1 )
colourDiff(0,-1,0 ,0.1,0.1,0.1)
-> state(seed,1);

Rule: #
+x{seed==1}
->colourCopy(%,1,0,0 ,0.8,0.8,0.8);

Rule: #
-x{seed==1}
->colourCopy(%,-1,0,0 ,0.8,0.8,0.8);

Rule: #
+z{seed==1}
```

```
-> colourCopy(%,0,0,1  ,0.8,0.8,0.8);

Rule: #
-z{seed==1}
->colourCopy(%,0,0,-1  ,0.8,0.8,0.8);

Rule: #
+y{seed==1}
-> colourCopy(%,0,1,0  ,0.8,0.8,0.8);

Rule: #
-y{seed==1}
->colourCopy(%,0,-1,0  ,0.8,0.8,0.8);

*/
```

Listing C.22: Water colour blur grammar

```
state seed

Rule: #
+x{seed==1}
-> state(seed,1) colourCopy(%,1,0,0  ,0.9,0.9,0.9) [0.7];

Rule: #
-x{seed==1}
-> state(seed,1) colourCopy(%,-1,0,0  ,0.9,0.9,0.9) [0.7];

Rule: #
+z{seed==1}
-> state(seed,1) colourCopy(%,0,0,1  ,0.9,0.9,0.9) [0.7];

Rule: #
-z{seed==1}
-> state(seed,1) colourCopy(%,0,0,-1  ,0.9,0.9,0.9) [0.7];

Rule: #
+y{seed==1}
-> state(seed,1) colourCopy(%,0,1,0  ,0.9,0.9,0.9) [0.7];

