Increasing the Performance and Realism of Procedurally Generated Buildings

Brian Sowers

Thesis submitted to the
College of Engineering and Mineral Resources
at West Virginia University
in partial fulfillment of the requirements for the degree of

Masters of Science in Computer Science

Tim Menzies, Ph.D., Chair
Tim McGraw, Ph.D.
Arun Ross, Ph.D.

Lane Department of Computer Science and Electrical Engineering
Morgantown, West Virginia
2008

Categories and Subject Descriptors:
K.8.1 Graphics, K.8.0 Games, I.2.6 Learning (Parameter Learning),
I.3.4 Graphics Utilities, I.3.5 Computational Geometry and Object Modeling

Keywords: GPGPU, split grammars, parametric learning, rule induction

Abstract

Increasing the Performance and Realism of Procedurally Generated Buildings

Brian Sowers

As multimedia such as games and movies grow, so does the need for content. Textures, 3D models, expansive terrain, sound effects, and other data must be generated to support and enrich these multimedia productions. As this need for content continues to grow, two critical problems emerge: the cost of hiring artists to create the content becomes extremely large, as does the amount of memory needed to store and manipulate the content.

To combat these issues, procedural content generation, or content generated algorithmically rather than via an artist, has been introduced. Algorithmically generating content allows for rapid creation of large amounts of certain classes of content with little human effort; further, this content can be represented extremely compactly, often by only exposing a handful of parameters.

In the realm of 3D building generation, split grammars have proven useful for generating a wide variety of buildings while being relatively intuitive. These split grammars have been used to generate entire cities full of detailed buildings with a fairly small number of rules.

Split grammars have two important areas which can be expanded upon: first, the writing of an appropriate grammar can require a significant amount of work and knowledge, especially when a grammar is required that must follow a certain building style while providing a high degree of variation. Second, applying these grammars to produce a building can be slow, often requiring an offline pregeneration phase which eliminates the usefulness the size benefits of the grammar's compactness.

For the first problem, we propose a data mining approach to refining preexisting grammars, wherein a user can specify buildings which they prefer, and from these preferences a set of rules will be generated that will guide future building generation. We will show that the generated rules have a high degree of accuracy when used to predict whether a user will like or dislike a building, often in the upper 90%.

For the second problem, we provide two areas of improvement: a preprocessing step which parses a split grammar to make it easier and more efficient to apply the grammar without loss of generality, and a scheme that allows the execution of a grammar entirely within a geometry shader on a modern graphics processing unit (GPU) such that building generation can take advantage of the parallelization found on modern graphics cards. We will show that this second improvement can provide a speed benefit anywhere between 3 and 10 times a purely CPU approach, with further speed benefits possible depending on the nature of the grammars.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank West Virginia University and its faculty for providing a quality computer science education. I would especially like to thank my research advisers Tim Menzies and Tim McGraw, who were exceptionally helpful throughout the duration of this research.

Thanks to Frances VanScoy, Andrei Smirnov, and Ismael Celik, all of whom provided me great opportunities to do research outside of the areas of this thesis. A special thanks to VanScoy, whose program convinced me to pursue a graduate degree. Thanks to Cindy Tanner and Jim Mooney, who allowed me to try my hand as a teaching assistant under them.

Arun Ross, John Atkins, Tim McGraw, Tim Menzies, Dr. Hiergeist, Jim Mooney, Elaine Eschen, and Dr. Cukic all deserve recognition for being exceptional teachers who I enjoyed every class with. Particularly, Ross, McGraw, and Menzies all taught an assortment of useful topics which contributed heavily to my research. I would like to thank them for serving as my research committee.

I would also like to thank my colleagues: Zach Milton, Daniel Baker, Nathan Moore, Omid Djalali, and Andrew Matheney, all of whom served as great sounding boards for advice and ideas.

Finally, thanks to my entire family for all the help they've given me.

Chapter 1

1 Introduction

Multimedia is a big business. As of 2006, the video game industry alone made over $12.5 billion [73], and that number has only grown since [12]. On the same token, multimedia is *big*. Recent games such as Gears of War, Army of Two, and God of War depict huge virtual environments composed of myriad buildings, plants, characters, and environments.

All this content does not come without a cost. Game teams involve large numbers of artists working for upward of two years. The time and money necessary to make modern virtual environments has grown tremendously since the days of Mario when players were content with 2D worlds made of frequently repeating images. Indeed, analysts have estimated that Halo 3 cost between $30 and $60 million to create [27].

Further, this content has become large in terms of memory consumption. 1997's Final Fantasy 7 took up three CD's. The recent Lost Odyssey takes up four DVD's, with most current console games requiring at least one DVD. Such huge memory requirements clearly increases installation times for non-console games and effectively prohibits online distribution for a large number of consumers.

To combat these problems, procedural content generation has been proposed [67]. Procedural content generation involves the creation of art assets algorithmically as opposed to requiring an artist to create all the assets. By doing this, large variations of assets can be created

with the same algorithms to, in effect, create more content faster. Furthermore, content can often be represented as the parameters input to the procedural algorithm, thus allowing a much smaller representation of the final content, resulting in huge memory savings.

There is no one-size-fits-all approach to generating procedural content. Nearly every type of content, from textures to buildings to foliage, requires its own specialized algorithms. For instance, textures can be generated via Perlin noise [19] while plants can be generated via L-Systems [61] while buildings can be generated via split grammars [48].

Even within a category of content, there are often a wide variety of algorithms to generate different classes of the content. Textures synthesis can make use of Perlin noise [19] or reaction-diffusion [81] or particle systems [65] to achieve various effects. Procedurally generating terrain can make use of fault formation [70] or midpoint displacement [71] or fractional Brownian motion [19], among other approaches. Clearly, there are a huge variety of algorithms to choose from depending on the nature of the content required.

This thesis will focus on procedurally generating buildings. Again, there are a large variety of methods for generating procedural buildings [74, 82, 48, 49, 77, 17, 16, 7, 51, 43, 42], and no one method is perfect for every application. We will focus on the approach which we believe is one of the most promising, split grammars [48].

Split grammars are a relatively simple approach based on L-Systems [61] to generate buildings. They are based on the concept of repeatedly subdividing and manipulating individual components of a building to achieve a desired building. They have been shown to be extremely flexible for generating a large variety of buildings while being intuitive enough for humans to be able to achieve a desired set of buildings [31, 48].

However, split grammars are not without their limitations. It is considerably slower to generate a building via split grammars than it is with simpler, more limited forms of building

generation [26]. This speed deficiency makes generating entire cities via split grammars unwieldy; if a city is generated every time an application starts, the generation may take a long time (more than an hour [48]), which would not be acceptable in a game setting or other applications. If a city is generated on-demand, there would be a noticeable lag as buildings are being generated, especially if many buildings must be generated to fill the current viewport. If a city is generated in a preprocessing stage and stored as models which could be more quickly loaded, we would effectively lose the memory benefits of using split grammars in the first place (though we would still have the benefit of being able to generate a large number of buildings with considerably less artistic involvement).

It can also be difficult to write a split grammar reflecting exactly what a user wants, especially if the user is unversed in architecture. As more and more variation is introduced into the system, ensuring that the generated buildings are always what the user intended is challenging.

The goal of this thesis is to approach solving both problems. We would like to explore various methods of making building generation faster, and we would also like to explore methods for refining building generation such that a building can be "tuned" to user preferences. We believe that working on these problems can give split grammars a wider appeal, and hopefully in the future they will be used in actual game settings.

We explore two methods for approaching the speed problem. The first is a preprocessing stage on the grammars – an interpretation with Lex and Yacc – such that grammars can be easily (and thus quickly) evaluated. We believe that this preprocessing stage is important, because it transforms the grammar into a form that can be more readily interpreted by simple algorithms, making an implementation of split grammars easier and making execution time faster. We can also show that this transformation has useful side-effects: the result of transformation need only

be performed once per grammar and can be reused, and the transformation leads naturally into our second approach for the speed problem.

The second approach for the speed problem involves taking advantage of the inherent parallelization found in recent graphics cards [24] to generate a building via processing various components in parallel, thus achieving greater turn-around time during generation. We have observed that generating buildings via split grammars can take great advantage of this parallelization, as the nature of split grammars can be naturally done in parallel.

For the second problem – that of refining split grammars to approximate user desires – we present a machine learning technique to isolate various "decisions" that occur throughout building generation. By taking advantage of common machine learning techniques, the most important of which being rule induction [34], we can determine which series of decisions lead to a certain building being generated and thus refine those decisions to better approximate user desires. By doing so, we can generate further buildings which more closely represents what the user wants.

The result of this research is a program which allows us to perform all of the above executions: load in and execute a split grammar rule set, preprocess the rule set for quick execution, increase the speed of the execution via GPU parallelization, and refine the buildings generated via split grammars. We believe this program is important to validate our belief that building generation is indeed faster and that we can refine building generation to reflect user preferences, and it is also a useful program in its own right to make use of the techniques we have created.

To help validate our beliefs on refining buildings, we make use of the Weka (Waikato Environment for Knowledge Analysis) to perform our rule induction and to evaluate the effectiveness of the rule induction. By using this standard tool, we can make more accurate

statements about our building refinement that would not be possible by simply attempting to observe the results to make sure they "feel right" under complex conditions.

Thus, throughout this paper we will discuss the various novel techniques we implemented to solve the issues of split grammars, and we will use the above tools to evaluate our techniques and make statements on their effectiveness.

1.1 Contributions of this Thesis

This thesis contributes a variety of findings to the literature, including:

- A preprocessing stage that can be run on split grammar rule sets to make building generation faster. This preprocessing stage will help to transform the rule set into something that can be more easily interpreted and transformed into other forms.
- A novel scheme using geometry shaders for generating buildings entirely on modern graphics hardware.
- A thorough comparison and analysis of speed differences when generating buildings on the GPU as opposed to the CPU.
- A scheme for determining critical points ("decision points") in building generation that determine the final appearance of the building.
- A rule induction scheme for refining decision points such that they more accurately represent user preferences.
- An objective study of the effectiveness of the rule induction.
- Various techniques for ensuring that future buildings adhere to the rules generated during the rule induction stage.

1.2 Structure of This Document

The remainder of this document is organized as follows:

- Chapter 2 is a literature review describing related work.
- Chapter 3 describes our method for increasing the speed of building generation. It includes both a description of our preprocessing and a description of our GPU parallelization scheme. It also includes various experiments to test the effectiveness of the parallelization scheme.
- Chapter 4 describes our method for refining building generation using rule induction. It includes various experiments to test the effectiveness of the rule induction. It also includes potential ways to enforce inducted rules to ensure that future buildings adhere to the rules.
- Chapter 5 concludes the study by summarizing the accomplishments towards each thesis goal and describes future work.

Chapter 2

2 Related Work

2.1 Procedural Content Generation

2.1.1 Overview

As games and other media grow in complexity and hardware advances, the need for content grows accordingly. Traditionally, artists would be tasked to create such content. Roden and Parberry identify four problems with this approach [67]:

1. As technology expands, artists need increasingly more time to create content.
2. Artist-created content becomes difficult to modify, especially as it becomes more complex.
3. Content creation tools often output content in their own formats as opposed to the formats necessary for the project using the content.
4. Interactive games are becoming huge, and artists will become incapable of generating all the necessary content.

Procedural generation of the content has been proposed to combat these problems. Procedural content is content that is generated algorithmically instead of by artists [30], where appropriate controls are given to allow the generated content to match certain criteria [67].

Throughout the remainder of this section, we will provide an overview of methods used to procedurally generate various types of content, including textures, buildings, cities, and even

worlds. This will provide some frame of reference when we expand upon building generation and will also provide methods for making procedural buildings (and environments containing them) more realistic.

2.1.2 Textures

Textures, or images which can be laid on geometry to increase realism, are very commonly procedurally generated. We observe that there are essentially four classes of procedural textures: those which rely heavily on some specified noise algorithm, those which rely on simulating natural phenomenon, those which exploit the structural regularity of certain textures, and those which extract patterns and repetitions based on example textures.

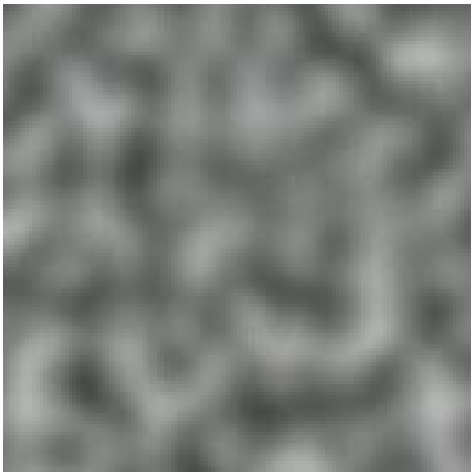

Figure 1: Two dimensional Perlin noise.

From the first class, the utilization of a well-defined noise function (often Perlin noise) is common (Figure 1) [19]. The function, which resembles white noise after a Gaussian blurring, can be used as a basis for a large class of textures, including marble, stone, clouds, and wood. Perlin provided a high-level programming environment for the generation of such textures [58].

From the second class, various natural phenomenon can create interesting looking textures. Witkin proposed simulating the biological process of reaction-diffusion, which has been useful for generating patterns resembling zebra stripes or animal spotting [81]. Reeves used particle system simulations to generate "fuzzy" textures such as fire, water, and clouds [65].

From the third class, we can clearly see that certain textures, such as brick patterns or

tiles, possess some structural regularity which can be exploited. Miyata, for instance, uses structural regularity to generate stone wall patterns [47]. Voronoi diagrams [60] can be utilized to create flagstone textures. Legakis, et al attempted to increase the appeal of structural texture generation by performing a structural analysis of the underlying model to be textured to generate a more appealing texture [38]. Lefebvre and Poulin provide a mechanism for analyzing structural textures to extract features (such as brick spacing or mortar thickness) which can then be used to generate new textures [36].

From the fourth class, Wei and Levoy use vector quantization to quickly synthesize large textures which resemble a smaller texture [79]. Lefebvr and Neyret provide a mechanism for sampling provided patterns to generate a large, non-repetitive texture for use with landscapes [37].

2.1.3 Terrain

Figure 2: Musgrave's erosion model used to generate terrain. [50]

Terrain is also commonly procedurally generated, given how difficult it can be to convincingly model a large body of land. Two very simple algorithms for procedurally generating terrain are the Fault Formation [70] and Midpoint Displacement [71] algorithms. Fault Formation, as its name implies, generates fault lines in a terrain heightmap and modifies the nearby heights accordingly. Midpoint Displacement continually subdivides a heightmap, modifying the height of the center of each subdivision by some function of its neighboring heights. While both algorithms produce decent results, they can be rather unconvincing for real terrain.

A more convincing approach was proposed by Musgrave (Figure 2) [19] which utilizes fractional Brownian motion (which is built upon Perlin noise) to generate more realistic terrain.

To increase realism, Musgrave also proposed an erosion model which can be applied to terrain [50].

Olsen also attempted to further increase the realism of generated terrain by using a combination of midpoint displacement and Voronoi diagrams to generate a heightmap. He also introduced an erosion model to increase realism [53].

Regardless of how the terrain is generated, rendering large terrain in real-time requires special algorithms. For this, the ROAM (real-time optimally adapting meshes) algorithm [18] was created along with the newer Clipmaps [41].

2.1.4 Planets

There has been very little work done toward generating entire 3D planets. Greuter et al proposed a framework for generating procedural worlds which builds heavily off of procedural city and building generation [25].

O'Neill actually explored the generation of entire planet-sized terrain (without populating the terrain with buildings or cities) [54]. For this, he used fractional Brownian motion with a special implementation of the ROAM algorithm to make the terrain appear spherical.

2.1.5 Dungeon / Level

Random dungeon / level generation has been featured in games (Diablo, Diablo II, Hellgate: London). A simple random dungeon generator (which can still be seen in games such as Persona 3) is trivial to create, requiring only the creation of rooms and connections between those rooms. Adams provided a more robust scheme for random dungeon generation [1]. He

proposed a solution based on *graph grammars*, which are grammars operating on graphs as opposed to strings. He also demonstrated several enhancements to make the levels more convincing and engaging.

2.1.6 Plants

It has been observed that plants possess some structural similarities to fractals which has opened up a large area of research on plant generation based on L-Systems [61], which we will talk heavily about in the next section. Prusinkiewicz et al provide an elaborate discussion on L-Systems and how they can be used for the modeling of plants, which is depicted in Figure 3 [62].

Figure 3: A tree generated using L-Systems.

After using an L-System, a series of points and lines are generated which can then be used to generate more photorealistic models. Bloomenthal presented a method of doing this that involved generated conical shapes based on the connections [8]. Spatz and Speck presented methods to make the plants more realistic by integrating biomechanics into the plant models to simulate things such as gravity or

environmental factors [29]. Lluch et al further provided multiresolution techniques to increase the realism of the generated imagery [39].

2.1.7 City Generation

Kelly and McCabe provide a recent summary of various techniques for city generation, including the strengths and weaknesses of each method [31]. We will provide a brief summary of some relevant techniques here.

Perhaps the simplest form of city generation was conceived by Greueter et al, which simply fills the view frustum with as many (also procedurally generated) buildings as it will hold [26].

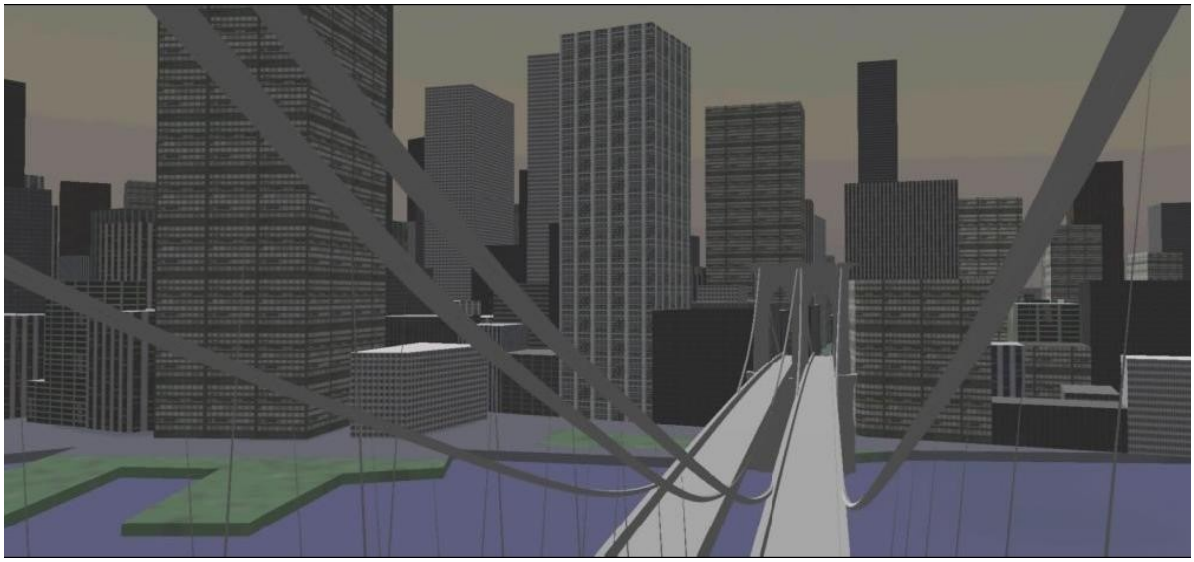

Figure 4: A city generated via Parish and Müller's generation scheme.

Kelly and McCabe attempt to generated road networks that adapt to terrain while servicing land area. The user has some control over the way the road network is generated. Once the network is generated, regions enclosed by primary and secondary roads can then be

filled with buildings [32].

Parish and Müller use a similar approach, the primary difference being the way the road network is developed [56]. For this, they utilize L-Systems, extending the systems to consider global goals and local constraints. Their system allows for the input of various image maps such as land-water boundaries and population density, such that the generated road network can automatically achieve various desirable features (make denser road networks in more populated areas or avoid harsh changes in elevation, for instance). A picture of one of their generated cities can be seen in Figure 4.

Figure 5: Building generated via split grammars.[48]

Sun et al also use various input images to guide their city generation, but they dispense with L-Systems in favor of a rule-based approach [76]. This rule-based system is careful to adjust itself to meet constraints on the road network.

Laycock and Day take a different approach, using building footprint data and LIDAR information to construct a full 3D model of urban environments [35]. This allows for a much greater degree of accuracy when modeling actual environments, but does not lend itself well to generate novel cities.

2.1.8 Buildings

Building generation techniques can often be found alongside city generation techniques [31, 26, 32, 56]. Again, Greuter et al proposed a very simple model for building generation which involved the semi-random combination of simple geometric solids [26]. This model allows for the very fast creation of urban buildings, but is not suited to creating more varied architecture [31].

Figure 6: Building generated via split grammars (2). [48]

Shape grammars [74] have been introduced to provide more varied buildings and allow a user greater control over the produced building. A shape grammar is a grammar which operates on shapes, "rewriting" shapes with other shapes until there are no more shapes to rewrite. These are similar to L-Systems; however L-Systems traditionally only operate on strings [61]. It has been shown that shape grammars can generate varied buildings, including Queen Anne houses [75, 22].

Wonka et al proposed a similar approach dubbed *split grammars* [82] (a basic form of which is addressed in [56]), which we will talk heavily about in section 2.3. These split grammars were also the focus of [48]. A split grammar similarly operates on shapes (building components), focusing on splitting these shapes and manipulating the resulting shapes to generate complex architecture. [48] illustrates how these split grammars can lead to diverse, rich

architecture and show an example of the construction of Pompeii. Müller et al proposed a way to automatically generate a split grammar for building facades using image analysis techniques [49]. Figures 5, 6 and 20 show buildings generated from split grammars.

Outside of grammars, there is a field of research concerned with taking existing photographs or building models and extracting information or generalizations to create more buildings, either identical to the template or varied. Thiemann and Sester presented a method for generalizing 3D buildings based on input models such that the key features of the building are preserved and can be used to generate derivative buildings [77]. Debevec et al presented a method for taking sparse sets of photographs and modeling scenes based on those photographs [17].

Other approaches concerned with rendering architecture attempt to provide more generic modeling capabilities. Cuter et al demonstrated a scripting language that could be used to generate solid models [16], which could conceivable be used for building generation (though its uses were not limited to this). Birch et al illustrated a tool based on drawing models from a large library of prototypes to avoid repetition in architectural structures [7].

Most of the above methods are concerned with building exteriors or small components of buildings. Noel presented a method for generating interior floor plans using a graph rewriting approach [51], which Martin attempted to expand upon [43] (a picture of a floor plan using this method can be seen in Figure 7). Martin provided a description of his extensions plus a summary of some mass model and exterior generation in [42].

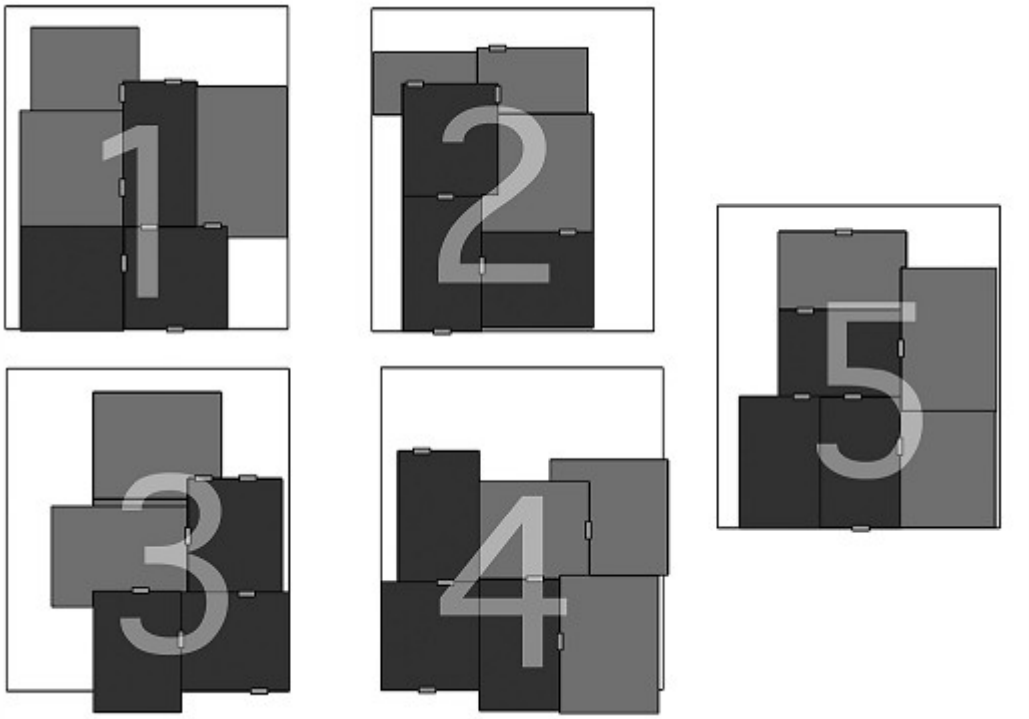

Figure 7: Interior floor plans generated by Martin. [43]

2.2 L-Systems

2.2.1 Overview

L-Systems are basic string rewrite grammars which are the backbone of various procedural content [61, 31, 56] and which also serve as a basis for split grammars [82]. The rest of this section will provide a thorough description of L-Systems, their capabilities, and how they can be interpreted for procedural content. An extensive treatment of L-Systems can be found in [61]. Programs which allow a user to experiment with L-Systems can be found in [59] and [44].

