Brock University

Department of Computer Science

Automatic Evolution of Conceptual Building Architectures

Corrado Coia
Technical Report # CS-11-06
May 2011

Brock University
Department of Computer Science
St. Catharines, Ontario
Canada L2S 3A1
www.cosc.brocku.ca

Automatic Evolution of Conceptual Building Architectures

Corrado Coia

Computer Science

Submitted in partial fulfillment
of the requirements for the degree of

Masters of Science

Faculty of Computer Science, Brock University
St. Catharines, Ontario

Abstract

This thesis describes research in which genetic programming is used to automatically evolve shape grammars that construct three dimensional models of possible external building architectures. A completely automated fitness function is used, which evaluates the three dimensional building models according to different geometric properties such as surface normals, height, building footprint, and more. In order to evaluate the buildings on the different criteria, a multi-objective fitness function is used. The results obtained from the automated system were successful in satisfying the multiple objective criteria as well as creating interesting and unique designs that a human-aided system might not discover. In this study of evolutionary design, the architectures created are not meant to be fully functional and structurally sound blueprints for constructing a building, but are meant to be inspirational ideas for possible architectural designs. The evolved models are applicable for today's architectural industries as well as in the video game and movie industries. Many new avenues for future work have also been discovered and highlighted.

Acknowledgements

I would like to thank the following individuals and organizations for their tireless support:

- Brian J. Ross for over four years of excellent supervision, guidance, and funding on all the projects accomplished.
- Beatrice Ombuki-Berman and Michael Winter for their participation on the supervisory committee.
- Cale Fairchild and Robert Flack for their seemingly limitless knowledge on Linux systems, networking, and all unsolvable problems solved.
- Brock University for providing me with the privilege and the facilities for furthering my knowledge and accomplishing something great.

Thank you, and goodnight.

C.C.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Evolutionary Design of Building Architecture

Building design is a complex task which relies heavily on many different fields of study [38]. Architects create concepts for building designs according to the fundamental principles of architecture, such as beauty, symmetry, style, function, form and more [5, 38]. Even when building design is handled by a computer, an architect is still the key factor in the final design. The architect must ensure that, for example, the building is able to withstand a multitude of elements such as high winds and earthquakes, as well as ensure the structural integrity of the building.

One method in which computers are used to create designs is through the use of grammars. Grammars are able to procedurally encode a series of building instructions used to create a design [34]. An advantage of using grammars is that they hold sets of instructions which can be used multiple times in the construction of the object. They can also be fine-tuned to give automated variations within designs. The major disadvantage of grammars is that they are difficult and time-consuming to create and edit. Any minor change in the grammar can result in vastly significant changes in the final product. In addition to being time-consuming to create, grammars can be challenging to learn. As a result, architects are not necessarily skilled in the field of grammar programming. Moreover, detailed grammars tend to create more detailed designs, in which case even more time and trail-by-error discovery is needed to use them. When someone is tasked with the job of

creating a grammar, it can be difficult to visualize the required grammar structure which would be needed to create the particular building, especially if the building has a lot of complex details to it. As such, manually coding a grammar which will create such a complex building according to multiple and conflicting geometric properties can be extremely difficult.

Since computers are a viable option as a design tool, many studies and methods have been explored to speed-up this design process. One method which has been explored is combining computer-aided design with evolutionary computation. This has been done using genetic algorithms as well as genetic programming techniques. Many applications have been studied, for example, tables[14], artifical flowers[18], architectural structures[4] and more.

Architects could make use of an automated program which creates building designs such that they would use the system as a design tool to give them new ideas and inspiration. A possible use scenario could be as follows. An architect is hired to design a high-rise office building. They could use an automated system to generate a multitude of building ideas and concepts which could be incorporated into their design. Additionally, if one were to drive through a developing urban subdivision, one would find a series of similar looking housing. These houses are made in bulk with a minimal focus on aesthetic value. To solve this, the design system could to generate possible new ideas for unique houses, while minimizing the effort of designing them by hand.

In the entertainment industry, an automated building design system would be extremely valuable. The buildings rendered in movies and video games are artificial, and their visual appeal is a main criteria of their design. They are often seen in the background of chase scenes, overhead city sweeps, or just as generic houses and buildings along a street. In this case, the key people involved in the development of these rendered buildings are three dimensional modelers and designers. In this situation, a designer could use an automated building design system to create buildings according to specified shape criteria and other constraints, having the system create a suite of buildings all similar in style. Alternatively, an automated system could be run many times with different constraints and criteria to create a multitude of different buildings. These buildings can also be generated each time a player starts a new game, effectively creating a new immersive environment for the player, and re-kindling the players interest in the game.