Rule: #
-y{seed==1}
-> state(seed,1) colourCopy(%,0,-1,0  ,0.9,0.9,0.9) [0.7];
```

Bibliography

[1] Ortega Alfonseca. Representation of some cellular automata by means of equivalent l systems. *Complexity International*, 7:1–16, 2000.

[2] David Andre, Forrest H. Bennett, III, and John R. Koza. Discovery by genetic programming of a cellular automata rule that is better than any known rule for the majority classification problem. In *Proceedings of the 1st Annual Conference on Genetic Programming*, pages 3–11, Cambridge, MA, USA, 1996. MIT Press.

[3] H. Arata, Y. Takai, N.K. Takai, and T. Yamamoto. Free-form shape modeling by 3d cellular automata. In *Shape Modeling and Applications, 1999. Proceedings. Shape Modeling International '99. International Conference on*, pages 242–247, Mar 1999.

[4] Pravin Bhat, Stephen Ingram, and Greg Turk. Geometric texture synthesis by example. In *Proceedings of the 2004 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, SGP '04, pages 41–44, New York, NY, USA, 2004. ACM.

[5] Martin Bokeloh, Michael Wand, and Hans-Peter Seidel. A connection between partial symmetry and inverse procedural modeling. *ACM Trans. Graph.*, 29(4):104:1–104:10, July 2010.

[6] David E. Breen, Sean Mauch, and Ross T. Whitaker. 3d scan conversion of csg models into distance volumes. In *Proceedings of the 1998 IEEE Symposium on Volume Visualization*, VVS '98, pages 7–14, New York, NY, USA, 1998. ACM.

[7] Ron Breukelaar and Th Bäck. Using a genetic algorithm to evolve behavior in multi dimensional cellular automata: emergence of behavior. In *Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 107–114. ACM, 2005.

[8] John W. Buchanan. Simulating Wood Using a Voxel Approach. *Computer Graphics Forum*, 17(3):105–112, August 1998.

[9] B. Chang, D. Cha, and I. Ihm. Computing local signed distance fields for large polygonal models. *Computer Graphics Forum*, 27(3):799–806, 2008.

[10] Arturo Chavoya and Yves Duthen. Using a genetic algorithm to evolve cellular automata for 2d/3d computational development. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, GECCO '06, pages 231–232, New York, NY, USA, 2006. ACM.

[11] Noam Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2):137–167, 1959.

[12] Noam Chomsky. *Syntactic structures*. Mouton de Gruyter, 2nd edition, 2002.

[13] Zacharia Crumley, James Gain, and Patrick Marais. Voxel-Space Shape Grammars. In *WSCG 2012: 20th International Conference on Computer Graphics*, pages 1–10, Plzen, Czech Republic, 25-28 June 2012. Visualization and Computer Vision 2012.

[14] David S. Ebert, Steven Worley, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Kenton F. Musgrave. *Texturing and Modeling*. Academic Press, Inc., Orlando, FL, USA, 2nd edition, 1998.

[15] Alexei A. Efros and William T. Freeman. Image quilting for texture synthesis and transfer. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pages 341–346, New York, NY, USA, 2001. ACM.

[16] Peter Eichhorst and Walter J Savitch. Growth functions of stochastic Lindenmayer systems. *Information and Control*, 45(3):217–228, 1980.

[17] N Ferrando, M A Gosálvez, J Cerdá, R Gadea, and K Sato. Octree-based, GPU implementation of a continuous cellular automaton for the simulation of complex, evolving surfaces. *Computer Physics Communications*, 182(3):628–640, 2011.

[18] Michael S. Floater and Kai Hormann. Surface parameterization: a tutorial and survey. In Neil A. Dodgson, Michael S. Floater, and Malcolm A. Sabin, editors, *Advances in Multiresolution for Geometric Modelling*, Mathematics and Visualization, pages 157–186. Springer Berlin Heidelberg, 2005.

[19] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 209–216, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[20] Stéphane Gobron and Norishige Chiba. Visual simulation of crack pattern based on 3d surface cellular automaton. In *Proceedings of the Seventh International Conference on Parallel and Distributed Systems: Workshops*, ICPADS '00, pages 181–, Washington, DC, USA, 2000. IEEE Computer Society.

[21] Stéphane Gobron, Denis Finck, Philippe Even, and Bertrand Kerautret. Merging cellular automata for simulating surface effects. In *Proceedings of the 7th International Conference on Cellular Automata for Research and Industry*, ACRI'06, pages 94–103, Berlin, Heidelberg, 2006. Springer-Verlag.

[22] N. Greene. Voxel space automata: Modeling with stochastic growth processes in voxel space. In *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '89, pages 175–184, New York, NY, USA, 1989. ACM.

[23] Mark Hendrikx, Sebastiaan Meijer, Joeri Van Der Velden, and Alexandru Iosup. Procedural content generation for games: A survey. *ACM Trans. Multimedia Comput. Commun. Appl.*, 9(1):1:1–1:22, February 2013.

[24] Dong hui Xu, Arati S. Kurani, Jacob D. Furst, and Daniela S. Raicu. Run-length encoding for volumetric texture. In *The 4th IASTED International Conference on Visualization, Imaging, and Image Processing*, pages 452–458, Marbella, Spain, 2004.

[25] Takashi Ijiri, Takashi Ashihara, Takeshi Yamaguchi, Kenshi Takayama, Takeo Igarashi, Tatsuo Shimada, Tsunetoyo Namba, Ryo Haraguchi, and Kazuo Nakazawa. A Procedural Method for Modeling the Purkinje Fibers of the Heart. *The Journal of Physiological Sciences*, 58(7):481–486, 2008.

[26] Lawrence Johnson, Georgios N. Yannakakis, and Julian Togelius. Cellular automata for real-time generation of infinite cave levels. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, PCGames '10, pages 10:1–10:4, New York, NY, USA, 2010. ACM.

[27] Tao Ju, Frank Losasso, Scott Schaefer, and Joe Warren. Dual contouring of hermite data. *ACM Trans. Graph.*, 21(3):339–346, July 2002.

[28] Sicilia F Judice, Bruno Barcellos, and Gilson A Giraldi. A cellular automata framework for real time fluid animation. In *Proceedings of the Brazilian Symposium on Computer Games and Digital Entertainment*, pages 169–176, 2008.

[29] Arie Kaufman. Volume visualization. *The Visual Computer*, 6(1):1–1.

[30] George Kelly and Hugh Mccabe. A survey of procedural techniques for city generation. In *ITB Journal*, number 14, pages 87–130, 2006.