2.2.2 String Rewriting

A simple L-Systems consists of a set of non-terminals N, terminals T, production rules R, and some axiom A. Starting with the axiom, for each rewrite over a given number of iterations I, each non-terminal in the string is rewritten based on an operation which applies to that non-terminal. For instance, given the following rules where F is a non-terminal and + is a terminal:

F -> F + F

And the axiom:

FFFF

After one application of the grammar rules, we are left with:

F+FF+FF+FF+F

And after a second iteration, we end up with:

F+F+F+FF+F+F+FF+F+F+FF+F+F+F

This process is continued until we have applied the grammar rules I times or we are left with

only non-terminals. As a second example, consider the grammar:

F -> FA

A -> AB

B -> BC

And the axiom:

ABCF

After one iteration, the string becomes:

ABBCCFA

And after a second iteration, the string becomes:

ABBCBCCCFAAB

2.2.3 Parameterized L-Systems

In the realm of procedural content, researchers make heavy use of *parameterized L-Systems*. These systems operate similarly to basic L-Systems except that the terminals and non-terminals can take some number of parameters, and mathematical operations can be performed on these parameters. The general format for a parameterized rule resembles the following:

F(a) -> *some operation*

Where *a* is a parameter of F. For example, consider the following parameterized L-System:

F(a) -> F(a/2) + F(a/2)

If we are given the axiom:

F(30)

Then after a single iteration, we end up with:

F(15) + F(15)

And after two iterations, we have:

F(7.5) + F(7.5) + F(7.5) + F(7.5)

It is perfectly acceptable that a non-terminal should take any number of parameters. So the following grammar is valid:

F(a,b,c) -> F(a/1, b/2, c/3)

X(x,y) -> X(x * x * x, y)

2.2.4 Predicates

When dealing with parameterized L-Systems, the concept of predicates, or conditions which must be true for a rule to activate, becomes useful. A predicate is a boolean operation that must evaluate to true for the given rule to apply. For example, we present the rule:

F(a) : a > 0 -> F(5 / a)

Here the predicate a > 0 must be evaluated if F(a) is to be replaced with F(5 / a). In this instance, the predicate is useful for preventing a divide-by-zero error.

2.2.5 Stochastics

For procedural content, some degree of randomness is often desired to provide variation in the produced content. Randomness gives us the useful feature of *database amplification* [61], or the ability to expand our database of produced content without actually creating any new content ourselves.

With L-Systems, it is possible to provide a probability that some given rule will apply. When evaluating a rule, we first must determine if that rule actually activates based on its

probability (often by selected a uniform random number between 0 and 1 and evaluating whether that number falls within the given probability).

Given the example:

F -> F + F : 0.5

Here, the rule F->F+F has a 50% chance of actually being activated. Thus, when given the axiom:

F

There is an even chance that either of the following two strings will be generated after one application:

F

F+F

Note that we must evaluate the probability at each rewrite; thus, the string:

FF

May end up as any of the following:

FF

F+FF

FF+F

2.2.6 Interpretation for Procedural Content

Figure 8: Using a control string to move a LOGO-esque turtle. [61]

By treating a string as a series of commands to a LOGO-esque "turtle" [40], it is possible to generate an image from the string (see Figure 8). The turtle T is specified by a position P=(x,y) and an angle R. Each terminal or non-terminal character in the string represents a command that uses T's state and alter's T's state accordingly. Following are a list of possible interpretations for characters:

F: move the turtle forward N units, drawing a line as the turtle "moves"

f: move the turtle forward N units without drawing a line

+: rotate the turtle D degrees clockwise

-: rotate the turtle D degrees counter-clockwise

Thus, if we specify a given N and D, it is easy to see how a string of commands given to the turtle might produce some output. For example, if N is 5 units and D is 90 degrees, it is easy to see that the string

F+F+F+F

will produce a square with area 5*5.

As the command string becomes more complex, the complexity of the resulting image also increases. Figure 9 shows the result of executing an L-System four times to produce a

command string which is interpreted by our turtle.

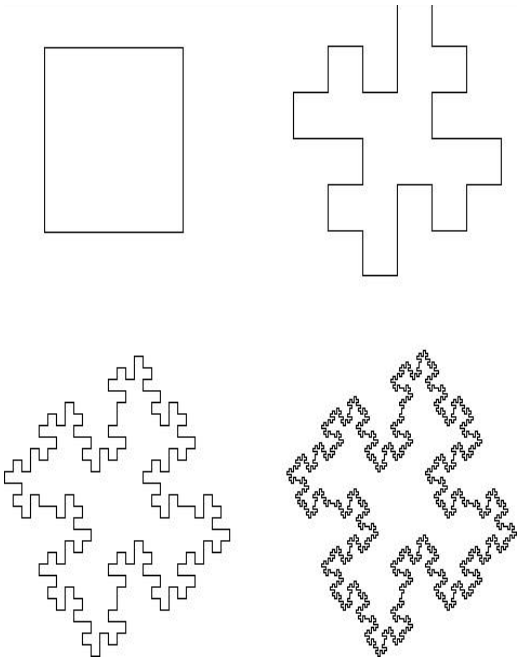

Figure 9: Applying an L-System multiple times to yield a final, more complex command string. [61]

Instead of specifying an N and D a priori, it is possible to use the notion of parameterization discussed earlier as a means of providing a unit number or degree for each individual command. As an example, consider the string

F(10)

where here the parameter represents the number of units to move forward. The same concept can also be applied to + or -, where the parameter represents the number of degrees to rotate.

Although using this system to generate images opens up a vast number of possible line-based images, there are three important extensions that make the command-string interpretation more suitable for procedural plant generation. The first is the implementation of a state stack for the turtle. Using this state stack, the turtle's position/rotation can be pushed onto the stack and then retrieved later, exposing the possibility for inducing “branches” in a generated image. Two simple commands are introduced to facilitate this stack: a push (“[“) and a pop (“]”), which are

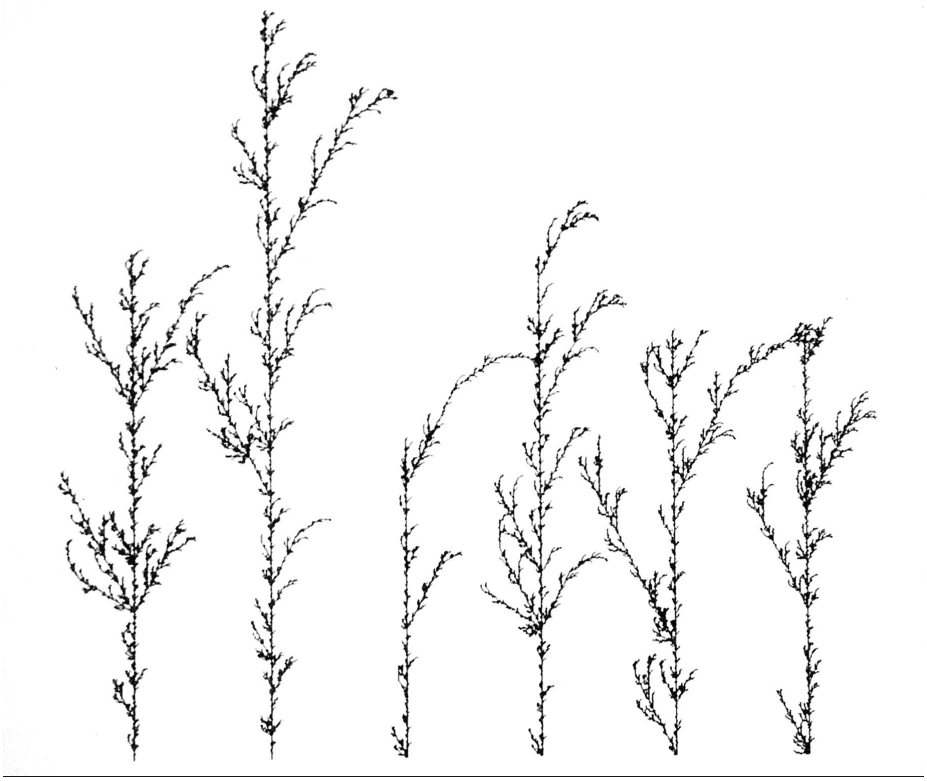

Figure 10: Plants generated via an L-System with branching and stochastics. [61]

incorporated into the command string like any other command. To illustrate, consider the following example:

F(5) [+(30) F(10)] [-(30) F(10)] F(10)

In this example, the turtle would first start by moving forward five units. Then it would push its state, rotate clockwise thirty degrees, and move forward ten more units. Then it would pop its state, essentially undoing the rotation and move, and perform the same action in the

counter-clockwise direction. Just before the last command, the turtle would be located in the same position it was after the first command. Finally, it would move straight ten units. The resulting image would have a stem and three "prongs" extending from it, which would be considerably more difficult to achieve without the use of the state stack. Figure 10 shows a use of the stack.

The second extension introduces the concept of replacing characters in the command-string with images or textures, for instance to add leaves. For this, special commands must be added to the string representing these images (or a single parameterized command can be used). When these are encountered, instead of drawing lines, one places an image at the turtle's position, which can be rotated by the same angle as the turtle. Consider the following extension to the above string:

F(5) [+(30) F(10) Ileaf] [-(30) F(10) Ileaf] F(10) Ileaf

The string will essentially tell the turtle to drop a "leaf" image at the end of each prong. These special commands can be further parameterized to indicate the state of the turtle after inserting the image. This would allow a user to tell the turtle to continue from the image at a certain position or angle.

The third extension pushes the interpretation into 3D. Here, alter the definition of the turtle such that it is defined by a three dimensional position P and an arbitrary orientation vector R. The commands F and f require no change – they still move the turtle along its direction, drawing or not drawing a line respectively. The extension will dispense with the + and – commands in favor of the following, more specific commands:

X(angle): rotates R about the X axis

Y(angle): rotates R about the Y axis

Z(angle): rotates R about the Z axis

Notice here that although we originally provided + and – as two separate rotations going clockwise and counter-clockwise, this is not necessary with our new parameterized commands; rotating about a positive angle will rotate clockwise and a negative angle will rotate counter-clockwise.

With these new commands in place, it is perfectly possible to construct a command-string via an L-System which represents the skeleton of a fully 3D tree. From this skeleton, it is then possible to generate a 3D mesh that more accurately represents a tree [8].

2.2.7 Other Uses of L-Systems

As illustrated earlier, trees, foliage, and other plant life make heavy use of L-Systems. Prusinkiewicz has shown how they can be used to generate music [63]. Müller and Wonka have extended L-Systems to allow procedural building generation [82, 48], which will be the focus of the next section.

2.3 Split Grammars

2.3.1 Overview

Split grammars began as a method for modeling facades of buildings [56] but later expanded into a method for generating full building exteriors [82, 48]. At their core, split grammars are very similar to L-Systems, complete with stochastic capabilities and parameterization. Split grammars differ in that instead of operating on strings, they operate on building components. An axiom will be a set of building components (often a single component representing an initial lot), and that axiom will be replaced by other building components via grammar operations until no more grammar operations can be performed. The final list of building components – those which can not be acted upon by any grammar rules – represents the final building. The grammar itself provides a set of operators useful for subdividing (hence 'split') and manipulating mass models to form complex geometry. For geometry which can not be represented naturally by splits, there is the capability to insert precreated 3D models.

2.3.2 Component Representation

As was stated previously, split grammars operate on building components. Here, a building component is represented by a name, a type, and three three-tuples: a position, a size, and an orientation vector. The type of a component is the type of mass model the component represents; this may be a cube, a cylinder, a face, or some other arbitrary model. The position is the absolute position in the building's local coordinate system that the building occupies; the size specifies the total bounding region that the component encompasses – if the component is a cube

or face, it will occupy this entire volume, but otherwise it may not. The orientation vector represents which way the component is facing, which can alternatively be represented as its rotation about its own coordinate system. Figure 11 shows a depiction of this representation.

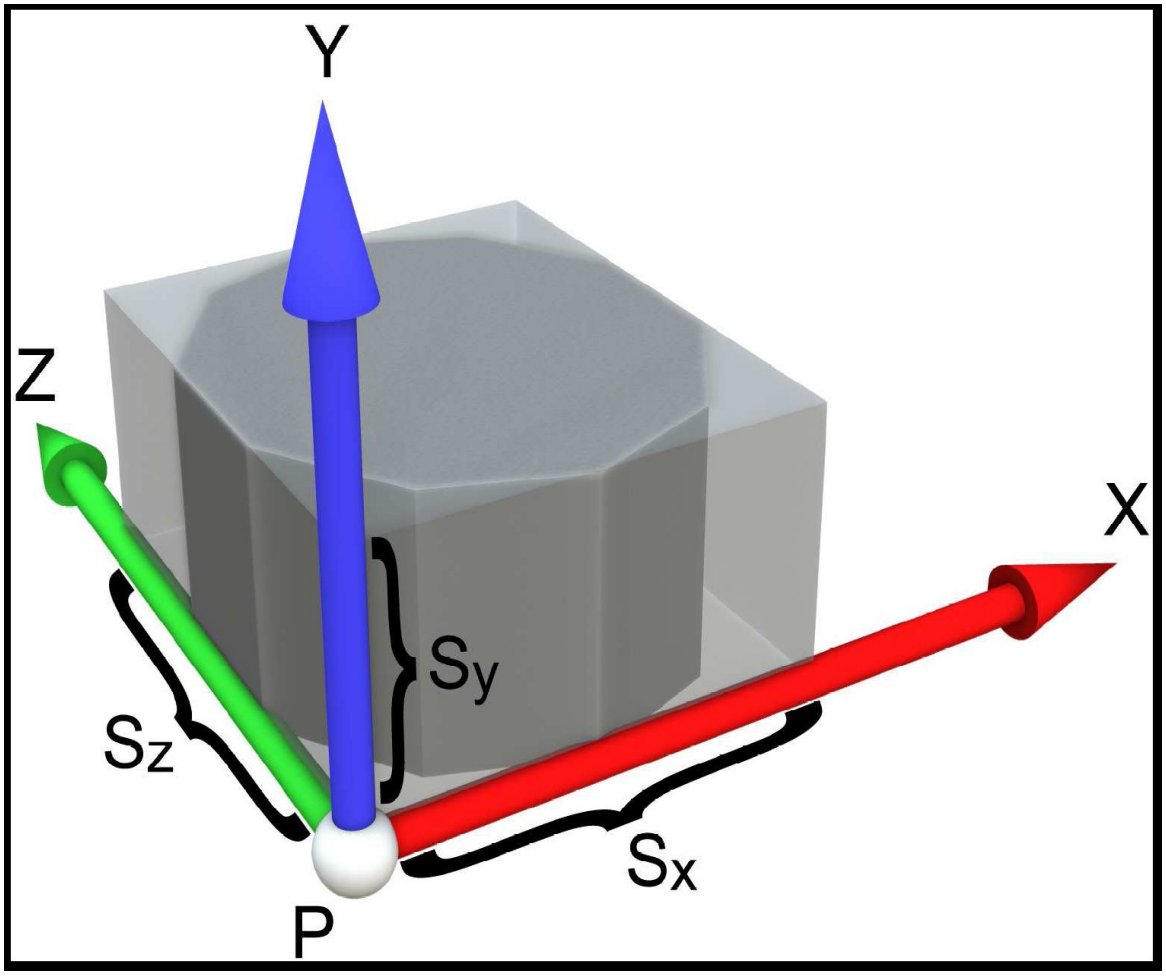

Figure 11: Building component representation. [48]

2.3.3 Basic Operation

A general split grammar rule takes the form:

ID : COMPONENT -> OPERATION { REPLACEMENTS }

Here, the ID is a number uniquely identifying this rule. COMPONENT is the building

component that this rule acts upon. OPERATION is some action to perform on the component. These operations are somewhat more involved than the simple string rewrites of L-Systems, so we will reserve talking about them until the next section. REPLACEMENTS represents a (sometimes optional) list of components that may replace the current component. A REPLACEMENT of "empty" means that a certain component is to be removed.

Note that predicates and stochasticism can be added almost identically to the way they are added to L-Systems:

ID : COMPONENT : PREDECESSOR -> OPERATION { REPLACEMENTS } : PROB

Where PREDECESSOR is again a boolean condition that must evaluate to true for the rule to activate, and PROB is the probability of this rule firing.

Split grammars can also specify that multiple operations should occur in the event of a rule activating. In this case, the syntax looks like the following (omitting predecessors and probabilities):

```
ID : COMPONENT   -> OPERATION1 { REPLACEMENTS }
                    OPERATION2 { REPLACEMENTS }
                    OPERATION3 { REPLACEMENTS }
```

and so on. When this rule activates, all of the operations will be performed in order.

We may also provide a probability for lists of operations, such that at least one operation list will always be chosen, where the probability of it being chosen is related to the probability of the other operation lists being chosen. The syntax resembles the following:

```
ID : COMPONENT   -> OPERATION1A { REPLACEMENTS }
                    OPERATION2A { REPLACEMENTS } : PROB_A
                    -> OPERATION1B { REPLACEMENTS }
                    OPERATION2B { REPLACEMETNS } : PROB_B
```

Here, PROB_A and PROB_B should sum to 1. PROB_A specifies the probability that all the operations before it will be executed when the current rule activates, and PROB_B specifies the probability that OPERATION1B and OPERATION2B will be performed.

Thus, using the above grammar rules, we start with an axiom (an initial set of building components). For each component, find a grammar rule whose COMPONENT matches the name of the component and who has a PREDECESSOR evaluating to true (if there is no PREDECESSOR, the rule can be evaluated), evaluating any necessary probabilities to determine if the rule should apply. If no rules apply, the component is added to the *final* list which represents the final building. If a rule does apply, perform the operation and add any necessary replacements to the working set of building components such that, on the next pass through the grammar, the added components can be acted upon. If no grammar rules are activated, execution of the grammar is finished and the set of components in the *final* list represents the final building.

2.3.4 Split Operations

Here we will provide a description of many of the different operations which can be applied to components. The first operation is Size, which resizes a component

Size(NewX, NewY, NewZ)

Here, NewX, NewY, and NewZ represents the new size of the component. These sizes can be specified in absolute sizes (10 units, 20 units, etc) or *relative* sizes (denoted by an ^), where by relative we mean relative to the components current size. For instance, using

Size(2,1,1)

makes the component 2 by 1 by 1 in size, whereas

Size(2^,1^,1^)

extends the component to twice its size along the X axis while not changing its size along the Y or Z axis. Thus, if the component was 50 by 20 by 20, it would now be 100 by 20 by 20.

If a replacement is given, the current component will be replaced by a newly named component of the same size.

The next command is *Subdiv*, which subdivides a component along a given axis (Figure 12 depicts a facade subdivided into distinct sections). The syntax is the following:

Subdiv(axis, Size1, Size2, ..., SizeN) { Replacement1, Replacement2, ..., ReplacementN }

Here, Subdiv will take the current component and split it up into N new components with sizes specified by Size1 to SizeN and names given by Replacement1 through ReplacementN. So, for instance, given a component that is 50 by 20 by 20, calling the following:

Subdiv(X, 20, 30) { LEFT | RIGHT }

will create two new components, LEFT and RIGHT, where LEFT denotes the left portion of the component and has a size of 20 whereas RIGHT has a size of 30. Again, we can use relative sizes in Subdiv. Consider

Subdiv(Y, 2^, 1^, 1^) { BOTTOM | FLOOR1 | FLOOR2 }

The above command will divide a component into three new sections, where the BOTTOM section is twice as large as the top two sections.

Again, specifying "empty" as the replacement will remove a component. Thus:

Subdiv(X, 1^, 2^, 1^) { LEFT | empty | RIGHT }

will remove the center of a component, where the center is twice as large as the LEFT and RIGHT components.

Repeat is similar to Subdiv:

Repeat(axis, splitSize) {REPLACEMENT)

Repeat will split a component up into smaller components each having size splitSize and name REPLACEMENT. Repeat will continue splitting the component until it can make no further splits. If the component's size is not even divisible by splitSize, the final component will be smaller than the other splits.

The *Comp* command takes a component and breaks it apart into its various components. For instance, a 3D cube component is made up of faces, edges, and points. The Comp command allows us to break apart that cube and treat each component individually such that the grammar can act upon those components. The Comp command has the following syntax:

Comp(type) { REPLACEMENT }

Here, type represents the type of component the current component should be broken into. For instance, specifying *faces* as the type breaks the component up into its component faces, where each face as the name REPLACEMENT. One could also specify *points*, *edges*, or *sidefaces* (all the faces except the top and bottom).

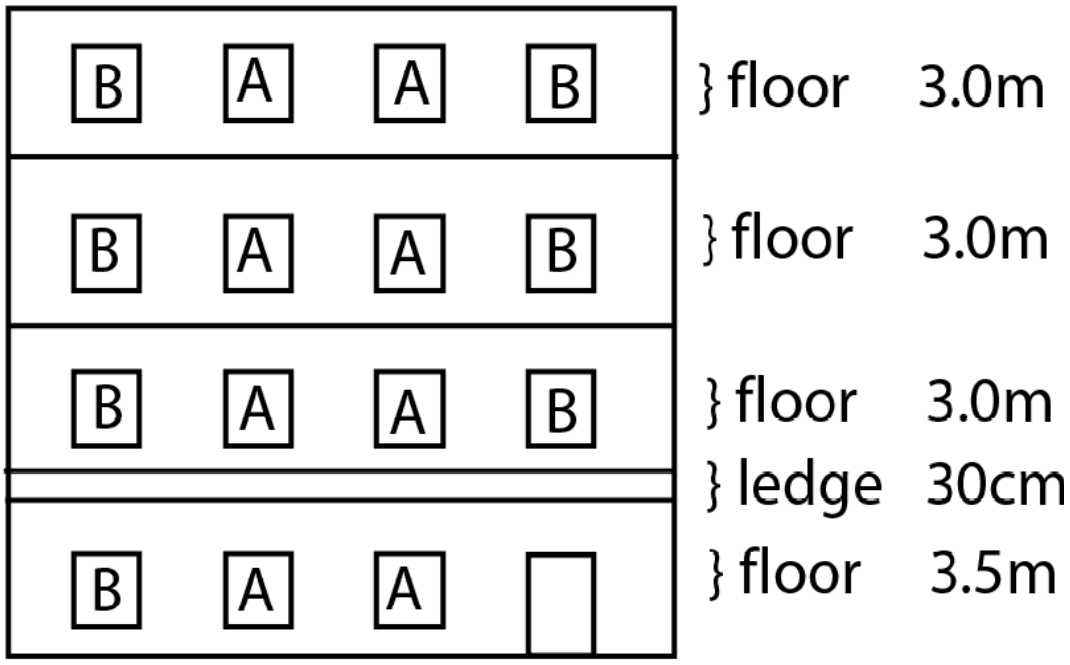

Figure 12: A simple building facade from [48] which can be represented as a series of splits.

2.3.5 Other Operations

The previously mentioned operations can be used to make a large variety of buildings, but there are still many others. There are operations to move and rotate the current component (*Translate* and *Rotate*, respectively). *Replace* will simply replace one component with another of a new name (or if "empty" is specified as the replacement, will remove the current component). For a more thorough description of all the different operations, we refer the reader to [48].

2.3.6 The Current Working Component and the Component Stack

It is often useful when writing an operation to refer back to the current component to use its properties. For instance, when resizing a component, one may wish to take the current component and add some size to it, where this addition is not relative. Split grammars provide this capability via use of the *Scope* structure. Scope.sx represents the current component's size along the X axis, Scope.sy represents its size along the Y axis, and so on. Via use of this structure, we can access all the different elements of the current working component (its position - .px, .py, .pz; its size - .sx, .sy, .sz; its rotation - .rx, .ry, .rz). Thus, if we wanted to perform the above example, we could write:

Size(Scope.sx + 10, Scope.sy + 20, Scope.sz + 30)

When manipulating the component - for instance when translating, rotating, or resizing it - we may wish to make some modification, then return to the previous state of the component to perform some other operation. This idea is parallel to the idea of returning the turtle of an L-System to its previous state, and the solution is practically identical: split grammars provide a

component stack, where the current component can be pushed onto the stack and popped off at a later time. As with L-Systems, [represents a push and] represents a pop.

2.3.7 More Randomness

Providing probabilities for operation lists does not give the full amount of randomness necessary. For instance, a user may want to resize a component within some degree of randomness, and providing a different rule for each different size is both limiting and cumbersome. Therefore, [48] introduces the *rand* function which can be used within operations:

rand(min, max)

rand will pick a random variable from a uniform distribution between min and max. Thus, to perform the above example:

Size(Scope.sx * rand(0.5, 1.0), 1^, 1^)

This will resize the component such that it can be between half its length or its full length along the X axis.