1.2 Goals and Measuring Success

The goal of this thesis is to propose an evolutionary design system which uses genetic programming to evolve conceptual building architectures according to multiple geometric criteria. The genetic program will accomplish this by:

1. Automatically creating shape grammars.
2. Using multiple objective criteria.
3. Working with an established commercial product.

The resulting application accomplishes the following goals:

1. An automated design tool, using geometric criteria.
2. Evolving three dimensional models though the use of multiple criteria, which often conflict.
3. Automating the programming of shape grammars.

The goal of this research is to use genetic programming for the automatic creation of shape grammars which construct external three dimensional building models. These models will satisfy multiple geometric criteria as specified by the user. Since creating grammars is a challenging task which requires a great deal of skill and experience to create by hand, a goal of this system is to have a fully automated method of creating these shape grammars.

Many interactive evolutionary systems have been created which require human interaction to guide the system's process though each generation within the evolutionary process. Interactive systems still consume the users time, as well as removes the creation of non-human designs which is often seen in fully automated systems. Manually reviewing each evolved individual is mentally taxing on the user, as the user must review and assign a score to each of the individuals. This limits the number of individuals that can be generated by interactive evolution.

With the approach presented in this thesis, the design exploration is handled automatically by the evolutionary program. The resulting external building models are not intended to be structurally sound blueprints of a building, but a set of possible structures in which an architect or designer could use as conceptual designs or as inspirations. The models create a basis of what could possibly become a real design. Another benefit of this system is

that it allows a user without any knowledge of shape grammars and grammar programming to development grammars and their respective models.

In order to measure the success of evolution, the resulting three dimensional models are evaluated by multiple pre-determined fitness scores which evaluate the models on different geometric properties. However, the true measure of success comes from reviewing the final models. The user can then visually identify if the models have satisfied the design criteria and present interesting and inspirational design ideas. It should be noted that some solutions generated by the evolutionary system, even if satisfying the multiple criteria, may not be accepted by the user due to the users tastes or other personal preferences. This study does not evaluate the aesthetics of the resulting models.

The criteria used is to evaluate the resulting three dimensional models according to different geometric properties. For example, one geometric property of a sphere is that each polygon on the sphere has a unique surface normal, a unique vector representing the direction that the polygon is facing. Therefore one fitness criteria which models that geometric property of a sphere, would be to evaluate the building model based on the quantity of unique surface normals present. In this case, the higher the number of unique surface normals the building has, the closer it becomes to matching that property of a sphere. Other constraints can be specified such that the desired square footage or volume of the building is to lay withing the footprint of the lot which the building is meant to be constructed on.

When the evolutionary program is instructed to evolve the building according to its specified geometric criteria, the evolutionary process searches though the solution space in its mission to find a close match. With such a vast solution space of potential 3D models, the evolutionary program could take a great amount of time before it is able to find acceptable matches to the specified criteria. Search can be made more effective by introducing constraints that represent desired features. In order to limit the size of the solution space, and therefore reduce the amount of computation needed, specific features such as the height of the building can be encoded within the starting grammar. This prevents the genetic program from unnecessary search through irrelevant designs.

The spectrum of results which can be obtained by specifying different shape criteria can be specific and well-defined, or vague and open-ended. For example, by specifying criteria such that the resulting building surface must contain 95% unique surface normals, as well as being exactly 500 units tall

and fill a square footprint 75 units wide, would over-specify the criteria and limit the amount of possible results. However, it is possible to under specify the criteria. For example, by only specifying the criteria to maximize the amount of unique surface normals, the resulting buildings could consume vast areas and be wildly unstructured. Therefore a good balance of criteria is needed to create desired buildings with a healthy amount of diversity.

1.3 Overview of the Thesis

In the next section, background information on architectural design, shape grammars, genetic programming and the CityEngine program are covered. In Chapter 3 the created system is explained in detail, explaining the functionality and structure of the program and how it interacts with the commercial tool, CityEngine. Chapter 4 explains and and shows a set of basic experiments and their results which ran on the system using simple shape criteria, where Chapter 5 and 6 show a more advanced series of experiments and their results which involve shape matching. Chapter 7 follows up with a discussion on the results followed by the conclusion.

Chapter 2

Background

2.1 Genetic Programming

Genetic Programming (GP) is a form of evolutionary computation which attempts to solve problems by evolving computer programs or expressions. It does this by encoding possible solutions into tree-like structures which the GP then executes and evaluates according to fitness criteria [20, 21, 30].