[31] Joe Kniss, Simon Premoze, Charles Hansen, and David Ebert. Interactive translucent volume rendering and procedural modeling. In *Visualization, 2002. VIS 2002. IEEE*, pages 109–116. IEEE, 2002.

[32] Johannes Kopf, Chi-Wing Fu, Daniel Cohen-Or, Oliver Deussen, Dani Lischinski, and Tien-Tsin Wong. Solid texture synthesis from 2d exemplars. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2007)*, 26(3):2:1–2:9, 2007.

[33] A Lagae, S Lefebvre, R Cook, T DeRose, G Drettakis, D S Ebert, J P Lewis, K Perlin, and M Zwicker. A survey of procedural noise functions. *Computer Graphics Forum*, 29(8):2579–2600, 2010.

[34] John Levine and Levine John. *Flex & Bison*. O'Reilly Media, Inc., 1st edition, 2009.

[35] Bruno Lévy, Sylvain Petitjean, Nicolas Ray, and Jérome Maillot. Least squares conformal maps for automatic texture atlas generation. *ACM Trans. Graph.*, 21(3):362–371, July 2002.

[36] Aristid Lindenmayer. Mathematical models for cellular interactions in development I. Filaments with one-sided inputs. *Journal of Theoretical Biology*, 18(3):280–299, 1968.

[37] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, pages 163–169, New York, NY, USA, 1987. ACM.

[38] U. Meier, O. Lpez, C. Monserrat, M.C. Juan, and M. Alcaiz. Real-time deformable models for surgery simulation: a survey. *Computer Methods and Programs in Biomedicine*, 77(3):183 – 197, 2005.

[39] Melanie Mitchell, James P. Crutchfield, and Rajarshi Das. Evolving cellular automata with genetic algorithms: A review of recent work. In *In Proceedings of the First International Conference on Evolutionary Computation and Its Applications (EvCA'96)*, 1996.

[40] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. In *ACM SIGGRAPH 2006 Papers*, SIGGRAPH '06, pages 614–623, New York, NY, USA, 2006. ACM.

[41] Pascal Müller, Gang Zeng, Peter Wonka, and Luc Van Gool. Image-based procedural modeling of facades. In *ACM SIGGRAPH 2007 Papers*, SIGGRAPH '07, New York, NY, USA, 2007. ACM.

[42] Ken Museth. Vdb: High-resolution sparse volumes with dynamic topology. *ACM Trans. Graph.*, 32(3):27:1–27:22, July 2013.

[43] Radomír Měch and Przemyslaw Prusinkiewicz. Visual models of plants interacting with their environment. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, pages 397–410, New York, NY, USA, 1996. ACM.

[44] Gregory M. Nielson and Bernd Hamann. The asymptotic decider: Resolving the ambiguity in marching cubes. In *Proceedings of the 2Nd Conference on Visualization '91*, VIS '91, pages 83–91, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[45] Michael O'Neill and Conor Ryan. Under the hood of grammatical evolution. In *Proceedings of the Genetic and Evolutionary Computation Conference.*, 1999.

[46] Norman H Packard and Stephen Wolfram. Two-dimensional cellular automata. *Journal of Statistical Physics*, 38(5-6):901–946, 1985.

[47] Yoav I. H. Parish and Pascal Müller. Procedural modeling of cities. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pages 301–308, New York, NY, USA, 2001. ACM.

[48] Alexander Pasko, Oleg Fryazinov, Turlif Vilbrandt, Pierre-Alain Fayolle, and Valery Adzhiev. Procedural function-based modelling of volumetric microstructures. *Graphical Models*, 73(5):165–181, 2011.

[49] Darwyn R. Peachey. Solid texturing of complex surfaces. *SIGGRAPH Comput. Graph.*, 19(3):279–286, July 1985.

[50] K. Perlin. Tutorial: Computer graphics; image synthesis. chapter An Image Synthesizer, pages 333–342. Computer Science Press, Inc., New York, NY, USA, 1988.

[51] Nico Pietroni, Paolo Cignoni, Miguel Otaduy, and Roberto Scopigno. Solid-texture synthesis: A survey. *IEEE Comput. Graph. Appl.*, 30(4):74–89, July 2010.

[52] P Prusinkiewicz. Graphical applications of l-systems. In *Proceedings on Graphics Interface '86/Vision Interface '86*, pages 247–253, Toronto, Ont., Canada, Canada, 1986. Canadian Information Processing Society.

[53] P. Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[54] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[55] Michael Schwarz and Hans-Peter Seidel. Fast parallel surface and solid voxelization on gpus. In *ACM SIGGRAPH Asia 2010 Papers*, SIGGRAPH ASIA '10, pages 179:1–179:10, New York, NY, USA, 2010. ACM.

[56] Noor Shaker, Miguel Nicolau, Georgios N. Yannakakis, Julian Togelius, and Michael O'Neill. Evolving levels for super mario bros using grammatical evolution. In *Computational Intelligence and Games (CIG), 2012 IEEE Conference on. IEEE*, 2012.

[57] Ruben M Smelik and Klaas Jan De Kraker. A survey of procedural methods for terrain modelling. In *Proceedings of the CASA Workshop on 3D Advanced Media In Gaming And Simulation (3AMIGAS)*, pages 25–34, 2009.

[58] Thomas H Speller Jr, Daniel Whitneya, and Edward Crawleyb. Using shape grammar to derive cellular automata rule patterns. *Complex Systems*, 17:79–102, 2007.

[59] André Stauffer and Moshe Sipper. On the relationship between cellular automata and l-systems: The self-replication case. *Phys. D*, 116(1-2):71–80, May 1998.

[60] G Stiny and J Gips. Shape Grammars and the Generative Specification of Painting and Sculpture. In C V Friedman, editor, *Information Processing '71*, pages 1460–1465, 1972.

[61] P. M. Sweetser and J. H. Wiles. Virtual clothing in hybrid cellular automata. *Scripting versus Emergence: Issues for Game Developers and Players in Game Environment Design*, 4(1):1–9, 2005.

[62] Alexander Tarakanov and Andrew Adamatzky. Virtual clothing in hybrid cellular automata. *Kybernetes*, 31(7/8):1059–1072, 2002.

[63] Julian Togelius, Georgios N. Yannakakis, Kenneth O. Stanley, and Cameron Browne. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):172 – 186, 2011.

[64] Li-Yi Wei, Sylvain Lefebvre, Vivek Kwatra, and Greg Turk. State of the Art in Example-based Texture Synthesis. In *Eurographics '09 State of the Art Reports (STARs)*. Eurographics, March 2009.

[65] Peter Wonka, Michael Wimmer, François Sillion, and William Ribarsky. Instant architecture. In *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, pages 669–677, New York, NY, USA, 2003. ACM.

[66] Steven Worley. A cellular texture basis function. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, New York, NY, USA, 1996. ACM.

[67] Takashi Yokomori. Stochastic characterizations of E0L languages. *Information and Control*, 45(1):26–33, 1980.