2.3.8 Injecting 3D Models

Although the simple resize/split/subdivide paradigm works well to represent many buildings, more complicated geometry can not be represented naturally using the split grammar concept. A more flexible system is necessary. For this, split grammars provide the capability to inject arbitrary 3D models into the system. Although this loses some of the benefit of procedural generation since these models must be precreated and are relatively inflexible, it is sometimes necessary for realism. For instance, it is difficult to model a roof procedurally, so providing a

model for portions of a roof can improve the generated building considerably.

This functionality is exposed via the Model operation:

Model(modelfile)

Although it is not a technical limitation that the inserted model should automatically be a terminal component, this is typically done in practice.

Müller and Wonka provide a separate operation for specifying a roof:

Roof(type)

Where type is the type of roof (hatched, flat, etc – refer to Figure 13). However, this functionality can be emulated using the Model operation.

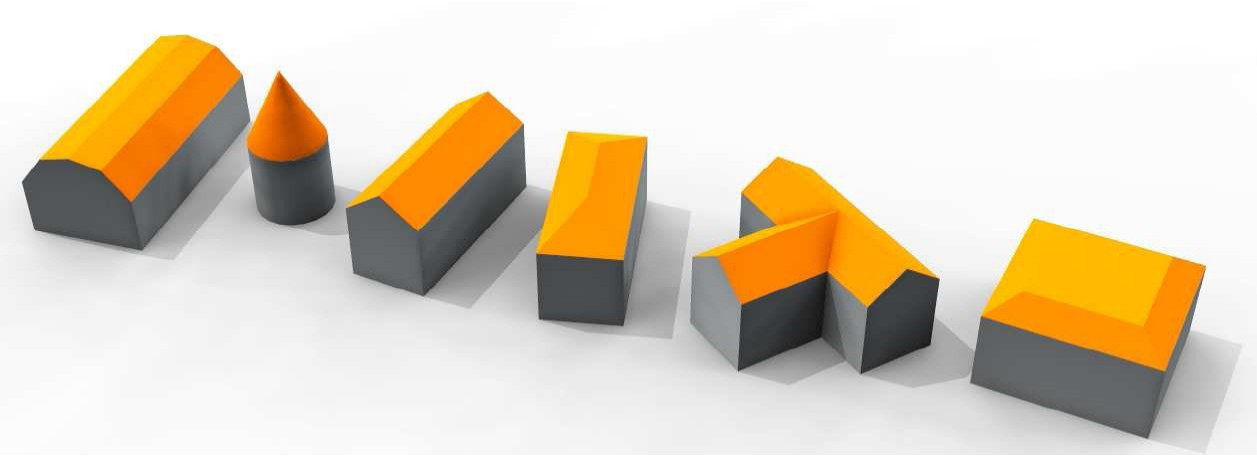

Figure 13: Depiction of various roof structures. [48]

2.3.9 Occlusion Queries and Snap Lines

Müller and Wonka provide the capability to test if components will intersect with other components before placing them to avoid awkward looking intersections. They provide a series of predicate functions to check for such occlusions. See [48] for a full description of this capability. They also provide a capability for rules to "snap" to dominant faces or edges,

providing for improved alignment.

2.3.10 In Practice

There are very few strict rules when writing a split grammar, but Müller and Wonka use some methods which they believe work well in practice.

- Start with a flat axiom and resize it upward. From this point, do not make the mass model larger; that is, "sculpt" the model in such a way as to ensure that the building remains within the confines of its lot.
- Prioritize the rules such that the high priority rules come first, ensuring that the rules are executed in priority ordering. This can also help with level of detail construction.
- As the building is being generated, construct an octtree which can be used for quick occlusion tests.

2.3.11 Limitations

Split grammars are useful even for large variations in architecture style and complexity, allowing the creation of complicated environments such as a model of Rome (Figure 14). They do, however, have some limitations. First, the necessity of injecting 3D models for different types of geometry imposes some non-proceduralism into the system, thus decreasing the flexibility and disabling the ability to represent an entire building as a handful of parameters. Second, it can be slow to generate complex buildings, given that each component may need to be checked against each rule and the number of components can quickly become large. Third, it can be nonintuitive to write the grammar rules. Some knowledge of architecture is required for

realistic buildings; also, for complicated grammars there are a large number of branches in the grammar that must be considered, and the addition of stochastics and predicates compounds this problem.

Figure 14: Rome generated via Parish and Müller's split grammars.[48]

2.4 GPGPU Programming

2.4.1 Modern Graphics Processing Unit (GPU) Description

A modern graphics card will *pipeline* all incoming data. When working with OpenGL, this pipeline is given by Figure 15 [28].

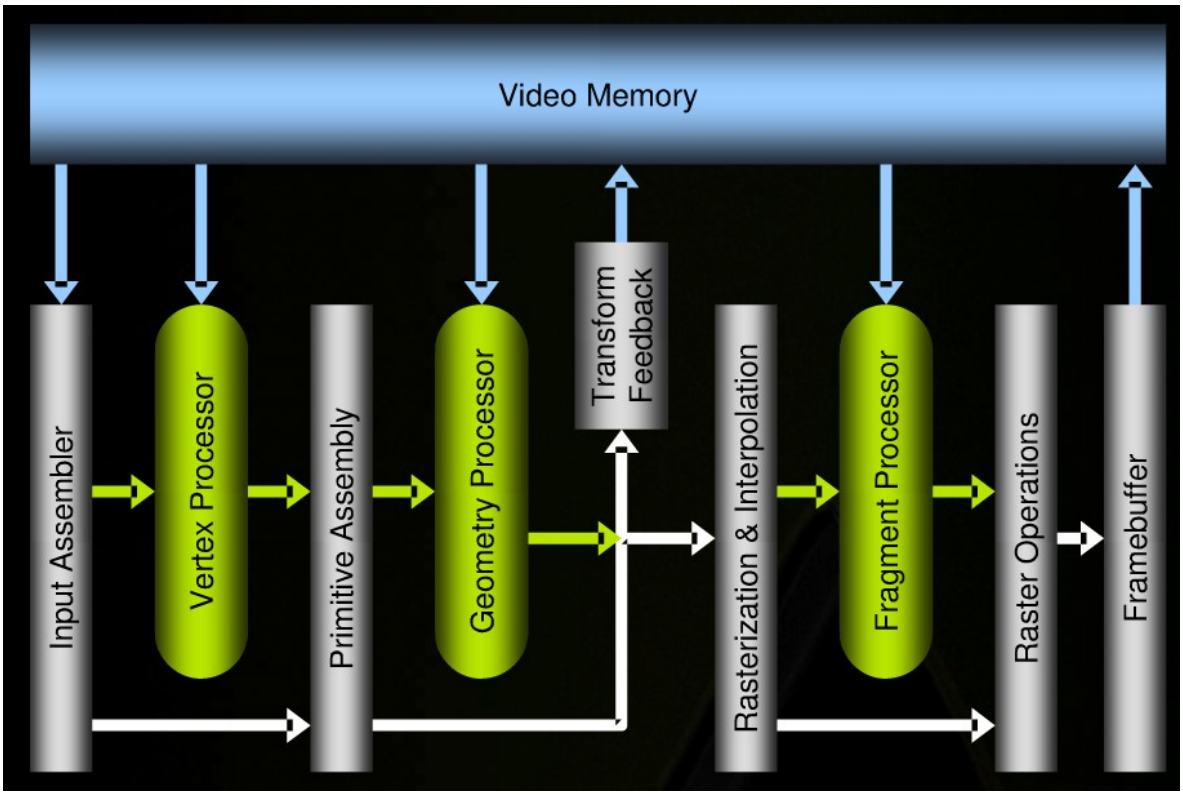

Figure 15: The modern OpenGL graphics pipeline. [28]

All incoming geometry is first sent to the vertex processor, which is responsible for transforming vertices by OpenGL's modelview and projection matrices [5]. These vertices are then sent to the primitive assembly stage, which takes the vertices and constructs OpenGL primitives (triangles, quads, points) based on what the user specified when sending the data. This data is then sent through a geometry processor, a new stage in the pipeline which takes the primitives and does further transformation or constructs new primitives [52]. At this point, the

transformed geometry data can be sent back to the user via the Transform Feedback stage [52], although this is not required.

After going through the geometry processor, rasterization occurs. There are various vertex attributes which need to be interpolated across geometry, such as vertex normals or texture coordinates [4]. This is done before sending the data into the fragment processor, which is responsible for coloring and lighting the individual fragments (pixels) on the screen. The fragments are then altered by any raster operations before finally being output to the framebuffer, which may represent the screen or some arbitrary texture which is the target of the rendering.

Many of these pipeline stages can perform their actions in parallel. For instance, during the vertex processing stage, it is possible to process large numbers of vertices simultaneously, as the vertices are independent of each other. The same is true for the geometry and fragment processing stages. This parallelization is one of the defining features of the GPU, allowing huge speed boosts to a large array of graphics-related processing [20].

2.4.2 Shaders and Shader Programming

2.4.2.1 Overview

In the above diagram, all the green stages of the pipeline are *programmmable.* This means that a user can write a program (termed a *shader*) which will perform the appropriate actions instead of relying on the built-in functionality that those stages provide (which is referred to as the fixed-function pipeline [72]). This programmability provides a greater degree of flexibility, allowing a user to perform actions specific to his needs and even allowing implementation of features which the fixed function pipeline does not support or which are difficult to implement using the fixed function pipeline [68]. By writing shaders, users can

implement such effects as non-photorealistic rendering, fully per-pixel lighting, and parallax mapping, which before were effects difficult to achieve with the graphics card [21].

2.4.2.2 Shader Languages

Each major graphics API (OpenGL and DirectX) has its own language for creating shaders (GLSL and HLSL, respectively) [4]. Since our research uses OpenGL, we will focus on GLSL, though the two languages share similarities.

GLSL is a C-based language containing special data types for vectors, matrices, and samples (textures). It is beyond the scope of this paper to provide full coverage of GLSL; see [68] for a thorough treatment.

2.4.2.3 Vertex Shaders

A vertex shader is performed during the vertex processing stage of the pipeline. A vertex shader occurs once per incoming vertex and is responsible primarily for transforming the vertex into the appropriate coordinates so that the vertex can be later rasterized.

The vertex shader is also responsible for passing on information which can later be used by the fragment processor. For instance, although the vertex shader is not responsible for the application of color or lighting, it must still pass on color and normal information sent in by the user so that the information can be used by the fragment processor (if the fragment processor needs this information to behave appropriately). The vertex shader is free to manipulate or use this information for its own purposes, which can be useful for a large number of different effects [68, 21, 20].

Further, since the vertex shader is only executed once per vertex as opposed to once per fragment, it is often beneficial to calculate things in the vertex shader which do not need to be calculated in the fragment shader and then pass that information on. Doing this can provide a performance increase [4]. As a worst-case example, consider a single triangle that fills the screen. Performing a calculation once per vertex means that that calculation will be performed three times. Performing it once per fragment means that that calculation will be performed for every pixel in the window, which for an application running at an 800x600 resolution is 480000 times.

2.4.2.4 Fragment Shaders

A fragment shader is performed during the fragment processing stage of the pipeline. A fragment shader occurs once per fragment, or pixel, and is responsible primarily for coloring and lighting a pixel.

Fragment shaders are free to receive information sent from the vertex shader and act on it appropriately. For instance, the vertex shader can send through color information, which the fragment shader can then use to color the current fragment. Lighting information, such as normals or light positions/colors, are also commonly sent through. The fragment shader can perform its own calculations on this data to calculate a final color for the fragment.

2.4.2.5 Geometry Shaders

One of the limitations of the vertex shaders is that no new vertices can be generated by their execution. That is, a vertex shader receives one vertex to act upon, and it outputs

information to the fragment shader, but it can not create new vertices itself or discard vertices.

The geometry processing stage overcomes this limitation. In the geometry processing stage, primitives (which are a combination of the vertices processed in the vertex processing stage) can be further manipulated and transformed. Even more important, a geometry shader is able to generate completely new primitives or discard the current primitive.

As an example, consider a triangle. Each vertex of that triangle is transformed by the vertex processor and then assembled into the actual triangle during primitive assembly. The geometry shader then receives the triangle and is free to further transform it, to discard it completely, or to generate new triangles based on the received triangle. The geometry shader could subdivide the triangle into multiple smaller triangles, increasing the geometric complexity of a mesh, or it could reduce a triangle list down to a simpler triangle, decreasing the geometry complexity.

2.4.2.6 Shader Limitations

Shaders are highly parallel, and as a result, certain limitations are imposed on them. First, shaders of the same type can not share information between each other. A fragment shader can not send information which other invocations of the shader can use for their own calculations. Second, vertex shaders and fragment shaders are strictly single input, single output. That is, a vertex shader can not create new vertices, and a fragment shader can not extend to other fragments (fragment shaders can render output to multiple targets, which is a slightly different concept). Third, there is no concept of "global" memory which every shader can read to/write from. Fourth, sending arbitrary blocks of data to a shader is awkward, requiring filling a texture with that data and then reading from that texture within the shader – this will become an

issue when we get into GPGPU programming. Fifth, there is no way to retrieve intermediate calculations within a shader (short of using transform feedback or outputting special fragments), making them difficult to debug.

Beyond the parallel nature, shaders also have other graphics-card dependent restrictions. For instance, older graphics cards do not have hardware support for branching, and dynamic looping is often not supported. Although the nVidia 8800 is supposed to support fully dynamic looping [52], in practice we have found that this is not always true. The performance of certain features (texture accesses, buffer reads) across varying graphics cards is hard to predict, especially with older graphics cards which do not support many newer features.

Shaders do not support pointers, meaning that certain data structures are unnaturally represented in a shader. Linked lists, trees, and other dynamic structures can not be implemented in a standard fashion. Although it is possible to use static arrays to implement many of these data structures, creating large arrays quickly exhausts the memory available to a shaders.

Finally, some problems do not map well to parallel processing [57]. Although often vertex and fragment processing can take advantage of parallel processing, trying to perform more general purpose calculations inside these shaders or calculations which rely on highly sequential ordering may not map well to the GPU programming.

2.4.3 General Purpose Computations on the GPU

2.4.3.1 Overview

Recently, the processing power of the GPU has been utilized to perform more general computations which can take advantage of the parallel vector-processing capabilities of the card [21, 20]. The parallel nature of the vertex and fragment processors allows highly parallelizable

problems to gain a significant speed increase similar to that which can be seen via cluster computing [57], so long as those problems can be overcome within the previously mentioned restrictions of the GPU. In this section, we will provide an overview of a GPGPU programming methodology described in [24], where Göddeke provides OpenGL examples for performing many of the described tasks.

2.4.3.2 Stream Processing

The problems which map most naturally to GPGPU solutions are those which have a number of inputs, where each input can be independently acted upon to produce an output. This type of problem can take great advantage of parallelization, since there is no need to communicate between processes and thus each shader unit can work concurrently.

Stream processing problems differ from more "linear" problems in how they must be approached. Instead of iterating through all the inputs and computing outputs sequentially, stream processors receive their input, and that input is distributed among multiple *cores* or *kernels* which can concurrently work on their own piece of the input. For example, consider a problem where someone wishes to calculate the sine of all the values between 0 and 360. A linear approach might look like the following:

```
for i = 0 to 360
        output(i) = sin(i)
```

Whereas a stream processing approach might simply be composed of

```
Kernel(input)
        return sin(input)
```

In this example, if 360 kernels were present, a single input could be distributed to each kernel, which could then produce its result while all the other kernels are producing theirs, thus effectively reducing the time taken to process all inputs. Note that this is a best-case scenario – not all problems neatly parallelize this way.

Further, there are complications which must be considered in stream processing, such as how the data is distributed (and how fast) and how the results are returned [24]. The following sections will discuss how these problems are addressed in GPGPU schemes.

2.4.3.3 Textures as Input

A way is necessary of sending arrays of data (inputs) into a shader so that those inputs can be acted upon and the outputs computed. The GPU was not designed to take arbitrary arrays of data, but it does have something which can be used: floating point textures. When using a GPGPU scheme, a 2D texture can be filled with inputs (a 1D texture *can* be used, but there are limitations on texture size which make them less useful). This texture can then be mapped onto OpenGL geometry as if it were a standard image texture, and thus the fragment shader will have access to the data stored in the texture.

2.4.3.4 Shaders as Kernels

The native computation unit of the GPU is the shader. Shaders run in parallel and can execute arbitrary code (within limits), making them similar to cluster kernels. Recall that vertex shaders do not have a way of returning their results back to the user. Thus, we are forced to use geometry or fragment shaders to perform our calculations. We will focus here on using fragment

shaders, but later in this thesis we will show a use of geometry shaders.

As has been discussed, fragment shaders are used by the GPU when rendering fragments to the framebuffer. The fragment shader is invoked by each fragment being rendered, and these fragment shaders often perform lighting and color calculations to output a final color to the framebuffer. However, recall that these fragment shaders can perform arbitrary code; thus, instead of calculating color, we can use the shaders to calculate arbitrary information. For each fragment being rendered, we can read an input and calculate an output for that specific fragment.

2.4.3.5 Drawing as Output

The native operation of a fragment shader is to render some final output color. For GPGPU applications, color is very often not the desired output, but we can still use this functionality to obtain the desired results.

In the typical GPU pipeline, a fragment is calculated and rendered to a final framebuffer. We utilize this same behavior but change the meanings of various terms. In a GPGPU scheme, each time a fragment is being processed, it is processed on an input from the incoming texture. That is, each fragment represents one input, calculates its output based on operations in the fragment shaders, and "renders" its final output to the framebuffer. Here, the framebuffer no longer represents an image of colors but instead a buffer of results. Each point in the framebuffer is the result of the calculation performed in the fragment shader.

In practice, this behavior can be achieved in the following manner:

1. Fill a texture with the desired inputs.
2. Render a quad to the screen the size of the texture. Thus, each texture coordinate will be mapped to a single fragment.

3. In the fragment shader, read the texture data at the current texture coordinate, which will be the input for this shader.
4. Perform some calculation on the input data, and determine the final output.
5. “Render” the final output to the framebuffer.
6. Read the framebuffer into an array which the CPU can use more naturally.

Note that newer graphics cards allow the creation of floating-point buffers for use as textures and render targets, thus eliminating the need to represent all the data within buffers that more naturally represent 3 or 4 component color values.

Also note that although we have been using the phrase 'single input' and 'single output' frequently, this is somewhat of a misnomer. Multiple texture inputs can be bound to a single piece of geometry, meaning that a shader can have access to a fixed number of inputs (for instance, a 3-element vector can be represented as one component in each texture). Fragment shaders are also capable of reading from any place in the texture, not simply the current coordinate being processed. Also, there can be multiple output buffers attached to a single framebuffer, meaning that more than one output element can be rendered (again, a 3-element vector can be represented as one component in each render target). What can *not* be done is rendering multiple fragments in one unit, thus allowing a fragment shader to extend the reach of its current fragment, a capability which can be useful when a calculation can produce an arbitrary number of outputs.

2.4.3.6 Feedback

In many applications, a single input-compute-output cycle is not enough. The outputs of one pass often need to be used as inputs into another pass.

One technique for facilitating this need is called “ping ponging.” In this approach, the output of a pass is then bound as a texture for the next pass. The next pass is then computed using the previous pass as input and outputs to a different framebuffer. If further calculations are necessary, the two framebuffers – the input and output – are swapped, such that the first framebuffer is again the target and the second serves as the input for the next pass.

Note that current GPU requirements do not permit using the same texture for input and output at the same time. That is, if a texture is bound as input, it can not be the target of output operations. Thus the need for two textures instead of performing calculations based on the input and then simply outputting to the same place in the same texture.

2.4.3.7 Restrictions

GPGPU programming is complicated by the same restrictions as typical shader development. Unlike in cluster computing where kernels can communicate with each other [55], fragment shaders have no mechanism of communication. There is no capability to lock or block a specific shader unit until another finishes with its output, and there is no real way to predict in what order fragments are processed. There is no global memory that can be written to and read from later, preventing a shader unit from producing some calculation once that can be exploited later in processing. With these restrictions in place, certain algorithms (such as scatter or gather algorithms) are not as naturally represented on the GPU as they would be on a cluster network.

2.4.3.8 GPGPU Libraries

Recently, various libraries have been created to aid in using GPGPU techniques without

worrying about textures, individual shaders, or even graphics programming. Both nVIDIA and AMD provide libraries, CUDA [15] and Close-to-the-Metal [3] respectively, for their own specific graphics cards. Sh [69] and BrookGPU [9] have been introduced to allow GPGPU development regardless of the graphics hardware being developed on. All of these libraries allow using the GPU as a stream processor while abstracting away some of the graphics-programming specific requirements frequently necessary for GPGPU development.

2.5 Rule Learning

2.5.1 Overview

Langley and Simon identify five major paradigms for machine learning: representation of knowledge as a multilayered network akin to neural networks, representation of knowledge based on specific cases or experiences, genetic algorithms, rule induction, and analytic learning [34]. Rule induction, or systems that learn sets of predictive rules, will be our focus when talking about machine learning.

Rule induction systems have a number features that make them useful: rule sets are fairly easy for humans to understand [11], and these systems can outperform decision tree learners [55, 64, 80]. However, these systems often scale poorly in the presence of noisy data [13]. Later, we will show how all of these features will be taken into consideration for procedural building generation.

2.5.2 REP

A common approach to decision tree learning involves an *overfit-and-simplify* approach, where a complex tree is formed that overfits the data, and then this tree is simplified (or pruned) to increase the tree's generalization ability (that is, to reduce overfitting) [64]. *Reduced error pruning* (REP) has been proposed as a technique for tree pruning, and this method can be easily adapted to rule learning schemes [55, 10].

To adapt REP to rule learning schemes, data is first split into two sets: a *grow* and *prune* set. A set of rules is formed via some heuristic method to model the grow set; typically, these rules overfit the grow set. Afterward, this set of rules is repeatedly simplified via pruning

operations, such that the operation chosen is the one that reduces the error rate of the pruning set the most. These pruning operations typically involve deletion (in whole or in part) of a rule. Simplification ends when applying a pruning operation would no longer reduce the error rate on the pruning set.

Although REP for rules can often improve generalization performance on noisy data [55], it is inefficient. In the presence of noisy data, the computational complexity of REP has been shown to be O(n^4) [13].

2.5.3 IREP

To combat the inefficiencies of REP, Furnkranz and Widmer introduced *incremental reduced error pruning* (IREP) [23], a description and extension of which is provided by Cohen in [14]. We will describe the 2-class implementation given in [14] along with the extensions that allow IREP to work with more than two classes.

IREP acts as a separate-and-conquer algorithm, greedily building a rule set. When a rule is built, all examples covered by the rule are deleted and the process is continued until no positive examples remain or the generated rule has an unacceptable error rate.

To build a rule, IREP first partitions all uncovered example into two sets, a *growing* and *pruning* set. A rule is then grown: [14] uses a version of Quinlan's FOIL algorithm [6], which follows:

A rule R begins with an empty conjunction of conditions. The algorithm then considers adding conditions of the form An = v (if An is a nominal attribute), Ac <= theta, or Ac >= theta (if Ac is a continuous attribute). Conditions which maximize FOIL's information gain are added until the rule covers no negative examples from the *growing* set.

Once the rule has been grown, it is then pruned. Cohen deletes any sequence of conditions from the conjunction such that the following function is maximized:

v(Rule, PrunePos, PruneNeg) = (p + (N-n) / (P+N))

where P is the total number of examples in PrunePos, N is the total number of examples in PruneNeg, and p and n are the number of examples in PrunePos and PruneNeg respectively that are covered by the rule. Conditions are repeatedly removed until no removal would increase the value of v.

To extend this algorithm to work with multiple classes, Cohen proposes the following scheme [14]. The classes are ordered by prevalence, where the first class has the lowest prevalence and the last class has the highest. IREP is used to separate the first class from the remaining classes. All instances covered by the learned rules are then removed. IREP is then used to separate the next class from the remaining classes. This is done continuously until only one class remains, which is designated as the default class.

Cohen also proposed an extension to allow the algorithm to work with missing attributes. If a test involves an attribute A, that test is defined to fail if the instance has a missing value for attribute A.

Cohen showed that IREP is considerably fast; however, its generalization performance was not ideal, frequently performing worse than C4.5rules [14]. To combat this, Cohen introduced IREP*, which leads naturally into RIPPER [14].

2.5.4 IREP*

IREP* is an extension to IREP with two critical changes: the rule-value metric during the pruning phase is modified, and a new stopping condition is given for when to stop adding rules

to a set.

The modified rule-value metric addresses a problem where IREP can fail to converge when the number of examples increases. The new metric is

v*(Rule, PrunePos, PruneNeg) = (p-n) / (p+n)

which provides more reliable results.

The modification of the stopping heuristic avoids a problem wherein rule generation stops prematurely. That is, IREP stops adding rules when the generated rule has an unacceptable error rate, which (depending on the error rate – Cohen uses 50% in his implementation) can result in the algorithm stopping too soon. This is often the case when learning a rule set containing a large number of low-coverage rules.