2.1.1 Tree Structure

Each possible solution is represented within the GP as a tree-structure. A benefit to GP over other forms of evolutionary computation is that the tree structures have a variable length as opposed to a fixed length. This means that the GP has the freedom to evolve large individuals. However, to prevent the individuals from growing beyond useful and reasonable proportions, a maximum depth is often specified into the system.

Within the tree structure, there are terminal nodes and non-terminal nodes. Since the tree structure created is a possible solution to the problem, the GP will execute that tree as if it were a computer program in which terminal nodes represent variables and non-terminal nodes represent decisions or functions within the program. For example, if looking at a simple math function such as 1 + 2, the addition is be a non-terminal node as it is a function, and it would have two branches leading to terminal nodes, which are the two values in the equation.

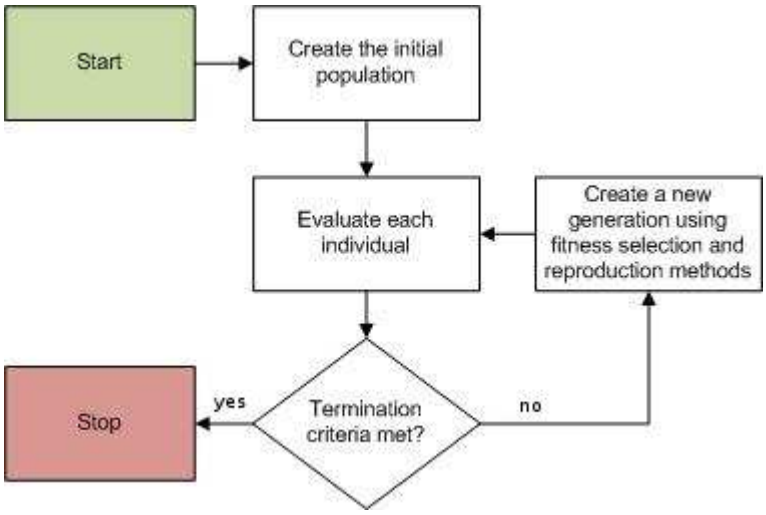


Figure 2.1: The structure and program flow of genetic programming.

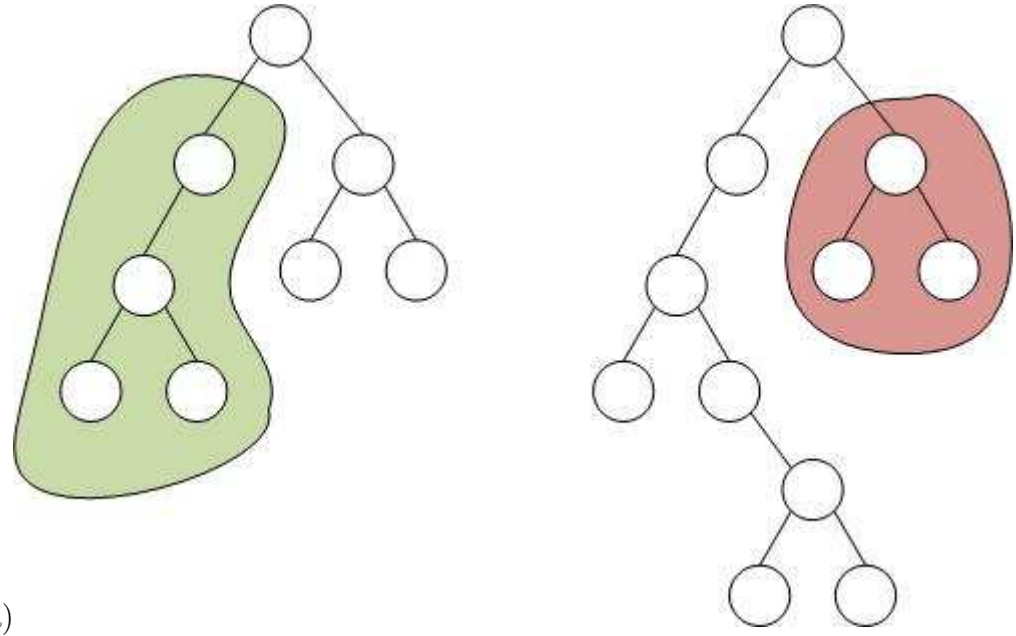
2.1.2 The Genetic Program Algorithm

A genetic program operates similarly to that of genetic algorithms, as shown in Figure 2.1. The GP creates an initial population, evaluates the individuals, then selects and reproduces individuals to create a new generation. This process is repeated until a termination criteria has been met. Termination can occur after a particular amount of generations have been evaluated, or when a target fitness value has been observed.