To combat this problem, Cohen modified the stopping condition to hinge on the *description length* of a rule set and examples as opposed to error rate, where the description length is the estimated number of bits required to represent the theory. When the description length becomes *d* bits larger (Cohen uses d=64) than the smallest description length obtained so far or there are no more positive examples, IREP stops adding rules. Afterward, rules are deleted to reduce the total description length and simplify the rule set.

2.5.5 RIPPER

To improve IREP*'s incremental approach to reduced error pruning, a new method was introduced as a postprocessing step for the rules generated by IREP*. Given a rule set generated by IREP*, R1,...,Rk, the postprocessor processes each rule sequentially in the order they were generated. For each rule, Ri, two alternate rules are generated, a *replacement* and a *revision*. The replacement (R'i) is an entirely new rule formed by a typical grow-then-prune step, such that

pruning is intended to minimize the entire rule set R1,...,R'i,...,Rk. The revision is similar, except that it is generated by greedily adding conditions to Ri as opposed to generating an entirely new rule. The postprocessor than decides which of the three rules (the replacement, revision, or original) to keep based on the description length. After the entire rule set is optimized, IREP* is used again to add any additional rules necessary to cover any remaining positive examples.

The above steps have been dubbed RIPPER. An iteration can be applied where the above steps are performed on the output of RIPPER to gain further optimization. This is referred to as RIPPER2; RIPPERk is a more general name for the algorithm that repeats this optimization *k* times [14].

Cohen has shown that RIPPER improves the performance of IREP* and performs very well in comparison to C4.5 and C4.5rules. RIPPER is also efficient even in the presence of noisy data sets, typically operating much faster than C4.5rules and scaling better as the number of training examples increases [14].

Chapter 3

3 Improving the Speed of Building Generation

3.1 Overview

Generating buildings using split grammars can quickly become prohibitively slow. Even for relatively simple buildings generated from a small number of rules, the number of components which need to be processed can quickly explode. A naïve implementation which must constantly reinterpret rules (via regular expressions or some other mechanism) and perform string manipulations would compound the problem.

The current speed of building generation requires that, for large scenes, most buildings must be generated in an offline pass and stored as models which can be more quickly imported. This renders one of the main advantages of procedural content generation – the ability to succinctly define content in a much smaller space than is necessary when storing a full description of that content – meaningless. It also means that split grammars can no longer be used for dynamic level-of-detail generation, since the time to go from lower resolution to higher resolution meshes can be a bottleneck on the system. Thus, it is necessary to improve these speeds.

In this section, we will show two steps we have taken to increase the speed of building generation without constraining the grammar or preventing the ability for grammars to be dynamically loaded from files. First, we will introduce a preprocessing step that can be run over

the grammars such that they can be efficiently interpreted. The preprocessing results can be easily serialized into a format such that preprocessing need only be done once on a grammar as opposed to every time the grammar is loaded.

Second, we will introduce a technique for generating buildings fully on the GPU. We will take advantage of the parallel nature of the GPU to gain speed increases between two and ten times faster than without the GPU for a single building. We will also show how multiple buildings can be generated concurrently on the GPU, potentially increasing the speed benefits when GPU memory restrictions permit. For all of this, we will be taking advantage of the most recent consumer-level graphics card technology, including geometry shaders and transform feedback [52].

3.2 Grammar Preprocessing Using Lex/Yacc

3.2.1 Overview

Although it is possible to use a complicated series of regular expressions and interpret a grammar rule each time it is being evaluated, which was one of our first approaches to implementing split grammars, this is slow. It requires a large amount of string processing which can be time consuming, especially for complicated rules. Further, it is difficult to implement this approach while still supporting all of the functionality of the grammars.

We have come up with a scheme using Lex and Yacc [2] to completely bypass the problem. We introduce a preprocessing phase wherein the grammar is run through a Lex/Yacc interpreter, where the output is a series of structures that can be more quickly interpreted. A full description of the internal mechanics of Lex and Yacc is beyond the scope of this thesis; see [2]

for a treatment of the two and compiler construction in general.

Unfortunately, we have no frame of reference for comparing our implementation with other implementations, since we have found no publicly available implementations of split grammars. We do know that our approach is considerably faster than our own earlier versions and does not lose the ability to dynamically load grammars from files (that is, we do not need to hard-code rules).

3.2.2 Lexical Analysis with Lex

To properly interpret the grammar, we must break the text up into a series of individual units (tokens) that each have some value. To do this, we must define a series of tokens which Lex can use for its lexical analysis. We identified the following tokens:

FUNCNAME	:	The name of a function (Subdiv, Comp, Size, and so on)
WORDVAL	:	A word in the grammar; a word must start with a letter, but then may contain letters, numbers, underscores and periods.
FLOATVAL	:	A floating point number; all numbers in our system are treated as floating point, as we do not need the precision of integers
^	:	a character to indicate that a number represents a relative value as opposed to an absolute value.
+ - / * < > = !	:	Mathematical operators, since grammars can make heavy use of these operations.
->	:	Separates the name of a component in a grammar with the rule that operates on that rule.

This list is not completely comprehensive. There are other tokens, such as parentheses and braces, which we did not explicitly mention but which are still important for appropriate parsing of a grammar. See Appendix A for the full body of the Lex file.

3.2.3 Yacc

Once Lex has performed its analysis on the body of the grammar and separated the tokens, Yacc can then properly interpret the grammar and fill the appropriate data structures. Yacc uses a series of grammar rules when interpreting a file conforming to the grammar; upon recognizing these rules, it performs some code, where in our case the code fills out appropriate data structures described in the next section. The grammar rules themselves follow in Figure 16:

```
PROGRAM    :        RULES '@'
        RULES      :        RULES RULE
                            RULE
        RULE       :        FLOATVAL ':' WORDVAL '~' ACTIONLISTS
                            FLOATVAL ':' WORDVAL ':' EXPRESSION '~' ACTIONLISTS
        ACTIONLISTS:        ACTIONLISTS '~' ACTIONLIST ':' FLOATVAL
                            ACTIONLIST ':' FLOATVAL
                            ACTIONLIST
        ACTIONLIST:         ACTIONLIST ACTION
                            ACTION
        ACTION     :        FUNC '{' REPLACEMENTS '}'
                            FUNC
        REPLACEMENTS:       REPLACEMENTS '|' WORDVAL
                            WORDVAL
        FUNC:               FUNCNAME '(' PARAMS ')'
                            FUNCNAME
        PARAMS:             PARAMS ',' EXPRESSION
                            EXPRESSION
        EXPRESSION:         '(' EXPRESSION ')'
                            EXPRESSION '*' EXPRESSION
                            EXPRESSION '/' EXPRESSION
                            EXPRESSION '+' EXPRESSION
                            EXPRESSION '-' EXPRESSION
```

```
                    EXPRESSION '>' EXPRESSION
                    EXPRESSION '<' EXPRESSION
                    EXPRESSION '=' EXPRESSION
                    EXPRESSION '!' EXPRESSION
                    OPERAND
OPERAND:            WORDVAL
                    FLOATVAL '^'
                    FLOATVAL
                    FUNC
```

Figure 16: Our entire Yacc grammar for parsing split grammars.

There are a few important things which must be noted in the grammar:

1. The grammar is *not* intended to enforce complete correctness. There are some qualities of an incorrect grammar which may be accepted by the grammar but which will still result in unexpected results. The grammar is intended to preprocess for speed alone and is not intended as the formalization of correct split grammars.
2. Special Yacc functionality is used to ensure that order of operation in mathematical expressions is preserved. This is not only very important for the mathematical accuracy, but it also allows us to store the expressions in a way that they can be easily and quickly computed later.
3. Although the rule ID is technically an integer, in the grammar we denote it as a floating point number. This was simply a matter of convenience.
4. This grammar does lose one of the capabilities mentioned in [48]. In the original presentation of split grammars, replacements themselves could be functions that spawned

other replacements. We remove this to simplify interpretation somewhat, and also because it isn't strictly necessary; the same functionality can be achieved by providing an intermediary replacement which is later acted upon with the desired function.

5. Notice that expressions can take the full range of mathematical operations, and operands can also be functions which themselves have expressions as parameters. Although this adds a greater degree of complexity, it is essential to maintain the full amount of expressibility.
6. The syntax above does not perfectly conform to Yacc standards. Further, we do not present the exact code that is triggered when a rule is interpreted. See Appendix A for the full Yacc file listing.

3.2.4 Filling the Data Structures

As Yacc interprets the grammar, our code fills out a series of data structures which fully represent the input grammar. These data structures are designed to store all the information in an easily navigable way, increasing performance. What follows is a description of our structures, starting with the topmost structure (a list of rules) down to a structure identifying specific items in an expression.

The RuleList structure is the top-level structure that stores all of the rules:

```
RuleList
        rules : list of Rule
        numRules : integer
```

The Rule structure contains all the information for a given rule, including its id, the name

of the building component the rule should act on, a predicate (if one is provided), and any actions that should occur when the rule is triggered:

Rule
id : integer
predecessor : string
requirement : Expression
actionLists : ActionLists

The ActionLists structure contains a list of all the ActionList structures for a given rule. This is somewhat non-intuitive, but was an implementation requirement.

ActionLists
list : ActionList
numLists : integer

The ActionList structure actually contains the list of actions, along with the probability that this specific action list will activate in the event that the grammar specifies a stochastic probability.

ActionList
actions : list of Action
numActions : integer
probability : float

The Action structure contains the function (Size, Repeat, etc) that should be used when the action activated. It also contains a list of replacements in the event the function requires that the component be replaced with other components.

Action
function : Function
replacementList : ReplacementList

The ReplacementList structure, as its name implies, stores all of the replacements which may be used by an action.

ReplacementList
numReplacements : integer
replacements : list of strings

The Function structure stores all the necessary information for a function – an identifying integer which determines the actual function to be used and a list of parameters:

Function
id : integer
paramList : ParamList

The ParamList structure stores a list of parameters for functions, which are all comma delimited in a grammar file.

ParamList
params : list of Expression
numParams : integer

The Expression structure represents a mathematical expression. Although expressions in a split grammar are in infix notation, expressions in our program are stored in postfix notation. This allows for quick calculation of an expression when a rule is being executed.

Expression
items : ExprItem
numItems : integer

The ExprItem structure represents a single item in a mathematical expression. This could be a value, an operator, a word (for a variable), or a function. The structure is designed to store any one of those items (the structure also stores which type of item it contains).

ExprItem
type : enumeration { EI_FLOAT, EI_VARIABLE, EI_RELATIVE, EI_FUNC }
op : character
val : float
name : string

```
function : Function
```

These are all the structures which Yacc fills out as it interprets the grammar. There is a small amount of extra information, and also Yacc implementation details make some requirements of the structure not presented here. See Appendix B for the source file containing the structures themselves.

3.2.5 Interpretation of the Structures

Once the data structures have been filled by Yacc, interpreting them for building generation is relatively straightforward. When processing a component, search for a rule whose predecessor matches the component being processed and whose requirement evaluates to true. At that point, choose an action list (if probability is used, a random number will have to be generated from a uniform distribution to determine which action list to execute). Execute each function in the action list (where the ID of the function will map back to the specific action desired); the code for executing functions may need to use the replacement list to add replacements into the working list of building components. Repeat this process for each building component until there are no more building components to process.

Evaluating expressions for function parameters deserves special mention, since it is not entirely straightforward. When being stored, the grammar will store the expressions in postfix notation. This means that the entire expression can be evaluated via a single pass through the expression list with the use of an extra stack according to the following pseudo code in Figure 17:

```
Given stack opStack, list items
        for each item in items
                if item.op == 0          //not an operand
                        push ExprItemValue(item) onto opStack
                else
                        item1 = pop item off opStack
                        item2 = pop item off opStack
                        perform desired operation using item1 and item2
                        push result on opStack
                end
        end
        final result = pop item off opStack
```

Figure 17: Algorithm to calculate an expression in postfix notation where each expression item is stored on a stack.

Thus, whenever a parameter needs to be evaluated, we can use the above code. This code works equally well if boolean expressions are incorporated, thus we can evaluate whether a predicate is fulfilled or not by representing the predicate as simply an expression.

3.2.6 Serializing the Structures for Later Use

These structures could trivially be exported to a file such that the preprocessing need only be done once, thus saving time in the future. If size is not a concern and the loading speed penalty is acceptable, an XML file can be generated by iterating through the structure lists, starting at the top level, and filling out the appropriate XML information. An example of a very simple file with a single rule that resizes a component to 20x10x10 could look like that shown in Figure 18 below.

```
<RULELIST>
        <RULE>
            <ID>0</ID>
            <PREDECESSOR>lot</PREDECESSOR>
            <ACTIONLISTS>
                <ACTIONLIST>
                    <ACTION>
                        <FUNCTION>
                            <ID>2</ID>          //2 = size
                            <PARAMLIST>
                                <EXPRESSION>
                                    <EXPRITEM>
                                    <TYPE>FLOAT</TYPE>
                                        <VAL>20.0</VAL>
                                    </EXPRITEM>
                                </EXPRESSION>
                                <EXPRESSION>
                                    <EXPRITEM>
                                    <TYPE>FLOAT</TYPE>
                                        <VAL>10.0</VAL>
                                    </EXPRITEM>
                                </EXPRESSION>
                                <EXPRESSION>
                                    <EXPRITEM>
                                    <TYPE>FLOAT</TYPE>
                                        <VAL>20.0</VAL>
                                    </EXPRITEM>
                                </EXPRESSION>
                            </PARAMLIST>
                        </FUNCTION>
                    </ACTION>
                </ACTIONLIST>
            </ACTIONLISTS>
        </RULE>
    </RULELIST>
```

Figure 18: Example of split grammars being transformed into XML.

Alternatively, it is similarly easy to write an equivalent flat-text ASCII file that conveys the same information and is faster to load than the XML version. If both size and speed of loading are a consideration, it is a straightforward extension to convert the structures to their equivalent binary representation and store that in a file.

3.3 Shifting Generation onto the GPU

3.3.1 Overview

To speed up the process of procedurally generating buildings, we can take advantage of the parallelization capabilities of the GPU. We observe that operations on building components can be treated independently; that is, to act on a building component, we do not need to have information concerning the other components of the building. Thus, we can clearly see that we can operate on the components in parallel.

To take advantage of the parallelization, we will use geometry shaders. Recall that geometry shaders allow an arbitrary number of outputs (within the restrictions set by the GPU). We require this behavior since operating on a component may require the spawning of an arbitrary number of new components; for instance, the Repeat operation may spawn multiple new components. Thus, performing generation in the fragment shader would be impractical.

There are various practical implications to consider, since even with geometry shaders the problem does not map perfectly to the GPU. To maintain generality, we require the ability to send an arbitrary grammar rule set to the GPU, which itself is not a straight-forward process since there is no clear mechanism to send our structures to the GPU or store them in global memory. We also need to ensure that when new components are spawned, they can be acted upon by the grammar rules, a requirement which makes it impossible to simply send our data over to the GPU and let the shader handle everything.

In this section we will show how we approached these practical considerations, and in doing so we have achieved significant speed improvements to building generation which we

hope will become even larger as geometry shaders mature.

3.3.2 Sending Rules to the GPU

As we stated previously, sending grammar rules to the GPU is a non-straightforward process. In the ideal case, we would be able to send our Lex & Yacc generated structures across and simply have a geometry shader act similarly to the CPU implementation, for each component iterating through the rules and determining the appropriate action. However, the GPU provides no facility for achieving this behavior – not only can we not send our structures across naturally, there is no place to globally store the structures such that all instances of the geometry shader would have access to them.

Initially, we considered serializing the structures into a texture and then deserializing the structures as they were being used. This solution, however, would be fairly complicated and would require a large number of texture reads, both of which would considerably slow down the performance of our building generation.

Instead, we have opted to dynamically generate a geometry shader which represents a rule set. Upon loading a rule set and preprocessing it, we interpret the Lex/Yacc structures to generate a geometry shader source file which can then act on components. When the shader receives a component, this code is called. Through this process, each rule set is transformed into a geometry shader which emulates those rules. Every time a rule set is loaded, a new geometry shader must be generated and sent to the GPU.

The observation that makes this approach valid is as follows: although each grammar rule is represented by its own syntax within a split grammar file, these rules can essentially be mapped to rather simple “if-then” statements. That is, for a rule that acts on component L and

has action list A, we can generate the following code:

```
if current_component == L
        perform A
```

Therefore, for each rule in our structure, we can transform the rule into similar if-then statements. We must first do a small amount of preprocessing on the rules: since GLSL provides no functionality for strings, we must give a unique integer ID to each component name and use that within the code instead.

The actions must be mapped to GLSL code that performs the appropriate action and outputs new components. For this, we can simply write a function for each action in GLSL which corresponds to the CPU equivalent. In the dynamically generated portion of the shader, we call those functions in the appropriate places (in the conditionals where those actions should take place). The parameters to each function can be mapped back to their original infix notation – that is, they will be executed as they normally would within the body of the code.

As an example, consider the following rule:

```
1: lot->Size(10,20,10)
```

This rule can be mapped to the following geometry shader code:

```
if (current_component == 0) {
        Size(vec3(10,20,10), vec3(0,0,0))
}
```

In the above example, Size is a function we wrote beforehand which simply emulates the CPU equivalent. At first this function may seem unintuitive, but consider that we no longer have access to the Lex/Yacc structures, and thus all information must be passed into the function explicitly. The first vector represents the desired size, and the second vector represents whether

those sizes are relative or not.

Things become a bit more complicated as replacements are necessary. Consider:

```
1: lot->Subdiv(X, 1^, 2^) {left | right}
```

This code may be mapped to the following code:

```
if (current_component == 0) {
        sizes[0] = 1;
        sizes[1] = 2;
        rel[0] = 1;
        rel[1] = 1;
        reps[0] = 1;
        reps[1] = 2;
        Subdiv(AXIS_X, 2);
}
```

Here, we have predefined three arrays: *sizes* stores the sizes to be used when subdividing, *rel* stores whether those sizes are relative or not, and *reps* stores the components to be used as replacements. Internally, Subdiv uses all this information to perform its operation and it creates the necessary new components. The only two parameters are the axis to subdivide along and the number of subdivisions to make. Although we could have written Subdiv to pass the arrays in as parameters, we chose to keep the global to simplify our functions.

In practice, we have found that all the desired operations in the split grammar (with some limitations) map to similar code.

When all the rules are placed in one geometry shader file, they form an if-then chain such that if none of the conditionals is entered, we know that none of the rules applied and that this is

a final component. Note that there are other considerations, such as whether probabilities are associated with certain action lists, that must be considered when transforming the structures.

Figure 19 is the psuedo-code that handles transformation of the structures to GLSL code:

```
for each Rule rule in RuleList
        output "if (id ==" + MapToInt(rule.predecessor) + ") {"
        if (rule has a requirement)
                output "if (" + WriteExpression(rule.requirement) + ") {"
        if (rule.actionLists.lists[0].probability > 0)
                output "float prob = rand(0.0,1.0);"

        let probSum = 0;

        for each ActionList al in rule.actionLists
                if (rule.actionLists.lists[0].probability > 0)
                        probSum = probSum + al.probability
                        output "if (prob <= " + probSum + "){"
                for each Action a in al
                        TranslateFunction(a.function.id)
                if (rule.actionLists.lists[0].probability > 0)
                        output "return;"
                        output "}"

        if (rule has a requirement)
                output "return; } }"
        else
                output "return; }"
```

Figure 19: Algorithm to convert a split grammar into GLSL shader code.

The meanings for the functions used in the above psuedo-code follow:

MapToInt - finds a unique integer for the predecessor, or returns one if it has already been assigned

WriteExpression- writes an expression out in infix notation so that it can work exactly like normal code

TranslateFunction- generates the appropriate code to call a function, writing any necessary parameters or filling any necessary arrays; this is a

relatively large function, since it must handle all the different actions that might be called

When combined with a "basis" file that defines all the functions and creates all necessary global variables to achieve the functionality necessary by split grammars, this output generated by the above pseudo-code will form the entirety of the geometry shader file that can then be sent to the GPU to handle building processing.

3.3.3 Representing and Sending Components

The code generated via transforming the structures can act on components, but we still need some way of representing the components in a geometry shader and outputting components.

The native input of the geometry shader is a primitive – a point, triangle, triangle list, etc. Obviously, none of these primitives maps directly to a building component. However, we can manipulate the meanings of certain aspects of a primitive to represent our component.

We chose to represent each component as a single point. Typically, a point can have any number of parameters associated with it – a size, a color, a texture coordinate, a normal, and so on. We use these parameters to store relevant data to us; that is, we change the meanings of the parameter but send them to the GPU in the same fashion.

The position, color, and texture coordinate of a point can all be represented as a 4-component floating point vector. In the first three components of the position, we store a building component's position. In the fourth component we store the component's unique integer identifier. In the first three components of the color, we store the size of a building component. In the fourth component we store any texture/model ID's that may apply to the component (for

instance, if later we want to replace the component with a 3D model, we store a unique integer ID of that model in this position). In the first three components of a point's texture coordinate, we store the rotation of a building component. In the fourth component, we store extra information that is not bound to the building but can still help us with generation (this position will be very important when we discuss random number generation).

Although packing the information so tightly may be somewhat counter-intuitive, it is important that we efficiently use all the memory provided to us. This is because geometry shaders are limited in the number of floating-point values they can output via transform feedback, and also because the data may have to travel between the CPU and GPU; in this case, sending less information will impact performance positively. It is possible that we can achieve the same result by using our own defined vertex attributes, but our approach is simpler.

In this manner, each component is represented. We store all the building components as points in a vertex buffer object (VBO) [6]. Initially, only a point representing the initial axiom is stored in this VBO, but later the VBO will store building components which are generated by the geometry shader.

When we want to send the components to the GPU to be acted upon, we simply draw the contents of the VBO as points. The points will first be processed by our vertex shader, which only serves to pass the components on to the geometry shader. The points will then be processed via our geometry shader (one instance of the shader per point), where the GPU will handle distribution/parallelization. The shader will act on its own component, and will often spawn new components, which will be sent via transform feedback into another VBO meant for receiving incoming points. We need two VBOs, since a VBO can not be designated as a source for rendering and a destination for transform feedback at the same time.

3.3.4 Feedback Loop

The process given above only works for a single pass over the components. It will not operate on new components created as the result of processing done in the geometry shader. Thus, we need to run another pass over the data to act on those components, and a pass to act on new components as a result of that pass, and so on. We achieve this via the earlier described process "ping-ponging."

We use two VBO's, V1 and V2. Before the first pass, we fill V1 with the initial set of building components – the axiom. We bind V2 as the target for transform feedback, essentially designating that all output of the geometry shader should be routed into V2. We then draw with V1, and all the points in V1 are processed via the geometry shader and the resulting components are placed in V2.

In the second pass, we swap the roles of V1 and V2. V1 is designated as the target for transform feedback, and V2 is used for drawing. In this manner, all the components that were created in the first pass are processed, and the result is stored in V1.

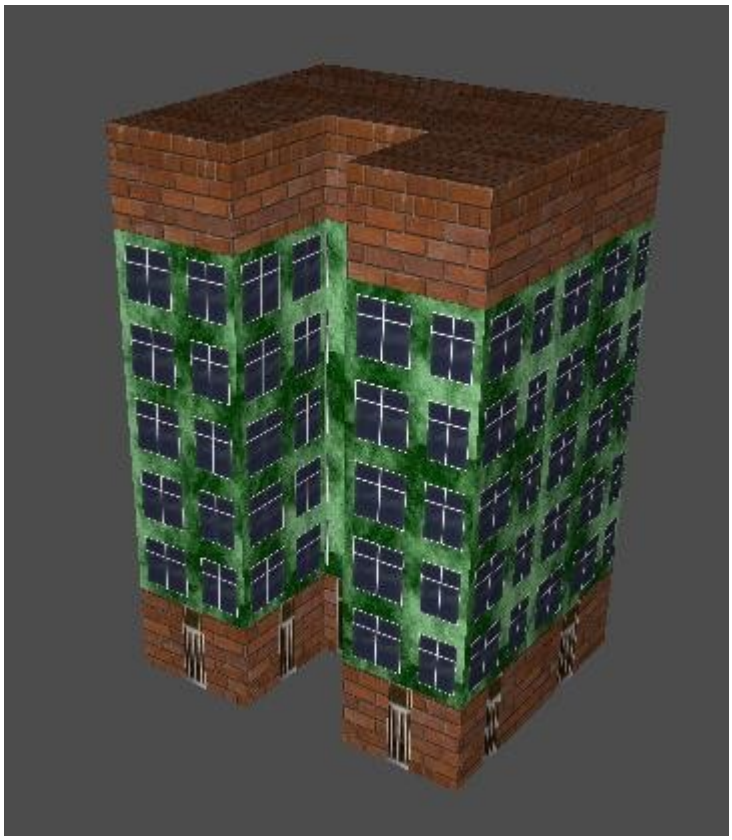

Figure 20: Simple building generated from our implementation of split grammars.

We continue swapping the roles of V1 and V2 until no rules are activated, which is equivalent to the CPU-based practice of

continually applying rules until no rules can apply (all the components have been fully processed) , indicating that we are finished processing. In practice, this is not a straightforward process – since all the geometry shader units are acting independently, there is no clear way for them to decide when no rules applied and the process should finish. Thus, we continue the feedback loop for a fixed number of iterations. The number of iterations can be specified by the user of our system, though often the number can be equal to the number of grammar rules.

It should be noted here that in our system, terminal components are *not* removed from the VBO and are continually processed in later phases of the feedback loop. We handle this by setting a terminal component's ID (its fourth component in the position vertex attribute) to a special ID that denotes terminal. If, in the geometry shader, the shader is acting upon a terminal component, it skips all processing and outputs the component as-is. Thus, when processing is finished, the VBO should be filled with terminal components. Although sending data which is not processed is wasteful, this is better than the alternative of removing the components from the VBO, which can be a very costly operation involving the reading and traversal of the entire VBO after each feedback loop.

3.3.5 Finalizing Geometry

Once the feedback loop has finished, we can read our data from the VBO and construct the final geometry of the building. Each point in the VBO represents a single building component. Thus, we can iterate through each point in the VBO and construct a full component out of that point. We can also replace appropriate points with textured components or full 3D models as necessary.

For simple components, it is possible that we can construct another geometry shader

which would output the final building geometry to a VBO, thereby completely avoiding having to read the VBO storing building components from GPU memory. However, this is not possible if we must inject 3D models into the final geometry, since the geometry shader will have no way of inserting that model.

3.3.6 Implementation Details

There are some limitations of modern GPUs that impact the implementation of our system. First, dynamic looping on the GPU is not very well supported. Dynamic looping is critical in the Repeat function and important for Subdiv. Unfortunately, we have found that using a fully dynamic counter based on a vertex attribute tends to cause the geometry shader to hang indefinitely. We have had more success with providing a constant upper limit for the loop and breaking early, but even this has been problematic in cases where that upper limit must be large. This limitation will hopefully be removed as GPU drivers advance.

Second, the GPU has no reliable method for generating random numbers. Although the specifications for OpenGL 2.0 provide a series of noise functions to generate random numbers, current nVidia implementations always return 0 [33]. To work around this, we implemented a linear congruential random number generator [78]. We pass an initial seed in with the first building component to use, and then for each new component we generate a new seed from the sum of the current seed and the sum of a series of random numbers to decrease predictability.

Third, the number of building components that can be generated is constrained by how much memory the GPU has, which is typically much smaller than a PC's memory. The nVidia 8800 GTX has 768MB of memory, though some of this will be used by the default framebuffer, among other things.

3.3.7 Performance

To test our system, we created a set of 11 grammars of varied length/complexity, some of which could be used for buildings while others were artificially created solely for testing purposes. They were all kept purposefully small to ensure that the generated shader would be able to fit within the GPU's memory limits and that the number of required looping instructions would be kept to a minimum, as frequent looping within the generated grammar rules often causes the shader to hang. An entire listing of these grammars can be found in Appendix C. Table 1 contains some useful information about the grammars.

All tests were performed on a Dell with a single Pentium Core 2 quad-core processor (2.4 GHz), 2 GB DDR2 SDRAM, and nVidia GeForce 8800 GTX graphics card.

Grammar	Number of Rules	Number of Dec. Points	Description
1	12	4	A simple building with a main section on the bottom and a series of windows along the top
2	14	6	A building which may have a 'wing' branch that operates as its own building with its own door/windows
3	11	2	Skyscraper with a variable number of floors (and thus windows)
4	11	6	Artificial building meant simply for testing purposes
5	9	6	More complex artificial building meant for testing purposes
6	16	3	Building divided into three sections, the top and bottom of which contain artistic extrusions
7	21	4	Heavily windowed building with a branch on each side
8	11	5	Building broken into five different sections of different sizes, each covered in windows
9	20	3	An apartment complex divided into floors each with their own balcony
10	14	7	Based on an example in [48]
11	18	16	Extremely simple mass model containing a large number of decision points

Table 1: Information regarding our 11 test grammars.

The simplest test we ran involved generating a building using a grammar 100 times – 50 on the CPU and 50 on the GPU. Each time a building was generated, we would calculate how long that building took to be generated. At the end, the results were averaged. The results are shown in Table 2.

Grammar	Average Generated Components	Average Time/Building (ms) (CPU)	Average Time/Building (ms) (GPU)
1	140.94	12.18	21.54
2	288.4	22.78	24.02
3	1824	126.68	41.8
4	1334.37	93.9	34.64
5	5725.2	357.88	83.92
6	2107	158.18	48.36
7	2036.45	187.22	53.04
8	5993	446.16	84.24
9	3968	279.56	83
10	750.54	89.86	37.12
11	2.76	3.12	24.96

Table 2: Times for generating a single building on the CPU and GPU based on our test grammars.

Table 2 clearly shows that, for a moderately large number of generated components (greater than 300), there is a distinct difference in the amount of time generation takes on the CPU versus the amount of time generation takes on the GPU, on average by a factor of 2.68.

It would be fallacious to plot a graph correlating the number of components generated to the average time of generation, as the grammars themselves are structurally different. Given that

some rules lead to the generation of multiple components which can then be processed simultaneously via multiple shader units whereas other rules have a more linear processing approach, we can not simply infer speed differences based on the number of components a rule set generates.

We next wanted to see the performance of generating multiple buildings simultaneously on the GPU versus generating them linearly on the GPU. Tables 3, 4, and 5 show the results when generating 9, 16, and 36 building cities respectively. We expected that, when multiple buildings were generated concurrently, the time delta for generation between the CPU and GPU generation would increase. Dashes in the chart mean that too many components were being generated – too many to store in memory – and a valid comparison could not be obtained.

Grammar	Time/City (ms) (CPU)	Time/City (ms) (GPU)
1	94	62
2	140	125
3	1154	874
4	858	561
5	3354	2652
6	1451	998
7	1544	764
8	3853	2715
9	2606	1950
10	748	530
11	15	16

Table 3: Times for generating a 9 building city in both the CPU and GPU.

Grammar	Time/City (ms) (CPU)	Time/City (ms) (GPU)
1	156	110
2	359	203
3	2059	608
4	1528	390
5	-	-
6	2605	671
7	2840	749
8	-	-
9	4649	1357
10	1482	827
11	15	32

Table 4: Times for generating a 16 building city in both the CPU and GPU.

Grammar	Time/City (ms) (CPU)	Time/City (ms) (GPU)
1	375	109
2	717	234
3	4649	1217
4	3354	843
5	-	-
6	5788	1529
7	6490	1482
8	-	-
9	-	-
10	3167	1591
11	16	32

Table 5: Times for generating a 32 building city in both the CPU and GPU.

The next table, Table 6, provides a comprehensive summary of how much speedup occurred under the various conditions.

Grammar	1 Building	9 Buildings	16 Buildings	32 Buildings
1	0.57	1.52	1.42	3.44
2	0.95	1.12	1.77	3.06
3	3.03	1.32	3.39	3.82
4	2.71	1.53	3.92	3.98
5	4.26	1.26	-	-
6	3.27	1.45	3.88	3.79
7	3.53	2.02	3.79	4.38
8	5.23	1.42	-	-
9	3.37	1.34	3.43	-
10	2.42	1.41	1.79	1.99
11	0.13	0.94	0.47	0.5
Average	2.68	1.39	2.22	3.12

Table 6: Comparison of how much speedup was gained from performing generation on the GPU, given by cpuTime/ gpuTime. Numbers less than 1 mean that there was a slowdown.

We can see from the above table that our initial intuition was not correct. For cities of modest size (9 and 16 buildings) we actually noticed a *decrease* in speed gain. That is, had we generated those buildings individually, we would have encountered more of a speed increase than trying to generate the buildings simultaneously.

Though we are not entirely sure of the reasons, we can try to explain the results as follows. Internally, the GPU must perform some sort of memory management and must coordinate transform feedback in a way as to prevent outputs from overwriting each other. When dealing with a large amount of data, this process can negate the improvements we might see from generating the buildings concurrently. It is only when the number of buildings being generated becomes larger (32 buildings) that the benefit of concurrent generation outweigh the tasks the GPU must perform.

Thus, we can determine that for cities of moderate size, it is better to generate the buildings individually as opposed to attempting to generate them all simultaneously. Only when the number of buildings being generated becomes large will generating them concurrently be useful. However, at this point generating them concurrently may become infeasible because of memory limitations.

Next we explored how the structure of the grammar contributed to the speed increases. It was our hypothesis that the more "parallel" the structure was, the more efficient it would perform.

Recall that the GPU will handle distribution of the component data to the individual shader units, and that our building generation requires that the data be sent in multiple passes to fully allow a rule set to operate. It was our belief that the more the GPU could do concurrently, the greater the speed benefit over a linear approach.

To test this, we developed a series of artificial grammars that facilitated varying degrees of parallelization. We developed three grammars, named "2^n", "3^n", and "4^n", which would simply subdivide a component into a certain number of components. For example, 2^n operated as follows:

1: lot->Subdiv(X,1^,1^) {a|a}

2: a->Subdiv(X,1^,1^) {b|b}

3: b->Subdiv(X,1^,1^) {c|c}

and so on. 3^n would create three subdivisions, and 4^n would create four. In this way, the structure resembled an n-ary tree.

Using these grammars, we produced Table 7:

Type	Total Components	Speedup Factor
2^n	65536	8.08
3^n	59029	6.77
4^n	65536	4.90

Table 7: Speedup gained with grammars emulating n-ary trees.

From this grammar, we can see that 2^n provided the greatest speed improvement even though 2^n required more passes to generate the same number of components – that is, it attempted to do less in parallel. Thus, our initial intuition about the parallelization of the rule set proves false.

These results serve to confirm the results found when attempting to generate entire cities in parallel. We observe that it's not necessarily true that trying to force more parallelization onto the GPU yields better results.

However, we can not ignore that allowing the GPU to perform the generation in parallel can give significant speed increases. Notice that in 2^n the speed of generation increases nearly by an order of magnitude over the CPU version. Also notice that as the number of components generated increases for a single building, the speedup factor typically increases as well. Thus, we observe that both the structure and the number of components generated do have some impact on the speed of generation, though that impact is hard to predict or determine ahead of time.

3.3.8 Limitations

Our method of random building generation on the GPU has been fairly robust in practice, but it does suffer from some drawbacks which we have yet to overcome. The biggest disadvantage is that we have no method of implementing the occlusion/intersection tests found in the original description of split grammars [82], which means that we can not guarantee the absence of certain visual artifacts such as windows intersecting in unnatural fashions. Implementing this would severely reduce the independent nature of our building generation and would thus make GPU generation considerably more difficult.

Our method also has cases where it does not necessarily offer a speed benefit. In the case where grammar rules do not parallelize well, such as when rules do not generate a number of components that can be acted upon independently, our method does not perform much better than the CPU version. As an example of this, consider the case where a series of rules each generate a single component that the next rule processes. In this case, handling of the grammar takes place in a mostly "linear" fashion, and there are no parallelization gains. However, in practice we have found that this case appears very infrequently.

We are also constrained by the previously mentioned implementation details, although we have shown a number of work-arounds to mitigate this difficulty.

3.4 Summary of Results

We have shown new methods for improving the performance of split grammars, including a preprocessing stage and GPU acceleration. With our GPGPU scheme, we have

found that a speedup between two and five times that of the CPU-driven approach is common, with certain artificial grammar rule sets performing nearly ten times faster. We have demonstrated the speed impact of generating multiple buildings in parallel, concluding that attempt this type of parallelization only benefits the system when a large number of buildings are to be generated. We have also explored the implications that rule set structure has on the speed difference between the GPU and CPU generation approaches.

Figures 21 and 22 show buildings generated entirely on the GPU without texturing or injecting 3D models.

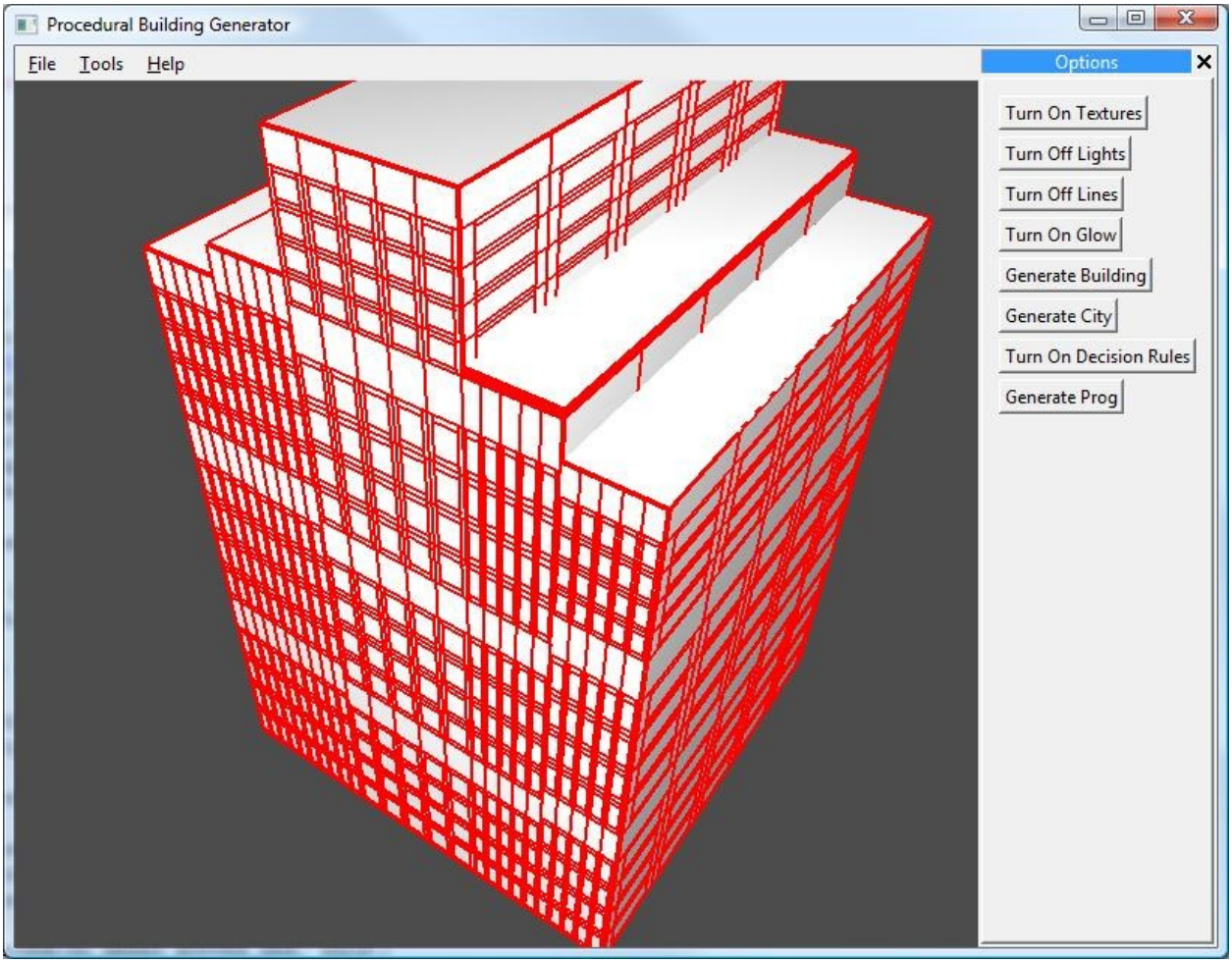

Figure 21: Generation of a building entirely on the GPU.

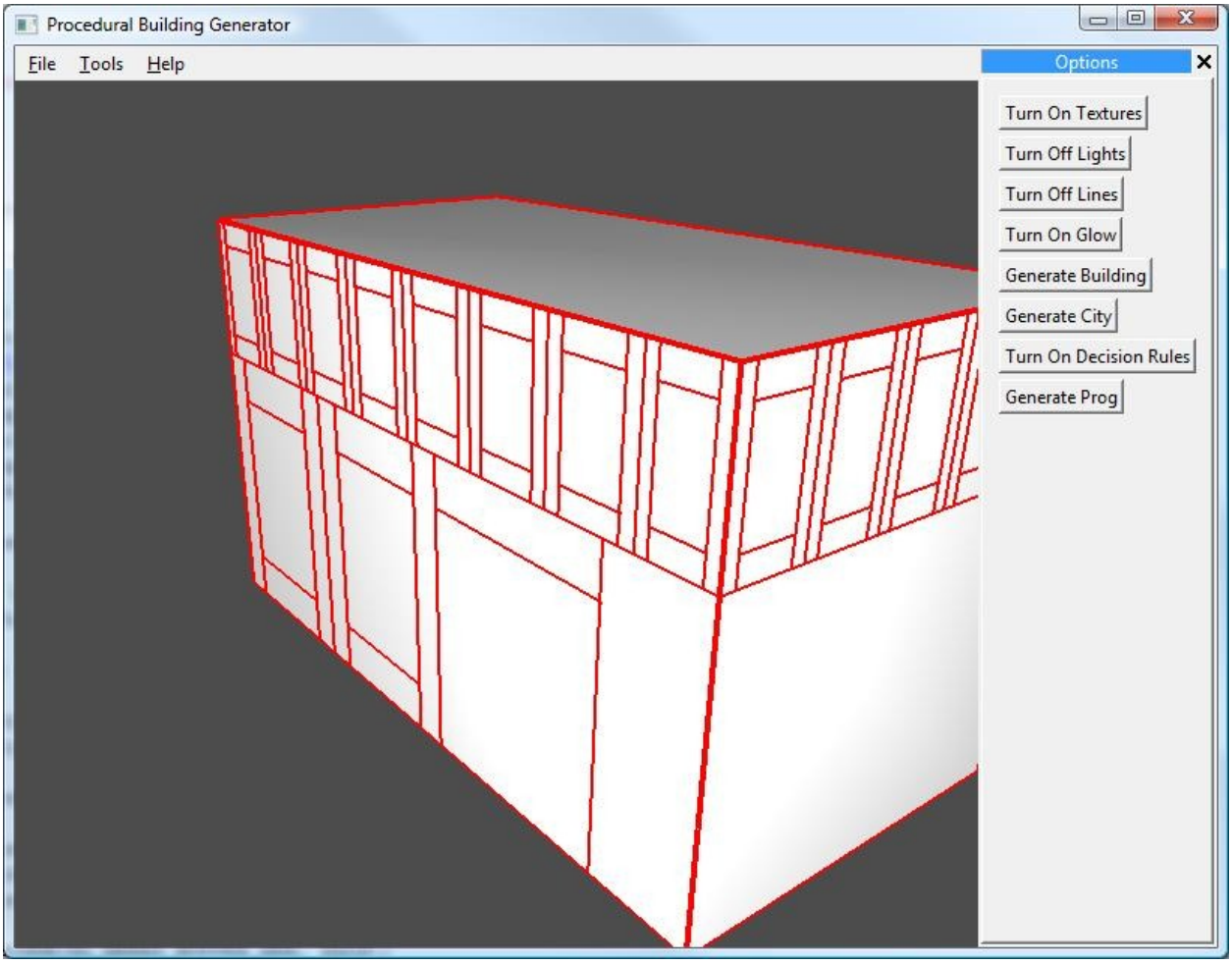

Figure 22: Generation of a building entirely on the GPU (2).

Chapter 4

4 Using Rule Learning to Refine Shape Grammars

4.1 Overview

So far, we have shown how to procedurally generate buildings and how to accelerate that building generation such that they can be generated much faster. However, we have not talked much about the intricacies required in authoring the grammar for a realistic building.

In essence, writing the grammar for a complex building requires more than knowledge of split grammars. It also requires some knowledge of shape and architecture. The problem is compounded when stochastics are introduced to provide large variations in a building via a single set of grammar rules, since it can be difficult to determine which set of random numbers eventually produced the final building.

We have observed that writing a convincing grammar, complete with stochastic choices, can be difficult for an average user. To help with this problem, we propose a method for refining the stochastic choices of grammars such that they can more closely approximate a user's desires. That is, we can present the user with a grammar, and the user is given the ability to gradually refine that grammar by indicating user preferences to the system.

Our process works by observing that, during the execution of grammar rules, whenever a random number is generated the system is making a "choice." Such choices include how high a component may be, whether to branch and generate new entire sections of a building, or how

large splits in a component may be. Initially, the outcome of these choices may not be what the specific user wants – that is, buildings may be generated that are too tall or that have too many floors. Thus, by changing the parameters of a choice – rather, by changing the distribution used when generating random numbers – we can refine these choices such that they more closely approximate user preferences.

It would be extremely cumbersome if the user were forced to look at a building and indicate everything that user liked or disliked. Further, it would not even be feasible; relations between different components would be hard for a person to indicate systematically. The system we propose uses a different approach. Instead of requiring the user to indicate preferences component-by-component, we allow the user to indicate preferences on a building level. A user indicates whether he *likes* or *dislikes* a building, and we use data mining techniques to attempt to isolate the choices which lead to the user's preference.

Our approach allows a user to quickly iterate through a group of buildings and indicate with a single button whether that building is liked or disliked. After a reasonably large sample size has been obtained (typically upward of 30 buildings), we use rule induction to produce a series of rules on the choice points such that, if those rules are followed, only buildings which the user preferred will be produced upon future generations. The user can then go back and modify the grammar to conform to these rules, or the rule set can be interpreted during building generation (with some efficiency penalty).

It should be noted that our approach works at a parametric level as opposed to a structural level. The *structure* of the grammar does not change. Thus, with our approach, it would be impossible to adapt a grammar to create entirely new styles of buildings that the grammar was not possible of creating prior. The *parameters* of building generation are what are modified; we can tweak and refine a current grammar, but we can not create new grammars. Creating new

grammars based on examples can be a subject of further research and is touched upon in [49].

4.2 Data Representation

When a user indicates that he likes or dislikes a building, we must output this information in a way that can be interpreted usefully by a rule induction system. We must take two things into consideration: the isolation of choice points, and how to best represent an entire building based on those choice points.

We identify a choice point as whenever a random number is generated. In our implementation of split grammars, this happens in two places: whenever a stochastic probability is given to an action list, and whenever the *rand* function is explicitly used.

Thus, consider the following example grammar:

```
1: lot        ->Size(20, 30 * rand(0.5,1), 20) {new_lot}
2: new_lot    ->Subdiv(Y, 1^, 3^) {bottom | top} : 0.5
              ->Replace(terminal) : 0.5
```

This grammar has exactly two choice points: the *rand(0.5,1)* statement "decides" how tall the initial component will be. Then, when new_lot is processed, the grammar rules decide whether that component will be subdivided into a bottom and top or simply replaced by a terminal component. The random numbers chosen at these points help to determine the appearance of the final building.

Then the problem becomes representing a building as the combination of all the choices which lead to the final building. Based on the previous example, a naïve implementation might represent a building as two numbers: the random number generated at the first decision point and

at the second. However, the representation can not be so straightforward in the presence of random numbers. For instance, consider the next example:

```
1: lot        ->Size(20,30 * rand(0.5,1),20) {new_lot}
2: new_lot    ->Subdiv(Y, 1^, 3^) {bottom | top}
3: top        ->Repeat(Y, 3) {floors}
4: floors     ->Size(1^,1^,2^) {extended_floor} : 0.5
              ->Replace(terminal) : 0.5
```

This example can not be neatly represented by our previous approach, because we have no way of being certain the total number of decisions that are going to be made. The initial height determination will contribute to how many floors are generated, meaning that the total number of decisions can vary between buildings. Thus, when trying to produce theories based on the building examples, it becomes unclear which number represents which decision point.

We opt for an approach which does not attempt to succinctly represent a single building as one example for rule induction to train upon. Instead, we observe that each terminal component has a distinct "path," or sequence of decisions which lead to it being generated (this observation requires one restriction which we will talk about later). If we have a list of all the decision points, we can represent any terminal component by the values chosen at the decision points which lead to the component being created, and whether the user liked or disliked the building. Note that not every decision point is necessarily invoked by the creation of a component, and in this instance the terminal component's "decision" at that point is specified as nonexistent or unknown (indicated by a ? in our datasets).

Instead of using a building as our instances to train rule induction with, we use terminal components. This is convenient, since we know that each terminal has a fixed number of decision points. This also eliminates the worry that buildings will result in a varied number of

terminal components, since this will simply result in more training instances being generated as opposed to an inconsistency in how the buildings are represented.

As a simple example, consider the following grammar:

1: lot ->Size(20,30 * rand(0.5, 1), 20) {new_lot}

When executed, this grammar results in the creation of one terminal component, a *new_lot*. This terminal component is what will be used for rule induction. Assuming that the value 0.7 was chosen in the *rand* statement and the user liked the building (denoted by a +), this component would be encoded as follows:

0.7, +

Consider the more involved example:

1: lot ->Size(20, 30*rand(0.5,1), 20) {new_lot}

2: new_lot: ->Subdiv(Y,1^,3^) {bottom | top}

3: bottom ->Size(1^,1^,2^) {new_bot} : 0.5

->Replace(terminal) : 0.5

4: new_bot ->Size(1^,Scope.sy * rand(0.5,1),1^) {final_bot}

This example has a greater number of variations which are possible. The *bottom* component may simply be replaced by a terminal component and processing on it stop. Alternatively, it may be extended in the Z direction, replaced by *new_bot*, and then resized in the Y direction.

Regardless, we isolate three decisions points: the initial *rand(0.5,1)*, the choice between how *bottom* should be transformed, and the *rand(0.5,1)* which determines how tall *final_bot* will be. If *bottom* is replaced by *terminal*, this final decision point will never be invoked.

Thus, in the case that generation stops with a *terminal* and *top*, we can encode the building using the following two instances:

0.7, 0.8, ?, + //terminal

0.7, ?, ?, + //top

Notice how *top* has unknown values for decision points 2 and 3, since it does not rely on the random numbers generated at those points. Terminal, however, relies on decision point 2, but still does not involve decision point 3.

If another building is generated, we may end up with the following:

0.7, ?, ?, + //top

0.7, 0.4, 0.6, +//final_bot

Here, *top* is similar to the previous example, but *final_bot* relies on all three decision points and thus has no unknowns.

In this way, we can represent any building that generates an arbitrary number of terminal components, since new components are simply new instances in the training set, and each decision point has *something* (even if it's unknown) for each terminal.

It must be noted that we can only represent buildings this way in the absence of any recursion/self-referencing components. Consider the following:

1:lot ->Size(20,Scope.sy + rand(2,5),20) {new_lot}

2:new_lot ->Replace(lot) : 0.5

->Replace(terminal) : 0.5

Notice that *new_lot* can be replaced by another *lot*, which will then extend the component further and by replaced by *new_lot* again. This process can repeat multiple times (until *new_lot* is replaced by *terminal)*. Thus, for a *terminal*, it is no longer clear how many decisions were involved – the *rand(2,5)* could have executed one time or a hundred times, resulting in different buildings. This has the same problem as representing buildings as all the decisions involved in generating the building – we can not predict exactly how many decisions were made. Care must be taken to avoid constructing rule sets like this.

4.3 Rule Generation

4.3.1 Generating JRip Rules

After the user has cycled through a series of buildings and indicated like/dislike, we are left with a data file containing a large number of terminal components and the decisions leading to each component.

We can then transform this data using a simple preprocessor into a form that the Weka's JRIP (a Java implementation of RIPPER; pseudo-code for RIPPER can be found in Figure 23 [46]) can understand and run the data through JRIP to generate a set of rules. These rules will be a conjunction of conditions which must evaluate to true to produce a given result (+ or -). For example:

(att0 > 0.5) && (att0 < 0.7) => +

=> -

The rule above indicates that if decision point 0 (the first decision point) is between 0.5 and 0.7, the user liked the building. Otherwise, the user disliked the building.

```
procedure BuildRuleSet(P,N)
P = positive examples
N = negative examples
RuleSet = {}
DL = DescriptionLength(RuleSet, P, N)
while P ≠ {}
      // Grow and prune a new rule
      split (P, N) into (GrowPos, GrowNeg) and (PrunePos, PruneNeg)
      Rule := GrowRule(GrowPos, GrowNeg)
      Rule := PruneRule(Rule, PrunePos, PruneNeg)
      add Rule to RuleSet
      if DescriptionLength(RuleSet, P, N) > DL + 64 then
            // Prune the whole rule set and exit
            for each rule R in RuleSet (considered in reverse order)
                  if DescriptionLength(RuleSet − {R}, P, N) < DL then
                        delete R from RuleSet
                        DL := DescriptionLength(RuleSet, P, N)
                        end if
                  end for
            return (RuleSet)
            end if
      DL := DescriptionLength(RuleSet, P, N)
      delete from P and N all examples covered by Rule
      end while
end BuildRuleSet

procedure OptimizeRuleSet(RuleSet, P, N)
for each rule R in RuleSet
      delete R from RuleSet
      UPos := examples in P not covered by RuleSet
      UNeg := examples in N not covered by RuleSet
      split (UPos, UNeg) into (GrowPos, GrowNeg) and (PrunePos, PruneNeg)
      RepRule := GrowRule(GrowPos, GrowNeg)
      RepRule := PruneRule(RepRule, PrunePos, PruneNeg)
      RevRule := GrowRule(GrowPos, GrowNeg, R)
      RevRule := PruneRule(RevRule, PrunePos, PruneNeg)
      choose better of RepRule and RevRule and add to RuleSet
      end for
end OptimizeRuleSet

procedure Ripper(P, N, k)
RuleSet := BuildRuleSet(P, N)
repeat k times RuleSet := OptimizeRuleSet(RuleSet, P, N)
return (RuleSet)
end Ripper
```

Figure 23: RIPPER pseudo-code. [46]

4.3.2 Experiments

It is difficult to make sweeping generalizations regarding rule induction given the wide

possibilities of grammar rule sets and the variable nature of what is “liked” and “disliked.” Further, what a user likes and dislikes may not map entirely to concrete decision criteria – the user could only have a vague idea of what is liked or disliked, or the qualities which contribute to a preference may not be the result of a single decision point but multiple decision points collaborating. Still, in this section and the next we will show that for a wide variety of preferences of varying complexity, our system performs well independent of the rule set size or number of decision points, and this performance should hold as complexity increases. To this end, we have developed a number of tests to verify our system.

To test rule generation, we went through individual data sets and indicated whether we liked or disliked a building based upon some decision criteria we established a priori. We cycled through fifty building examples from each rule set indicating like/dislike. We removed identical instances from the data set, to both lower the amount of 'noise' generated by buildings which created more components and also to reduce the amount of time needed to learn a model. We then validated that the generated rules matched what we expected.

The first test was whether JRIP would deduce a “boolean” decision. That is, a building was either too tall or too short – we did not care about variations in tallness or shortness. Table 8 indicates the grammars we used (from the set in Table 1) and the criteria we used to make our decision.

Grammar	Decision
1	Tall buildings were accepted and short buildings were rejected, where “tall” and “short” were based on our subjective judgment as decisions were being made.
1	Buildings which had windows lining the top were accepted while those without windows on the top were rejected.
2	Buildings with three main sections were accepted; all others were rejected.
7	Buildings with a tip at the top of the main section were accepted. If a tip was not present, buildings were rejected.
9	Only buildings with columns supporting the building were accepted.
11	Only buildings with an annex connecting two major sections were accepted.

Table 8: Simple decision criteria used when determining whether to like or dislike a building generated from the given grammar.

The second test involved determining ranges of values. That is, a building could be too tall *and* too short – only those buildings in between would be accepted. To find a building quality which we could reliably verify, we relied on building height in all cases to validate ranges. Table 9 indicates the grammars used and the decision criteria.

Grammar	Decision
1	Buildings which were too tall or too short were rejected; only buildings which fell within a desired height were accepted.
2	Same as Grammar 1.
4	Same as Grammar 1.

Table 9: The "continuous" decision criteria used when deciding whether to like or dislike a building generated from the given grammar.

In the third test, we tested for two conditions as opposed to one. By this we mean that might test against whether a certain building component existed *and* whether that component was tall enough or not. Table 10 indicates the tests that we performed.

Grammar	Decision
1	Buildings which were tall and which had *no* windows along the top section were accepted, whereas other buildings were rejected.
2	Buildings which had a middle section which we considered tall were the only buildings accepted.
10	Buildings which had a left major component that was above a given height were the only ones accepted.
11	Buildings which had an annex and which had a left major section which we considered tall were the only ones accepted.

Table 10: Description of the two-condition decision criteria.

After these tests, we started to perform more "arbitrary" tests to evaluate the generated rules. That is, we did not confine ourselves to individual conditions but instead allowed ourselves to determine whether we liked or disliked a building based on more free-form decision processes. These are harder to verify but are still necessary to test the flexibility of our system. Table 11 indicates the performed tests.

Grammar	Decision
1	Only buildings completely lacking windows on both the top and bottom sections are accepted.
1	If a building is short, it should not have any windows. If it is tall, it should have windows.
2	The generated building can only have a middle or right main section (not one on the left), and that section must be tall.
2	Building must either have three sections or must be tall.
7	The building must have both two towers on each side and a tip at the top.
9	If the building is short, it should have columns supporting its right side. If the building is tall, it should be solid where the columns would be.
10	If the building has two extruding sections, one must be short. If the building only has one extruding section, it must be tall.
11	"Triples" and buildings with annexes are never accepted. Of the other types, only tall buildings are accepted.
11	If the building is a single tower, it must be tall. Otherwise, it must be short.

Table 11: More general, less structured decision criteria over a large set of grammars.

The reader will likely notice that certain rule sets from our test set are rarely seen in the above tables. This is either because the grammars had too few decision points to make learning a worthwhile process or because the buildings generated had too many components, such that the memory requirements needed to learn a theory would be too large.

4.3.3 JRIP Performance

4.3.3.1 Rule Set Size and Performance

Table 12 shows the set of rules generated from the decision criteria in Table 8. By referencing back to the actual grammars, one can readily see that the rules generated for grammars 1, 2, and 11 clearly reflect our decision criteria. Though the rules generated in the first test seem somewhat convoluted, they do in fact do a good job of representing our decision criteria. Table 13 illustrates this – Table 13 maps the rule sets to their accuracy, precision for the + class, true positive rate (TP) for the + class, and false positive rate (FP) for the + class when performing a 10-fold cross-validation check against the data.

G	50 Buildings
1	(att0 <= 1.268) => class=- (102.0/0.0) (att0 <= 1.59613) and (att0 >= 1.56504) and (att1 <= 0.377453) => class=- (12.0/0.0) => class=+ (158.0/0.0)
1	(att1 >= 0.800775) and (att0 >= 1.01259) => class=+ (24.0/0.0) => class=- (218.0/0.0)
2	(att1 >= 30.1474) => class=+ (42.0/2.0) => class=- (58.0/0.0)
7	(att2 >= 0.58858) => class=- (24.0/0.0) (att0 >= 0.84279) and (att0 <= 0.934785) and (att1 >= 0.157231) => class=- (7.0/0.0) (att1 <= 0.547838) and (att1 >= 0.265755) and (att0 >= 0.968056) and (att0 <= 1.21582) => class=- (9.0/0.0) (att0 <= 1.13569) and (att0 >= 1.11255) => class=- (3.0/0.0) => class=+ (72.0/5.0)
9	(att1 >= 0.505356) => class=- (28.0/0.0) (att0 >= 0.861504) and (att0 <= 0.865477) => class=- (5.0/0.0) (att0 >= 0.888595) and (att0 <= 0.899637) => class=- (5.0/0.0) (att0 <= 0.778207) and (att0 >= 0.764794) => class=- (3.0/0.0) (att0 >= 0.880786) and (att0 <= 0.885061) => class=- (4.0/0.0) (att0 >= 0.93657) and (att0 <= 0.976315) => class=- (5.0/0.0) => class=+ (72.0/9.0)
11	(att1 >= 0.718162) => class=+ (60.0/0.0) => class=- (69.0/0.0)

Table 12: Rules generated after making decisions based on the criteria in Table 8.

Grammar	Accuracy (%)	Precision	TP	FP
1	100	1	1	0
1	99.17	0.92	1	0.009
2	98	0.95	1	0.033
7	82.61	0.78	0.985	0.396
9	76.23	0.70	0.952	0.441
11	100	1	1	0

Table 13: Accuracy, Precision (+), TP (+), and FP (+) after running 10-fold cross validation on the rules generated in Table 12.

We can see from the above tables that the simpler rules – the ones which understandably represent our decision criteria more closely – perform much better than the more complicated rules which do not approximate our decision criteria. For rule sets 7 and 9, the rules – though they seem somewhat convoluted – still do a reasonable job of prediction. Thus, we argue that generating buildings based on those rules will yield a building which more closely approximates the user's preferences than generating without those rules.

What is not immediately clear is the inclusion of att1 in the decision process, which for grammar 1 determines if the top will have windows or not. Our explanation is that within the 50 buildings sampled, the taller ones happened to have windows more often (simply by coincidence). A larger sample size would likely eliminate this anomaly, but this would take more time, and the results are good given the sample size.

We now move on to analyzing the more continuous decision criteria found in Table 9. Again, Table 14 shows the generated rules, and Table 15 shows the accuracy, PD, and PF of these rules.

G	50 Buildings
1	(att0 <= 1.51992) and (att0 >= 1.06189) and (att0 <= 1.37475) => class=+ (64.0/0.0) (att1 >= 0.568011) and (att0 <= 1.51992) and (att0 >= 1.40804) => class=+ (32.0/6.0) => class=- (168.0/2.0)
2	(att0 <= 0.847404) and (att4 <= 0.754112) => class=+ (36.0/0.0) (att0 <= 0.704157) => class=+ (2.0/0.0) => class=- (60.0/2.0)
4	(att1 <= 0.759941) and (att1 >= 0.604343) => class=+ (1544.0/0.0) (att1 <= 0.562059) and (att1 >= 0.560656) => class=+ (212.0/0.0) (att2 <= 0.674123) and (att1 <= 0.533418) => class=+ (86.0/0.0) => class=- (2911.0/87.0)

Table 14: Rules generated after making decisions based on the criteria in Table 9.

Grammar	Accuracy (%)	Precision	TP	FP
1	94.70	0.898	0.957	0.058
2	94.90	0.973	0.9	0.017
4	98.02	0.992	0.959	0.005

Table 15: Accuracy, Precision (+), TP (+), and FP (+) after running 10-fold cross validation on the rules generated in Table 14.

Again, a quick cross-reference with the rule sets shows that these rules are indicative of the decisions we were making. For instance, in grammar 1, the first decision point (att0) is a multiplier on the height. We can see from the rules that a building would only be accepted if its height multiplier were within some range.

Tables 16 and 17 show the equivalent data for the decision criteria from Table 10. Here the results become somewhat more mixed but are still promising. Grammar 1, for instance, has a very succinct rule which maps to the decision criteria perfectly. Grammars 2 and 10, which have significantly more complex rules, still perform well under the cross-validation.

G	50 Buildings
1	(att1 >= 0.817835) and (att0 >= 1.38643) => class=+ (8.0/0.0) => class=- (264.0/0.0)
2	(att1 <= 58.6569) and (att5 >= 0.561788) => class=+ (17.0/1.0) (att1 <= 58.6569) and (att1 <= 46.6805) => class=+ (15.0/3.0) (att3 >= 39.3204) and (att4 <= 0.86166) => class=+ (8.0/0.0) (att1 <= 58.6569) and (att1 >= 51.478) and (att4 >= 0.095645) and (att0 >= 0.770553) => class=+ (7.0/0.0) (att3 >= 54.8198) => class=+ (6.0/0.0) => class=- (49.0/1.0)
10	(att0 >= 0.393851) and (att0 <= 0.412064) => class=- (20.0/6.0) (att0 <= 0.326679) and (att0 >= 0.323597) => class=- (7.0/0.0) => class=+ (111.0/17.0)
11	(att10 <= 0.367272) and (att10 <= 0.2756) => class=+ (21.0/0.0) (att10 <= 0.367272) and (att0 >= 0.770388) and (att10 >= 0.308615) => class=+ (15.0/0.0) (att0 >= 0.960558) and (att0 <= 0.960558) => class=+ (3.0/0.0) => class=- (85.0/0.0)

Table 16: Rules generated after making decisions based on the criteria in Table 10.

Grammar	Accuracy (%)	Precision	TP	FP
1	100	1	1	0
2	77.45	0.737	0.84	0.288
10	74.64	0.773	0.92	0.711
11	94.35	0.848	1	0.082

Table 17: Accuracy, Precision (+), TP (+), and FP (+) after running 10-fold cross validation on the rules generated in Table 16.

Tables 18 and 19 move into our more free-form decision criteria. As such, they are most representative of how a user might behave. They are also the hardest to interpret by simple analysis. Thus, we are forced to rely on Table 19 to make our judgments.

G	50 Buildings
1	(att1 >= 0.866878) and (att0 <= 1.55002) => class=+ (6.0/0.0) => class=- (262.0/0.0)
1	(att0 <= 1.48209) and (att1 <= 0.688314) => class=- (114.0/0.0) (att1 >= 0.907804) and (att0 >= 1.48209) and (att1 <= 0.9476) => class=- (4.0/0.0) (att0 >= 1.96389) and (att0 <= 1.96389) => class=- (2.0/0.0) => class=+ (124.0/0.0)
2	(att1 >= 37.5368) and (att5 >= 0.579946) => class=+ (13.0/1.0) (att0 <= 0.914258) and (att0 >= 0.854784) => class=+ (14.0/2.0) (att3 <= 45.2879) and (att3 >= 41.3456) => class=+ (4.0/0.0) => class=- (69.0/10.0)
2	(att4 >= 0.535539) and (att0 <= 0.757277) => class=- (14.0/4.0) (att4 >= 0.565355) and (att4 >= 0.938047) and (att0 <= 0.971068) => class=- (4.0/0.0) (att4 >= 0.535539) and (att2 >= 36.0729) => class=- (10.0/2.0) (att4 >= 0.565355) and (att4 <= 0.584246) => class=- (4.0/0.0) (att1 <= 32.4693) => class=- (2.0/0.0) (att3 <= 31.8769) => class=- (4.0/0.0) => class=+ (62.0/0.0)
7	(att2 >= 0.516923) => class=- (18.0/0.0) (att1 >= 0.742271) => class=- (9.0/0.0) (att0 <= 0.974905) and (att0 >= 0.887744) and (att1 >= 0.146367) => class=- (5.0/0.0) => class=+ (81.0/9.0)
9	(att0 <= 0.848485) => class=+ (63.0/24.0) (att1 >= 0.497391) => class=+ (5.0/0.0) => class=- (59.0/5.0)
10	(att1 >= 0.526182) => class=- (13.0/4.0) (att3 <= 0.549162) => class=- (10.0/3.0) => class=+ (99.0/36.0)
11	(att1 >= 0.335521) and (att1 <= 0.636036) and (att0 >= 0.818601) => class=+ (19.0/0.0) (att8 >= 1.26352) => class=+ (2.0/0.0) => class=- (100.0/0.0)
11	(att0 <= 0.820423) and (att1 <= 0.775414) => class=+ (32.0/3.0) (att7 >= 0.500836) => class=+ (7.0/0.0) (att10 >= 0.358342) and (att0 <= 0.960833) => class=+ (6.0/0.0) (att0 >= 0.983547) and (att0 <= 0.983547) => class=+ (2.0/0.0) => class=- (68.0/0.0)

Table 18: Rules generated after making decisions based on the criteria in Table 11.

Grammar	Accuracy (%)	Precision	TP	FP
1	100	1	1	0
1	99.77	0.976	1	0.025
2	71	0.655	0.5	0.161
2	80	0.824	0.897	0.406
7	90.27	0.877	0.986	0.244
9	77.95	0.733	0.673	0.154
10	54.10	0.574	0.771	0.769
11	95.04	0.826	0.905	0.04
11	92.17	0.927	0.864	0.042

Table 19: Accuracy, Precision (+), TP (+), and FP (+) after running 10-fold cross validation on the rules generated in Table 16.

From all these tables, we show that the rule generation typically establishes rules with a high degree of accuracy – almost always greater than 75% and often greater than 90%. When accuracy starts to drop, we observe a trend in the TP and FP values: the TP values stay relatively high even while the FP values increase, suggesting that the rules are often indicating that a building component is accepted (liked) even though it should not be while few buildings are liked when they shouldn't be. This hints that the rules are too lenient, which we contend is better than being too restrictive and confining the variation in the building generation (although obviously domain needs dictate whether this is really better or not).

We have found that for our test set, 50 buildings is usually a good sample size. There is a risk that using too small a sample size yields misleading results – using 10 buildings often leads to 100% accuracy while providing rules that are in no way representative of the decision criteria. Using too large a sample size (100+) quickly becomes a burden on the user and would severely limit the usefulness of the system.

We would also like to note that, on average, the rule sizes are very small (usually about

3-4 rules) and the time to obtain these rules was almost instantaneous in all instances. The small rule size will become important in the next section, and the time can be a significant user consideration.

4.3.3.2 Limitations

We have noticed one critical limitation of the system: *relative* values of decision points can not be represented. For example, our system has no clean way of inducting that if the main section is tall, a subsection should be twice as tall, regardless of absolute values of height. This kind of relationship might be representable via a large set of decision rules, but that would be impractical. Resolving this limitation is an area for future work.

4.4 Adhering to Rules

4.4.1 User-Driven Approach

Often, the simplest way to adhere to a set of given rules is to modify the grammar itself. For example, if JRIP outputs something akin to:

(att0 > 1.5) => +

then a user could go back to the grammar and change the first decision point so that its minimum value is 1.5. The user could do this with all the rules, providing there are no conflicts.

This approach is the most efficient, requiring no extra processing to be done during building generation. However, it can often be difficult to change the rules such that all the rules are followed while still providing the full range of expressibility. That is, consider the following

rules:

(att1 > 1.5) and (att0 > 1.5)=>+

(att1 < 1.2) and (att0 < 1.2)=>+

=>-

Here, we have an instance where, if we followed the first rule and changed the minimum for att1 to 1.5 and the minimum for att0 to 1.5, we would lose the series of buildings allowed by the second rule, thus greatly limiting our grammar. This problem can be worked around via use of extra rules and predicates, but that solution is not always straight-forward, especially as the number of rules increases.

4.4.2 Rejection Sampling

There is a computational way of ensuring that building generation adheres to the generated rules, but it comes at a performance cost. The problem can be divided into two distinct sub-problems:

1. If the default class for the rules output by JRIP is '-', the problem is trivial. Every time a decision is made, ensure that there is still one rule which results in '+' that can be satisfied. If there are no such rules, re-evaluate the decision (recompute the random number), otherwise continue generating components.
2. If the default class is '+', resort to rejection sampling.

The second sub-problem is clearly the harder of the two. The problem lies in that we can not simply evaluate decisions as we go and ensure that all the rules will evaluate to false (meaning that all the '-' class rules will not be true and the building will be valid), because there is a possibility we will become “trapped” in mutually conflicting sections of rules, and we can

not be sure exactly which decision caused us to no longer be able to generate a valid building.

The solution to this is rejection sampling [66]. We generate an entire range of random numbers for each decision point a priori and ensure that they lead to a sequence of decisions which satisfies all the rules (leads to the '+' class). If they do not, we generate an entirely new set of random numbers, continuing this process until all rules are satisfied.

This solution obviously comes at a cost in terms of efficiency. We have found that, depending on the size of the generated rules and the number of decision points, building generation becomes prohibitively slow. This is due to the inherent "trial-and-error" of our non-adaptive rejection sampling scheme coupled with the fact that a valid set of decision values may be difficult to find. Improving this is an area for future research. One solution may be to modify JRIP such that its default class is always '-', while another may be to avoid JRIP altogether in favor of some other machine learning scheme.

4.5 Summary of Results

In this chapter, we have shown the effectiveness of rule induction to discover relevant parameters for refining building generation. Given a large breadth of decision criteria over various grammar rule sets, we have shown that our system typically performs well, frequently offering near-perfect accuracy. In cases where the generated rules were simple, we were able to verify our results via visual inspection, and in more complicated cases we employed metrics (accuracy, TP, FP, precision) to validate the system. We have also provided ways for verifying that building generation adheres to the rules produced by JRIP, though more robust methods are an area for future research.

Chapter 5

5 Conclusion

Recall the two primary goals of this thesis:

- Implement a method to increase the speed of building generation
- Implement a method for refining building generation based on user preference

Each of these goals was accomplished to varying degrees of effectiveness. The following subsections will summarize the accomplishments toward each goal.

Implement a method to increase the speed of building generation

This thesis provided two novel methods to improve the speed of building generation: preprocessing a split grammar rule set via Lex and Yacc, and using GPGPU techniques to parallelize building generation. Although we could not provide concrete evidence that the preprocessing benefited over the implementations of other authors, we fully believe that it could provide a significant speed increase. We showed that GPGPU parallelization provided even further speed benefits, typically doubling or tripling the speed of building generation. This progress was implemented in our custom tool, which is capable of handling the majority of split grammar operations and which could be further extended.

Implement a method for refining building generation based on user

preference

This thesis provided a rule induction scheme relying on RIPPER that can effectively isolate emergent properties of a building and refine the rule set generating that building such that future buildings will conform to user preferences. This thesis provided various tests against both useful and artificial grammars to validate the effectiveness of the rule induction. We utilized the Weka to both discover and analyze the rules with the Weka's implementation of RIPPER (JRIP). We also provided schemes for ensuring buildings adhere to the generated rules, although these schemes could potentially be expanded upon.

5.1 Future Work

Further work should be done to increase the ability of our GPGPU scheme to handle the more complicated aspects of split grammars. In addition, better methods for adhering to generated rules need to be researched.

5.1.1 Parallelizing the Complex Split Grammar Operations

As we stated previously, we do not currently have a scheme for parallelizing occlusion tests and snap lines, two important aspects of split grammars. Parallelizing snap lines would likely not be difficult. However, parallelizing occlusion tests would. This is because occlusion tests do not map naturally to GPU parallelization – performing a test for a component requires knowledge of other components, which is prohibited by the nature of the parallelization. This is

not to say that parallelization would be impossible, just that it would be significantly more complex and may not yield speed gains over an entirely CPU approach. This needs to be explored further.

Furthermore, there are various qualities of split grammar rule sets – size and number of loops – which have limited the rule sets which can be parallelized with the GPU hanging. It is our hope that future nVidia drivers will alleviate some of this, but there will likely always be a need to work around some of the GPU limitations. One possibility to avoid the size problem is to break the geometry shader into smaller geometry shaders and run those shaders in sequence. The ability to do this and the implications on speed need to be explored.

5.1.2 Real-World Case Studies

Wonka and Müller cite various rule sets above 100 rules to generate complex environments such as Rome [48]. Unfortunately, these rules were not released, and we did not have time to research architecture and create such a complicated rule set on our own. We would be interested in knowing how our schemes hold up when presented with real-world rule sets, although this may require the implementation of work-arounds discussed in 5.1.1.

5.1.3 Improved Rule Adherence

We illustrated cases in which ensuring that building generation adheres to the rules generated by rule induction is relatively straightforward. However, we also demonstrated cases where adhering to these rules is not so easy (specifically, when the default class generated by JRIP is '+'). Currently, the best solution to the problem we know is a simple rejection sampling

scheme, and this is less than ideal. We would like to explore other methods for ensuring that building generation adheres to rules.

One possible method may be to favor treatment learning [45] over rule induction in the learning phase. Treatment learning is focused on providing a set of *treatments* which can be used to ensure that a theory approaches a positive class. This may prove more useful than rule induction, which does not have this focus.

5.1.4 Extensions to the Application

Currently, our application is fairly rudimentary, with a programmer-centric user interface and little integration with the Weka to automatically send user preferences and receive the results of rule induction. This could be expanded to be more user-friendly and to integrate fully with the Weka (or custom-written algorithms) such that the user never needs to leave our application to process data.

5.1.5 Extensions to Split Grammars

We hypothesize that it would be fairly easy to extend split grammars to represent building interiors. The operations are mostly already there; special considerations need to be taken into account for interior doors and ensuring entrance points into rooms, but we believe this could be achieved by minor alterations (or the addition of new operations) to the split grammar language.

We also believe that split grammars could be used to generate certain classes of textures. For example, bricks and tiling could be represented very naturally within the split grammar. Other textures, such as flagstone, may also be possible. This would require very little change to

the split grammar language, possibly just providing the ability to apply material properties to building components.

Bibliography

[1] Adams, D. 2002. Automatic Generation of Dungeons for Computer Grames. Undergraduate Project Disseration, University of Sheffield. Available from http://www.dcs.shef.ac.uk/intranet/teaching/projects/archive/ug2002/pdf/u9da.pdf.

[2] Aho, A. V., Sethi, R., and Ullman, J. D. 1986 Compilers: Principles, Techniques, and Tools. Addison-Wesley Longman Publishing Co., Inc.

[3] AMD's Close-to-the-Metal. Available from http://sourceforge.net/projects/amdctm/.

[4] Astle, D., 2005. More OpenGL Game Programming. Course Technology PTR.

[5] Astle, D. and Hawkins, K. 2001 OpenGL Game Programming. Premier Press, Incorporated.

[6] ARB Vertex Buffer Object Specification. Available from http://www.opengl.org/registry/specs/ARB/vertex_buffer_object.txt.

[7] Birch, P. J., Browne, S. P., Jennings, V. J., Day, A. M., and Arnold, D. B. 2001. Rapid procedural-modelling of architectural structures. In Proceedings of the 2001 Conference on Virtual Reality, Archeology, and Cultural Heritage (Glyfada, Greece, November 28 - 30, 2001). VAST '01. ACM, New York, NY, 187-196. DOI=http://doi.acm.org/10.1145/584993.585023.

[8] Bloomenthal, J. 1985. Modeling the mighty maple. SIGGRAPH Comput. Graph. 19, 3 (Jul. 1985), 305-311. DOI= http://doi.acm.org/10.1145/325165.325249.

[9] BrookGPU. Available from http://graphics.stanford.edu/projects/brookgpu/.

[10] Brunk, C. and Pazzani, M. Noise-tolerant relational concept learning algorithms. In Proceedings of the Eighth International Workshop on Machine Learning, Ithaca, New York, 1991. Morgan Kaufmann.

[11] Catlett, Jason. Megainduction: a test flight. In Proceedings of the Eighth International Workshop on Machine Learning, Ithaca, New York, 1991. Morgan Kaufmann.

[12] Clough, Craig. Gaming Industry Explodes to New Heights. KCCI.com. Available from http://www.kcci.com/money/14434694/detail.html.

[13] Cohen, W. W. Efficient pruning methods for separate-and-conquer rule learning systems. In Proceedings of the 13th International Joint Conference on Artificial Intelligence, Chambery, France,1993.

[14] Cohen, W.. 1995. Fast effective rule induction. In Machine Learning: Proceedings of the Twelfth International Conference, Lake Tahoe, California.

[15] CUDA. Available from http://www.nvidia.com/object/cuda_home.html.

[16] Cutler, B., Dorsey, J., McMillan, L., Müller, M., and Jagnow, R. 2002. A procedural approach to authoring solid models. ACM Trans. Graph. 21, 3 (Jul. 2002), 302-311. DOI=http://doi.acm.org/10.1145/566654.566581.

[17] Debevec, P. E., Taylor, C. J., and Malik, J. 1996. Modeling and rendering architecture from photographs: a hybrid geometry- and image-based approach. In Proceedings of the 23rd Annual Conference on Computer Graphics and interactive Techniques SIGGRAPH '96. ACM, New York, NY,11-20. DOI= http://doi.acm.org/10.1145/237170.237191.

[18] Duchaineau, M., Wolinsky, M., Sigeti, D. E., Miller, M. C., Aldrich, C., and Mineev-Weinstein, M. B. 1997. ROAMing terrain: real-time optimally adapting meshes. In Proceedings of the 8th Conference on Visualization '97 (Phoenix, Arizona, United States, October 18 - 24, 1997). R. Yagel and H. Hagen, Eds. IEEE Visualization. IEEE Computer Society Press, Los Alamitos, CA, 81-88.

[19] Ebert, D. S., Musgrave, F. K., Peachey, D., Perlin, K., and Worley, S. 2002 Texturing and Modeling: a Procedural Approach. 3rd. Morgan Kaufmann Publishers Inc.

[20] Engel, Wolfgang (Editor), 2005. Shader X3: Advanced Rendering with DirectX and OpenGL. Charles River Media, Inc., Hingham, MA.

[21] Fernando, R., 2004. GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics. Addison-Wesley Professional; Har/Cdr edition.

[22] Flemming U, 1987. More than the sum of parts: the grammar of Queen Anne houses. Environment and Planning B: Planning and Design 14(3) 323-350.

[23] Furnkranz, J. and Widmer, G. Incremental reduced error pruning. In Machine Learning: Proceedings of the Eleventh Annual Conference, New Brunswick, New Jersey, 1994. Morgan Kaufmann.

[24] Göddeke, D., GPGPU::Basic Math Tutorial. Available from http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/tutorial.html.

[25] Greuter S., Parker J., Stewart N., Leach G. 2003. Undiscovered Worlds – Towards a Framework for Real-Time Procedural World Generation. In Fifth International Digital Arts and Culture Conference, Melbourne, Australia.

[26] Greuter, S., Parker, J., Stewart, N., and Leach, G. 2003. Real-time procedural generation of `pseudo infinite' cities. In Proceedings of the 1st international Conference on Computer Graphics and interactive Techniques in Australasia and South East Asia (Melbourne, Australia, February 11 – 14, 2003). GRAPHITE '03. ACM, New York, NY, 87-ff. DOI= http://doi.acm.org/10.1145/604471.604490.

[27] Halo 3 to sell 10 million copies, generate $700 million revenues, cost $30 million to dev. MaxConsole. Available from http://www.maxconsole.net/?mode=news&newsid=21099.

[28] Hart, E., nVidia GeForce 8800 OpenGL Extensions. Available from http://www.slideshare.net/icastano/geforce-8800-opengl-extensions/.

[29] Integrating biomechanics into developmental plant models expressed using L-systems. In: H.-Ch. Spatz and T. Speck (Eds.): Plant biomechanics 2000. Proceedings of the 3rd Plant Biomechanics Conference, Freiburg-Badenweiler, August 27 to September 2, 2000. Georg Thieme Verlag, Stuttgart, 2000, pp. 615-624.

[30] Introversion Software. Procedural Content Generation. Available from http://www.gamecareerguide.com/features/336/procedural_content_.php.

[31] Kelly, G. and McCabe, H. 2006. A Survey of Procedural Techniques for City Generation. In ITB Journal, Issue 14. Available from http://www.gamesitb.com/SurveyProcedural.pdf.

[32] Kelly, G. and McCabe, H. 2006. Interactive generation of cities for real-time applications. In ACM SIGGRAPH 2006 Research Posters (Boston, Massachusetts, July 30 - August 03, 2006). SIGGRAPH '06. ACM, New York, NY, 44. DOI=http://doi.acm.org/10.1145/1179622.1179673.

[33] Kilgard, M. 2005. NVIDIA OpenGL 2.0 Support. Available from

http://download.nvidia.com/developer/Papers/2005/OpenGL_2.0/
NVIDIA_OpenGL_2.0_Support.pdf.

[34] Langley, P. and Simon, H. A. 1995. Applications of machine learning and rule induction. Commun. ACM 38, 11 (Nov. 1995), 54-64. DOI= http://doi.acm.org/10.1145/219717.219768.

[35] Laycock, R. G. and Day, A. M. 2003. Automatically generating large urban environments based on the footprint data of buildings. In Proceedings of the Eighth ACM Symposium on Solid Modeling and Applications (Seattle, Washington, USA, June 16 - 20, 2003). SM '03. ACM, New York, NY, 346-351. DOI= http://doi.acm.org/10.1145/781606.781663.

[36] Lefebvre, L., and Poulin, P. 2000. Analysis and synthesis of structural textures. In Graphics Interface, 77-86.

[37] Lefebvre, S. and Neyret, F. 2003. Pattern based procedural textures. In Proceedings of the 2003 Symposium on interactive 3D Graphics (Monterey, California, April 27 - 30, 2003). I3D '03. ACM, New York, NY, 203-212. DOI= http://doi.acm.org/10.1145/641480.641518.

[38] Legakis, J., Dorsey, J., and Gortler, S. 2001. Feature-based cellular texturing for architectural models. In Proceedings of the 28th Annual Conference on Computer Graphics and interactive Techniques SIGGRAPH '01. ACM, New York, NY, 309-316. DOI= http://doi.acm.org/10.1145/383259.383293.

[39] Lluch, J., Camahort, E., and Vivó, R. 2003. Procedural multiresolution for plant and tree rendering. In Proceedings of the 2nd international Conference on Computer Graphics, Virtual Reality, Visualisation and interaction in Africa (Cape Town, South Africa, February 03 - 05, 2003). AFRIGRAPH '03. ACM, New York, NY, 31-38. DOI=http://doi.acm.org/10.1145/602330.602336.

[40] LOGO Foundation. Available from http://el.media.mit.edu/Logo-foundation/.

[41] Losasso, F. and Hoppe, H. 2004. Geometry clipmaps: terrain rendering using nested regular grids. ACM Trans. Graph. 23, 3 (Aug. 2004), 769-776. DOI= http://doi.acm.org/10.1145/1015706.1015799.

[42] Martin, J. 2004. The Algorithmic Beauty of Buildings Methods for Procedural Building Generation. Computer Science Honors Thesis, Trinity University, Texas. Available from http://lib.trinity.edu/digitalcommons/cs_honors/4/doc.pdf.

[43] Martin, J. 2006. Procedural House Generation: A method for dynamically generating floor

plans. In Symposium on Interactive 3D Graphics and Games. Available from http://www.cs.unc.edu/~jmartin/i3d/poster.pdf.

[44] Matthews, J. L-Systems Explorer. Available from http://www.generation5.org/content/2002/lse.asp.

[45] Menzies, T. and Hu, Y. 2003. Data Mining for Very Busy People. Computer 36, 11 (Nov. 2003), 22-29. DOI= http://dx.doi.org/10.1109/MC.2003.1244531.

[46] Menzies, T. Why learn rules? Available from http://csee.wvu.edu/~timm/cs591o/old/Rules.html.

[47] Miyata, K. 1990. A method of generating stone wall patterns. SIGGRAPH Comput. Graph. 24, 4 (Sep. 1990), 387-394. DOI= http://doi.acm.org/10.1145/97880.97921.

[48] Müller, P., Wonka, P., Haegler, S., Ulmer, A., and Van Gool, L. 2006. Procedural modeling of buildings. ACM Trans. Graph. 25, 3 (Jul. 2006), 614-623. DOI=http://doi.acm.org/10.1145/1141911.1141931.

[49] Müller, P., Zeng, G., Wonka, P., and Van Gool, L. 2007. Image-based procedural modeling of facades. ACM Trans. Graph. 26, 3 (Jul. 2007), 85. DOI= http://doi.acm.org/10.1145/1276377.1276484.

[50] Musgrave, F. K., Kolb, C. E., and Mace, R. S. 1989. The synthesis and rendering of eroded fractal terrains. In Proceedings of the 16th Annual Conference on Computer Graphics and interactive Techniques SIGGRAPH '89. ACM, New York, NY, 41-50. DOI= http://doi.acm.org/10.1145/74333.74337.

[51] Noel, J. 2003. Dynamic Building Plan Generation. Undergraduate Project Disseration, University of Sheffield, 2003. Available from http://www.dcs.shef.ac.uk/intranet/teaching/projects/archive/ug2003/pdf/u0jn.pdf.

[52] Nvidia OpenGL Extensions Specification for the GeForce 8 Series Architecture. Available from http://developer.download.nvidia.com/opengl/specs/g80specs.pdf.

[53] Olsen J., 2004. Realtime Procedural Terrain Generation. Department of Mathematics And Computer Science (IMADA) University of Southern Denmark.

[54] O'Neill, S. 2001. A Real-Time Procedural Universe, Part One: Generating Planetary Bodies. Available from http://www.gamasutra.com/features/20010302/oneil_01.htm.

[55] Pagallo, G. and Haussler, D. Boolean feature discovery in empirical learning. Machine

Learning, 5(1), 1990.

[56] Parish, Y. I. and Müller, P. 2001. Procedural modeling of cities. In Proceedings of the 28th Annual Conference on Computer Graphics and interactive Techniques SIGGRAPH '01. ACM, New York, NY, 301-308. DOI= http://doi.acm.org/10.1145/383259.383292.

[57] Pacheco, P. S. 1996 Parallel Programming with MPI. Morgan Kaufmann Publishers Inc.

[58] Perlin, K. 1985. An image synthesizer. SIGGRAPH Comput. Graph. 19, 3 (Jul. 1985), 287-296. DOI= http://doi.acm.org/10.1145/325165.325247.

[59] Perz, T. L-System 4. Available from http://www.geocities.com/tperz/L4Home.htm.

[60] Preparata, F. P. and Shamos, M. I. 1985 Computational Geometry: an Introduction. Springer-Verlag New York, Inc.

[61] Prusinkiewicz, P. and Lindenmayer, A. 1996 The Algorithmic Beauty of Plants. Springer-Verlag New York, Inc.

[62] Prusinkiewicz, P., Mark Hammel, Jim Hanan, and Radomir Mech. L-systems: from the theory to visual models of plants. In M. T. Michalewicz, editor, Proceedings of the 2nd CSIRO Symposium on Computational Challanges in Life Sciences. CSIRO Publishing, 1996.

[63] Prusinkiewicz, P. Score generation with L-systems. Proceedings of the 1986 International Computer Music Conference, pp. 455-457.

[64] Quinlan, J. R. Simplifying decision trees. International Journal of Man-Machine Studies, 27:221-234, 1987.

[65] Reeves, W. T. 1983. Particle Systems—a Technique for Modeling a Class of Fuzzy Objects. ACM Trans. Graph. 2, 2 (Apr. 1983), 91-108. DOI= http://doi.acm.org/10.1145/357318.357320.

[66] Robert, C. P. and Casella, G. 2005 Monte Carlo Statistical Methods (Springer Texts in Statistics). Springer-Verlag New York, Inc.

[67] Roden, T. and Parberry, I., From Artistry to Automation: A Structured Methodology for Procedural Content Generation. Available from http://www.eng.unt.edu/ian/pubs/SP1-roden-timothy.pdf.

[68] Rost, R. J. 2004 Opengl(R) Shading Language. Addison Wesley Longman Publishing Co., Inc.

[69] Sh. Available from http://libsh.org/.

[70] Shankel, Jason. Fractal Terrain Generation – Fault Formation. In Game Programming Gems. Rockland, Massachusetts, Charles River Media, 2000. 499 – 502.

[71] Shankel, Jason. Fractal Terrain Generation – Midpoint Displacement. In Game Programmming Gems. Rockland, Massachusetts, Charles River Media, 2000. 503-507.

[72] Shreiner, D. 1999 OpenGL Reference Manual: the Official Reference Document to Opengl, Version 1.2. 3rd. Addison-Wesley Longman Publishing Co., Inc.

[73] Sinclair, Brendan. NPD: Game industry reaches $12.5 billion in '06. GameSpot. Available from http://www.gamespot.com/news/6164101.html.

[74] Stiny G, 1980. Introduction to shape and shape grammars. Environment and Planning B 7(3) 343-351.

[75] Stiny, G., and Mitchell, W.J. 1978. The palladian grammar. Environment and Planning B 5, 5-18.

[76] Sun, J., Yu, X., Baciu, G., and Green, M. 2002. Template-based generation of road networks for virtual city modeling. In Proceedings of the ACM Symposium on Virtual Reality Software and Technology (Hong Kong, China, November 11 - 13, 2002). VRST '02. ACM, New York, NY, 33-40. DOI= http://doi.acm.org/10.1145/585740.585747.

[77] Thiemann, F., and Sester, M. 2004. Segmentation of Buildings for 3D-Generalisation. In ICA Workshop on Generalisation and Multiple Representation. 20-21.

[78] Von Neumann, J. 1951. Various techniques used in connection with random digits. Nat. Bur. ~ Stand. Appl. Math Ser. 12, 36-38.

[79] Wei, L. and Levoy, M. 2000. Fast texture synthesis using tree-structured vector quantization. In Proceedings of the 27th Annual Conference on Computer Graphics and interactive Techniques International Conference on Computer Graphics and Interactive Techniques. ACM Press/Addison-Wesley Publishing Co., New York, NY, 479-488. DOI=http://doi.acm.org/10.1145/344779.345009.

[80] Weiss, S. and Indurkhya, N. Reduced complexity rule induction. In Proceedings of the 12th International Jount Conference on Artificial Intelligence, Sydney, Australia, 1991. Morgan Kaufmann.

[81] Witkin, A. and Kass, M. 1991. Reaction-diffusion textures. SIGGRAPH Comput. Graph. 25, 4 (Jul. 1991), 299-308. DOI= http://doi.acm.org/10.1145/127719.122750.

[82] Wonka, P., Wimmer, M., Sillion, F., and Ribarsky, W. 2003. Instant architecture. ACM Trans. Graph. 22, 3 (Jul. 2003), 669-677. DOI= http://doi.acm.org/10.1145/882262.882324.

Appendix A – Lex and Yacc Files

Lex File

```
%{

#include <stdio.h>
#include <stdlib.h>
#include <string>
#include "a2.h"

#include "common.h"

%}

%option noyywrap

%%
"Subdiv" { yylval.id = SUBDIV; return FUNCNAME; }
"Comp" { yylval.id = COMP; return FUNCNAME; }
"Size" { yylval.id = SIZE; return FUNCNAME; }
"Replace" { yylval.id = REPLACE; return FUNCNAME; }
"rand" { yylval.id = RAND; return FUNCNAME; }
"end" { return '@'; }
"Texture" { yylval.id = TEXTURE; return FUNCNAME; }
"Repeat" { yylval.id = REPEAT; return FUNCNAME; }
"Translate" { yylval.id = TRANSLATE; return FUNCNAME; }
"RX" { yylval.id = RX; return FUNCNAME; }
"RY" { yylval.id = RY; return FUNCNAME; }
"RZ" { yylval.id = RZ; return FUNCNAME; }
"\[" { yylval.id = PUSH; return FUNCNAME; }
"\]" { yylval.id = POP; return FUNCNAME; }
"Model" { yylval.id = MODEL; return FUNCNAME; }
"Assign" { yylval.id = ASSIGN; return FUNCNAME; }

"->" { return '~'; }
\( { return yytext[0]; }
\) { return yytext[0]; }
, { return yytext[0]; }
\{ {return yytext[0]; }
\} {return yytext[0]; }
\| {return yytext[0]; }
: {return yytext[0]; }
```

```
[_a-zA-Z]+[.0-9_a-zA-Z]* { strcpy(yylval.word, yytext); return WORDVAL; }
[+-/*><=!] { return yytext[0]; }
[~]?[0-9]*[.[0-9]*]? {
        if (yytext[0] == '~')
        {
                yytext[0] = '0';
                printf("!!%s\n", yytext);
                yylval.val = -(float)atof(yytext);
        }
        else
        {
                yylval.val = (float)atof(yytext);
        }
        return FLOATVAL;
}
; { return yytext[0]; }
[\^] { return '^'; }
[\t\n ]+ { }

%%
```

Yacc File

```
%{

  #include <stdio.h>
  #include <stdlib.h>
  #include <string.h>
  #include "a2.h"

  extern int yylex(void);
  extern void yyerror(char *msg);

  RuleList gRuleList;
  int gNumSlots = 0;
%}

%union {
  int id;
  char word[256];
  char var;
  float val;
  ExprItem exprItem;
  Expression expression;
  ParamList paramList;
  Function function;
  Action action;
  ReplacementList replacementList;
  ActionList actionList;
  Rule rule;
  RuleList ruleList;
  ActionLists actionLists;
};

%left '<' '>' '=' '!'
%left '+' '-'
%left '*' '/'
%left '(' ')'

%token<id> FUNCNAME
%token<word> WORDVAL
%token<var> VARIABLE
%token<var> OPERATOR
%token<val> FLOATVAL
%type<exprItem> OPERAND
%type<expression> EXPRESSION
%type<paramList> PARAMS
%type<function> FUNC
%type<action> ACTION
%type<replacementList> REPLACEMENTS
%type<actionList> ACTIONLIST
%type<rule> RULE
%type<ruleList> RULES
%type<actionLists> ACTIONLISTS

%%
```

```
program: RULES '@' {
  gRuleList = $1;
  return 0;
}
;

RULES:
RULES RULE {
	$$.rules[$$.numRules++] = $2;
}
| RULE {
	$$.rules[$$.numRules++] = $1;
}
;

RULE:
FLOATVAL ':' WORDVAL '~' ACTIONLISTS {
	strcpy($$.predecessor, $3);
	$$.id = (int)($1 + 0.5);
	$$.actionLists = new ActionLists;
	*($$.actionLists) = $5;

	if ($$.actionLists->lists[0].probability != 0)
	{
		$$.actionLists->slot = gNumSlots++;
	}
}
| FLOATVAL ':' WORDVAL ':' EXPRESSION '~' ACTIONLISTS {
	strcpy($$.predecessor, $3);
	$$.id = (int)($1 + 0.5);
	$$.actionLists = new ActionLists;
	*($$.actionLists) = $7;
	$$.requirement = $5;

	if ($$.actionLists->lists[0].probability != 0)
	{
		$$.actionLists->slot = gNumSlots++;
	}
}
;

ACTIONLISTS:
ACTIONLISTS '~' ACTIONLIST ':' FLOATVAL {
	$$.lists[$$.numLists++] = $3;
	$$.lists[$$.numLists-1].probability = $5;

	if ($$.numLists > 1)
	{
		$$.lists[$$.numLists-1].probability += $$.lists[$$.numLists-2].probability;
	}
}
| ACTIONLIST ':' FLOATVAL {
	$$.lists[$$.numLists++] = $1;
	$$.lists[$$.numLists-1].probability = $3;
}
| ACTIONLIST {
```

```
		$$.lists[$$.numLists++] = $1;
}
;

ACTIONLIST:
ACTIONLIST ACTION {
		$$.actions[$$.numActions++] = $2;
}
| ACTION {
		$$.actions[$$.numActions++] = $1;
}
;

ACTION:
FUNC '{' REPLACEMENTS '}' {
		$$.function = $1;
		$$.replacementList = $3;
}
| FUNC {
		$$.function = $1;
}
;

REPLACEMENTS:
REPLACEMENTS '|' WORDVAL {
		strcpy($$.replacements[$$.numReplacements++], $3);
}
| WORDVAL {
		strcpy($$.replacements[$$.numReplacements++], $1);
}
;

FUNC:
FUNCNAME '(' PARAMS ')' {
		$$.id = $1;
		$$.paramList = $3;

		if ($$.id == RAND)
		{
			$$.slot = gNumSlots++;
		}
}
| FUNCNAME {
		$$.id = $1;
		$$.paramList.numParams = 0;

		if ($$.id == RAND)
		{
			$$.slot = gNumSlots++;
		}
}
;

PARAMS:
PARAMS ',' EXPRESSION {
		$$.params[$$.numParams++] = $3;
```

```
}
| EXPRESSION {
	$$.params[$$.numParams++] = $1;
}
| {
	$$.numParams = 0;
}
;

EXPRESSION:
'(' EXPRESSION ')' {
	for (int i = 0; i < $2.numItems; i++)
	{
		$$.items[$$.numItems++] = $2.items[i];
	}
}
| EXPRESSION '*' EXPRESSION {
	for (int i = 0; i < $3.numItems; i++)
	{
		$$.items[$$.numItems++] = $3.items[i];
	}
	$$.items[$$.numItems++].op = '*';
}
| EXPRESSION '/' EXPRESSION {
	for (int i = 0; i < $3.numItems; i++)
	{
		$$.items[$$.numItems++] = $3.items[i];
	}
	$$.items[$$.numItems++].op = '/';
}

| EXPRESSION '+' EXPRESSION {
	for (int i = 0; i < $3.numItems; i++)
	{
		$$.items[$$.numItems++] = $3.items[i];
	}
	$$.items[$$.numItems++].op = '+';
}
| EXPRESSION '-' EXPRESSION {
	for (int i = 0; i < $3.numItems; i++)
	{
		$$.items[$$.numItems++] = $3.items[i];
	}
	$$.items[$$.numItems++].op = '-';
}
| EXPRESSION '>' EXPRESSION {
	for (int i = 0; i < $3.numItems; i++)
	{
		$$.items[$$.numItems++] = $3.items[i];
	}
	$$.items[$$.numItems++].op = '>';
}
| EXPRESSION '<' EXPRESSION {
	for (int i = 0; i < $3.numItems; i++)
	{
		$$.items[$$.numItems++] = $3.items[i];
```

```
		}
		$$.items[$$.numItems++].op = '<';
}
| EXPRESSION '=' EXPRESSION {
		for (int i = 0; i < $3.numItems; i++)
		{
			$$.items[$$.numItems++] = $3.items[i];
		}
		$$.items[$$.numItems++].op = '=';
}
| EXPRESSION '!' EXPRESSION {
		for (int i = 0; i < $3.numItems; i++)
		{
			$$.items[$$.numItems++] = $3.items[i];
		}
		$$.items[$$.numItems++].op = '!';
}
| OPERAND {
		$$.items[$$.numItems++] = $1;
		$$.items[$$.numItems-1].op = 0;
}
;

OPERAND:
WORDVAL {
		$$.type = EI_VARIABLE; strcpy($$.name,$1);
}
| FLOATVAL '^' {
		$$.type = EI_RELATIVE; $$.val = $1;
}
| FLOATVAL {
		$$.type = EI_FLOAT; $$.val = $1;
}
| FUNC {
		$$.type = EI_FUNC;
		$$.function = new Function;
		*($$.function) = $1;
}
;

%%

#include <stdio.h>
#include "main.h"


int main(int argc, char *argv[])
{
		return MyMain(argc, argv);
}

void yyerror(char *msg)
{
		printf("%s\n", msg);
}
```

Appendix B – Data Structures to Store Grammar Rules

```
#ifndef a2_h
#define a2_h

#define EI_RELATIVE 0
#define EI_FLOAT 1
#define EI_FUNC 2
#define EI_VARIABLE 3

#define COMP 0
#define SUBDIV 1
#define SIZE 2
#define RAND 3
#define REPLACE 4
#define TEXTURE 5
#define REPEAT 6
#define PUSH 7
#define POP 8
#define TRANSLATE 9
#define RX 10
#define RY 11
#define RZ 12
#define MODEL 13
#define ASSIGN 14
#define NUMFUNCTIONS 15

#define MAXRULES 80
#define MAXACTIONLISTS 5
#define MAXACTIONS 10
#define NAMELENGTH 15
#define MAXREPLACEMENTS 10
#define MAXPARAMS 10
#define MAXEXPRITEMS 15

class Function;

class ExprItem
{
 public:
            char op;
            int type;
            float val;
```

```
        char name[NAMELENGTH];
        Function *function;
};

class Expression
{
 public:
        ExprItem items[MAXEXPRITEMS];
        int numItems;
};

class ParamList
{
 public:
        Expression params[MAXPARAMS];
        int numParams;
};

class Function
{
 public:
        int id;
        ParamList paramList;
        int slot;
};

class ReplacementList
{
 public:
        char replacements[MAXREPLACEMENTS][NAMELENGTH];
        int numReplacements;
};

class Action
{
 public:
        Function function;
        ReplacementList replacementList;
};

class ActionList
{
 public:
        Action actions[MAXACTIONS];
        int numActions;
        float probability;
};

class ActionLists
{
 public:
       ActionList lists[MAXACTIONLISTS];
       int numLists;
       int slot;
};
```

```
class Rule
{
 public:
        int id;
        char predecessor[NAMELENGTH];
        Expression requirement;
        ActionLists *actionLists;
};

class RuleList
{
 public:
        Rule rules[MAXRULES];
        int numRules;
};

#endif
```

Appendix C – Test Grammars

Grammar 1

```
1: lot          ->Size(80,40 * rand(0.8,2),40) {new_lot}
2: new_lot      ->Subdiv(Y,2^,1^) {bottom | top} : 0.8
                ->Subdiv(Y,2^,1^) {bottom | terminal} : 0.2
3: top          ->Comp("sidefaces") {top_faces}
4: top_faces    ->Repeat(X, Scope.sx / rand(4,8)) {window_div}
5: window_div   ->Subdiv(X, 1^, 4^, 1^) {terminal | window_divb | terminal}
6: window_divb  ->Subdiv(Y, 1^, 4^, 1^) {terminal | window_fin | terminal}
7: bottom       ->Subdiv(X, 4^, 2^, 1^) {left | door | right} : 0.8
                ->Subdiv(X, 1^, 4^) {door | right} : 0.2
8: left         ->Comp("sidefaces") {left_faces}
9: left_faces   ->Subdiv(X, 1^, 1^) {b_window | b_window}
10: b_window    ->Subdiv(X, 1^, 4^, 1^) {terminal | b_win_main | terminal}
11: b_win_main  ->Subdiv(Y, 1^, 4^, 1^) {terminal | b_win_fin | terminal}
12: door        ->Subdiv(Y, 5^, 1^) {door_main | terminal}
end
```

Grammar 2

```
1: lot     ->Size(100, 70 * rand(0.7,1), 70) {new_lot}
2: new_lot      ->Subdiv(X, 1^, rand(30,60), 1^) {wing | main | wing} : 0.5
                ->Subdiv(X, rand(30,60), 1^) {main | wing} : 0.25
                ->Subdiv(X, 1^, rand(30,60)) {wing | main} : 0.25
3: main         ->Size(1^, Scope.sy * rand(0.3,0.8), 1^) {new_main}
3: new_main     ->Subdiv(Y, 3^, 1^) {main_bot | main_top}
4: main_bot     ->Subdiv(X, 1^, 1^, 1^) {main_left | main_door | main_right}
5: main_door    ->Subdiv(Y, 5^, 1^) {main_door_f | terminal}
6: main_top     ->Subdiv(X, 1^, 1^, 1^, 1^) {window | window | window | window}
7: window       ->Subdiv(X, 1^, 4^, 1^) {terminal | window_c | terminal}
8: window_c     ->Subdiv(Y, 1^, 4^, 1^) {terminal | window_f | terminal}
```

```
9: wing          ->Subdiv(Y, 2^, 8^) {wing_bot | wing_top}
10: wing_bot     ->Subdiv(X, 1^, 2^, 1^) {wing_bot_left | wing_door | wing_bot_right}
11:wing_top      ->Comp("sidefaces") {wt_faces}
12:wt_faces      ->Repeat(Y, 20) {wt_floors}
13:wt_floors     ->Subdiv(Y, 1^, 2^, 2^, 2^, 1^) {terminal | big_window | big_window | big_window | terminal}
14:big_window    ->Subdiv(X, 1^, 5^, 5^, 1^) {terminal | bw_f | bw_f | terminal}
end
```

Grammar 3

```
1: lot           ->Size(1^, 200 * rand(0.7, 1), 1^) {new_lot}
2: new_lot       ->Subdiv(Y, 2^, 8^) {lot_bot | lot_top}
3: lot_bot       ->Subdiv(X, 1^, 2^, 1^) {terminal | door | terminal}
4: door          ->Subdiv(Y, 5^, 1^) {door_bot | terminal}
5: door_bot      ->Subdiv(X, 1^, 5^, 1^) {terminal | door_f | terminal}
6: lot_top       ->Repeat(Y, Scope.sy / rand(5,10)) { floors}
7: floors        ->Comp("sidefaces") {floor_faces}
8: floor_faces   ->Subdiv(X, 1^, 1^, 1^, 1^) {window | window | window | window}
9: window        ->Subdiv(X, 1^, 2^, 2^, 1^) {terminal | win_t | win_t | terminal}
10:win_t         ->Subdiv(Y, 1^, 5^, 1^) {terminal | win_f | terminal}
11: win_f        ->Subdiv(Y, 1^, 1^, 1^, 1^) {win_ff | win_ff | win_ff | win_ff}
end
```

Grammar 4

```
1: lot           ->Size(50 * rand(0.5,1), 50 * rand(0.5, 1), 50 * rand(0.5, 1)) {new_lot}
2: new_lot       ->Subdiv(X, 1^, 1^) {ra | ra}
3: ra            ->Subdiv(X, 1^, 1^) {rb | rb}
4: rb            ->Subdiv(Y, 1^, 1^) {rc | rc}
5: rc            ->Subdiv(Y, 1^, 1^) {rd | rd}
6: rd            ->Subdiv(X, 1^, 1^) {re | re}
7: re            ->Subdiv(X, 1^, 1^) {rf | rf} : 0.8
                 ->Replace(final) : 0.2
8: rf            ->Subdiv(Y, 1^, 1^) {rg | rg} : 0.8
                 ->Replace(final) : 0.2
9: rg            ->Subdiv(Y, 1^, 1^) {rh | rh} : 0.8
                 ->Replace(final) : 0.2
10: rh           ->Subdiv(X, 1^, 1^) {ri | ri}
11: ri           ->Comp("sidefaces") {final}
```

```
end

Grammar 5
1: lot          ->Size(70 * rand(0.5, 1), 70 * rand(0.5, 1), 70 * rand(0.5, 1)) {nl}
2: nl           ->Subdiv(X, 1^, 1^, 1^) {ra | ra | ra}
3: ra           ->Subdiv(Y, 1^, 1^, 1^) {rb | rb | rb}
4: rb           ->Subdiv(X, 1^, 1^, 1^) {rc | rc | rc}
5: rc           ->Subdiv(Y, 1^, 1^, 1^) {rd | rd | rd}
6: rd           ->Subdiv(X, 1^, 1^, 1^) {re | re | re} : 0.8
                ->Replace(final) : 0.2
7: re           ->Subdiv(Y, 1^, 1^, 1^) {rf | rf | rf} : 0.8
                ->Replace(final) : 0.2
8: rf           ->Subdiv(X, 1^, 1^, 1^) {rg | rg | rg} : 0.8
                ->Replace(final) : 0.2
9: rg           ->Comp("sidefaces") {final}
end

Grammar 6
1: lot          ->Size(1^, 50 * rand(0.7, 1.3), 1^) {new_lot}
2: new_lot      ->Subdiv(Y, 1^, 1^, 1^) {bot | mid | top}
3: mid          ->Comp("sidefaces") {new_mid}
4: new_mid      ->Subdiv(X, 1^, 1^, 1^, 1^) {middiv | middiv | middiv | middiv | middiv}
5: middiv       ->Subdiv(Y, 1^, 1^, 1^, 1^) {middivb | middivb | middivb | middivb | middivb}
6: middivb      ->Subdiv(X, 1^, 5^, 1^) {terminal | win_t | terminal}
7: win_t        ->Subdiv(Y, 1^, 5^, 1^) {terminal | win_f | terminal}
8: bot          ->Subdiv(X, 1^, 4^, 1^) {side | door | side}
9: top          ->Subdiv(Y, 1^, 1^, 1^) {floor | floor | floor}
10:floor        ->Comp("sidefaces") {floor_faces}
11:floor_faces  ->Subdiv(X, 1^, 1^, 1^, 1^) {topdiv | topdiv | topdiv | topdiv | topdiv}
12:topdiv       ->Subdiv(Y, 1^, 1^) {topdivb | topdivb}
13:topdivb      ->Size(1^, 1^, rand(0.7, 1.3)) {topf}
14:topf         ->Subdiv(X, 1^, 1^, 1^, 1^) {terminal | terminal | terminal | terminal | terminal} : 0.5
                ->Subdiv(Y, 1^, 1^, 1^, 1^) {terminal | terminal | terminal | terminal | terminal} : 0.5
15: side        ->Comp("sidefaces") {side_faces}
16: side_faces  ->Replace(new_mid)
end
```

Grammar 7

```
1: lot          ->Size(100, 100 * rand(0.7, 1.3), 50) {new_lot}
2: new_lot      ->Subdiv(X, 1^, 2^, 1^) {branch | main | branch} : 0.8
                ->Subdiv(X, 1^, 2^, 1^) {e | main | e} : 0.1
                ->Subdiv(X, 3^, 1^) {main | e} : 0.1
3: main         ->Subdiv(Y, 8^, 1^) {main_bot | tip} : 0.5
                ->Replace(main_bot) : 0.5
4: tip          ->Subdiv(X, 1^, 1^, 1^) {e | tip_t | e}
5: tip_t        ->Subdiv(Z, 1^, 1^, 1^) {e | tip_f | e}
6: tip_f        ->Subdiv(Y, 2^, 1^) {terminal | tip_top} : 0.5
                ->Replace(terminal) : 0.5
7: tip_top      ->Subdiv(X, 1^, 1^, 1^) {e | tip_top_t | e}
8: tip_top_t    ->Subdiv(Z, 1^, 1^, 1^) {e | tip_top_f | e}

90: main_bot    ->Comp("sidefaces") {mbt}
9: mbt          ->Subdiv(X, 1^, 1^) {mb_half | mb_half}
10: mb_half     ->Subdiv(Y, 1^, 1^) {mb_quart | mb_quart}
11: mb_quart    ->Subdiv(X, 1^, 5^, 1^) {terminal | mb_quart_t | terminal}
12: mb_quart_t  ->Subdiv(Y, 1^, 5^, 1^) {terminal | mb_quart_f | terminal}
13: mb_quart_f  ->Subdiv(X, 1^, 1^) {mbq_a | mbq_a}
14: mbq_a       ->Subdiv(Y, 1^, 1^) {mbq_b | mbq_b}
15: mbq_b       ->Subdiv(X, 1^, 1^) {mbq_c | mbq_c}
16: mbq_c       ->Subdiv(Y, 1^, 1^) {mbq_d | mbq_d}
17: branch      ->Comp("sidefaces") {branch_faces}
18: branch_faces->Subdiv(X, 1^, 1^, 1^) {branch_faces_f | branch_faces_f | branch_faces_f}
19: branch_faces_f->Replace(mbt)
99:     e               ->Replace(empty)
end
```

Grammar 8

```
1: lot          ->Size(50, 100 * rand(0.8,1.2), 50) {new_lot}
2: new_lot              ->Subdiv(X, 1^, 1^, 2^, 1^, 1^) {branchb | brancha | branch_s | brancha | branchb} : 0.2
                        ->Subdiv(X, 1^, 1^, 2^, 1^, 1^) {brancha | brancha | branch_s | brancha | brancha} : 0.2
                        ->Subdiv(X, 1^, 1^, 2^, 1^, 1^) {branchb | branchb | branch_s | branchb | branchb} : 0.2
                        ->Subdiv(X, 1^, 1^, 2^, 1^, 1^) {e | brancha | branch_s | brancha | e} : 0.2
                        ->Subdiv(X, 1^, 1^, 2^, 1^, 1^) {e | e | branch_s | e | e} : 0.2
3: brancha              ->Size(1^, Scope.sy * rand(0.7,0.9), 1^) {branch_s}
```

```
4: branchb              ->Size(1^, Scope.sy * rand(0.3,0.5), 1^) {branch_s}
5: branch_s       ->Repeat(Y, rand(20,25)) {branch_floors}
6: branch_floors  ->Comp("sidefaces") {floors_a}
7: floors_a             ->Subdiv(X, 1^, 1^, 1^, 1^, 1^) {win | win | win | win | win}
8: win                  ->Subdiv(Y, 1^, 1^, 1^, 1^, 1^) {win_b | win_b | win_b | win_b}
9: win_b                ->Subdiv(X, 1^, 5^, 1^) {terminal | win_t | terminal}
10:win_t                ->Subdiv(Y, 1^, 5^, 1^) {terminal | win_f | terminal}
99: e             ->Replace(empty)
end
```

Grammar 9

```
1: lot            ->Size(100, 100 * rand(0.7,1), 70) {new_lot}
2: new_lot        ->Subdiv(Y, 1^, 2^) {bottom | top}
3: bottom         ->Subdiv(X, 1^, 2^) {entrance | columns}
4: entrance       ->Subdiv(Y, 2^, 1^) {entrance_bot | terminal}
5: entrance_bot->Subdiv(X, 1^, 1^, 1^) {terminal | door | terminal}
6: columns        ->Subdiv(X, 2^, 1^, 4^, 1^) {e | column_t | e | column_t} : 0.5
                  ->Replace(terminal) : 0.5
7: column_t->Subdiv(Z, 1^, 5^, 1^) {column | e | column} : 0.5
                  ->Replace(column) : 0.5
8: top            ->Subdiv(Y, 1^, 1^, 1^) {floor | floor | floor}
9: floor          ->Subdiv(Y, 1^, 7^) {balcony | floor_main}
10: balcony       ->Size(1^, 1^, 1.2^) {balcony_f}
11: floor_main    ->Subdiv(X, 1^, 1^, 1^, 1^, 1^) {floor_door | window | window | window | window}
12: floor_door    ->Subdiv(X, 1^, 1^, 1^) {terminal | fdoor_t | terminal}
13: fdoor_t       ->Subdiv(Y, 5^, 1^) {fdoor_f | terminal}
14: window        ->Subdiv(X, 1^, 6^, 1^) {terminal | window_t | terminal}
15: window_t      ->Subdiv(Y, 1^, 6^, 1^) {terminal | window_f | terminal}
16: window_f      ->Subdiv(X, 1^, 1^, 1^, 1^, 1^) {win_a | win_a | win_a | win_a | win_a}
17: win_a         ->Subdiv(Y, 1^, 1^, 1^, 1^, 1^) {win_b | win_b | win_b | win_b | win_b}
18: win_b         ->Comp("sidefaces") {win_faces}
19: win_faces     ->Subdiv(Y, 1^, 1^, 1^) {terminal | terminal | terminal}
99:       e             ->Replace(empty)
end
```

Grammar 10

```
1:lot->Size(1^,30,1^) Subdiv(Z,Scope.sz*rand(0.3,0.5),1^) {facades | sidewings}
```

```
2:sidewings->Subdiv(X,Scope.sx*rand(0.2,0.6),1^) {sidewing|empty}
Subdiv(X,1^,Scope.sx*rand(0.2,0.6)) {empty|sidewing}
3:sidewing->Size(1^,1^,Scope.sz*rand(0.4,1.0)) {facades} : 0.5
->Size(1^,Scope.sy*rand(0.2,0.9), Scope.sz*rand(0.4,1.0)) {facades} : 0.3
->Replace(empty) : 0.2
4:facades->Comp(sidefaces) {facade}
5:facade->Subdiv(Y, 5, 1^, 5) {groundfloor | floors | topfloors}
6:groundfloor->Subdiv(X,2^,1^,2^) {groundtiles | entrance | groundtiles}
7:floors->Repeat(Y, 4) {floor}
8:floor->Repeat(X, 4) {tile}
9:entrance->Texture(officedoor.bmp)
10:tile->Subdiv(Y,1^,5^,1^) {tile_a | win_t | tile_a}
11:win_t->Subdiv(X,1^,5^,1^) {tile_a | win_f | tile_a}
12:win_f:Scope.sx>1.3->Texture(officewindow.bmp)
12:win_f:Scope.sx<1.3->Replace(tile_a)
13:tile_a->Texture(greenmarble.jpg, stretch)
end
```

Grammar 11

```
1:      lot->Size(20,30 * rand(0.7,1),20) {new_lot}
2:      new_lot->Replace(wings) : 0.3
        ->Replace(straight) : 0.2
        ->Replace(tiered) : 0.2
        ->Replace(else) : 0.3
3:      wings->Subdiv(Z, Scope.sz * rand(0.2, 0.5), 1^) { wings_back | wings_front }
4:      wings_front->Subdiv(X, Scope.sx * rand(0.2, 0.4), 1^) { wf_left | wf_right }
5:      wf_right->Size(1^, Scope.sy * rand(0.2, 0.8), 1^) {wf_rights}
6:      tiered->Subdiv(Y, Scope.sy * 0.5, 1^) {t_bottom | t_top}
7:      t_top->Size(Scope.sx*rand(0.5,0.5),1^,Scope.sz * rand(0.5,0.5)) {t_tops}
8:      straight->Size(Scope.sx * rand(0.5,0.7), Scope.sy * rand(1.0,1.3), Scope.sz * rand(0.5, 0.7)) {tall}
9:      else->Subdiv(X, 1^, Scope.sx * rand(0.2,0.4), 1^) {else_left | else_mid | else_right}
10:     else_mid->Subdiv(Y, 1^, Scope.sy * rand(0.1, 0.5), 1^) {em_bottom | em_mid | em_top }
11:     em_mid->Subdiv(Z, 1^, Scope.sz * rand(0.2, 0.4), 1^) {emm_back | emm_mid | emm_front}
13:     em_bottom->Replace(empty)
14:     em_top->Replace(empty)
15:     emm_back->Replace(empty)
16:     emm_front->Replace(empty)
```

```
17:     else_left->Size(1^, Scope.sy*rand(0.7,1.3), 1^) {el_sized}
18:     else_right->Size(1^, Scope.sy*rand(0.7,1.3), 1^) {er_sized}

end
```

Grammar 2^n

```
1:      lot->Size(50,50,50) {nl}
2:      nl->Subdiv(X, 1^, 1^) {a|a}
3:      a->Subdiv(X,1^,1^) {b|b}
4:      b->Subdiv(X,1^,1^) {c|c}
5:      c->Subdiv(X,1^,1^) {d|d}
6:      d->Subdiv(X,1^,1^) {e|e}
7:      e->Subdiv(X,1^,1^) {f|f}
8:      f->Subdiv(X,1^,1^) {g|g}
9:      g->Subdiv(X,1^,1^) {h|h}
10:     h->Subdiv(X,1^,1^) {i|i}
11:     i->Subdiv(X,1^,1^) {j|j}
12:     j->Subdiv(X,1^,1^) {k|k}
13:     k->Subdiv(X,1^,1^) {l|l}
14:     l->Subdiv(X,1^,1^) {m|m}
15:     m->Subdiv(X,1^,1^) {n|n}
16:     n->Subdiv(X,1^,1^) {o|o}
17:     o->Subdiv(X,1^,1^) {p|p}
end
```

Grammar 3^n

```
1:      lot->Size(50,50,50) {nl}
2:      nl->Subdiv(X, 1^, 1^,1^) {a|a|a}
3:      a->Subdiv(X,1^,1^,1^) {b|b|b}
4:      b->Subdiv(X,1^,1^,1^) {c|c|c}
5:      c->Subdiv(X,1^,1^,1^) {d|d|d}
6:      d->Subdiv(X,1^,1^,1^) {e|e|e}
7:      e->Subdiv(X,1^,1^,1^) {f|f|f}
8:      f->Subdiv(X,1^,1^,1^) {g|g|g}
9:      g->Subdiv(X,1^,1^,1^) {h|h|h}
10:     h->Subdiv(X,1^,1^,1^) {i|i|i}
11:     i->Subdiv(X,1^,1^,1^) {j|j|j}
```

```
end

Grammar 4^n
1:      lot->Size(50,50,50) {nl}
2:      nl->Subdiv(X, 1^, 1^,1^,1^) {a|a|a|a}
3:      a->Subdiv(X,1^,1^,1^,1^) {b|b|b|b}
4:      b->Subdiv(X,1^,1^,1^,1^) {c|c|c|c}
5:      c->Subdiv(X,1^,1^,1^,1^) {d|d|d|d}
6:      d->Subdiv(X,1^,1^,1^,1^) {e|e|e|e}
7:      e->Subdiv(X,1^,1^,1^,1^) {f|f|f|f}
8:      f->Subdiv(X,1^,1^,1^,1^) {g|g|g|g}
9:      g->Subdiv(X,1^,1^,1^,1^) {h|h|h|h}
end
```