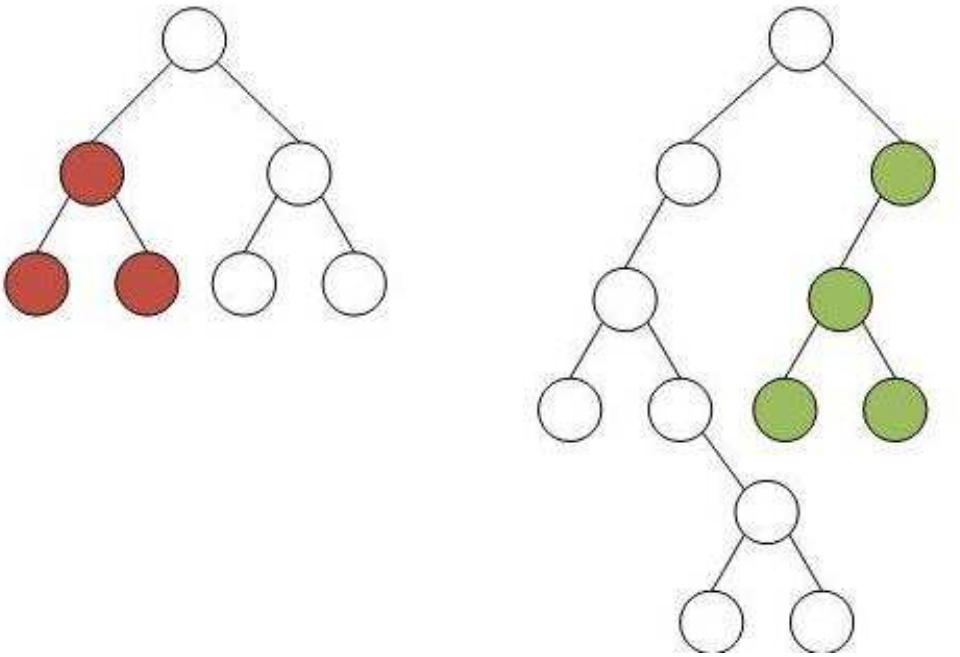
2.1.3 Reproduction Operators

Crossover takes two individuals and swaps portions of their tree structure with each other. It does that by selecting a random node within each individual, and then simply exchanging the subtrees rooted at those nodes. If an error arises such that one of the resulting new trees exceeds a maximum depth constraint, the GP will attempt to select new nodes and try again. The effects of crossover is shown in Figure 2.2.

It is important to note that only compatible nodes will be considered for swapping. This is ensured due to the strong typing of the parameters. As such, only parameters that are labeled as accepting integers, can be replaced by an integer type. All parameters within the grammar are strongly typed to ensure compatibility [24].



a)



b)

Figure 2.2: Image (a) shows the two parents which have been selected for crossover, and their respective subtrees selected for crossover. Image (b) shows the two parents and their resulting new trees after the crossover has taken place.

Mutation involves the alteration of a single subtree within the individual's tree structure. The GP does this by selecting a node and replacing the subtree rooted at that node with another randomly generated subtree of a compatible type. Similar to crossover, if the mutation operator creates an invalid tree, the GP will attempt to try another mutation on the same individual.

Another reproduction parameter which can be introduced to a GP is known as elitism. This reproduction operator selects k individuals which are the most-fit of the entire population, and copies them unaltered into the next generation.

2.1.4 Evaluation and Selection Methods

For every problem a GP is attempting to solve there needs to be a function which specifies one or more defined goals. This function evaluates each individual within the population and assigns them numerical values representing how well that individual accomplished one or more of the defined goals. This evaluation method is known as a fitness function, and the numerical values that are returned are known as fitness scores. After each individual within the population have been evaluated, the GP tends to select the fitter individuals for reproduction.

Many different selection methods can be used for the GP to determine which individuals are chosen for reproduction. The fitness selection method used in this study is tournament selection. Tournament selection selects k random individuals within the population (where k is the size of the tournament) and compares all selected individuals to determine which one of the k individuals has the best fitness score. The best fit individual is then selected for reproduction.

During the run of a GP, the individuals within the generations may tend to converge towards one common suboptimal solution. When this occurs, evolution becomes difficult since the individuals start to become identical, making crossover less effective. In order to help preserve a unique population, a penalty can be added to the individuals which have identical fitness scores. This penalty makes the identical individuals less favorable to the fitness selection method and is known as a diversity penalty.

2.1.5 Multi-Objective Fitness Evaluation Methods

When solving a problem, there can exist many different goals or objectives in which the user wishes to consider. Many of these goals may conflict with one another. This happens in many cases where having a high value in one fitness would result in a lower value in another fitness. There are many different methods of evaluating multi-objective problems [6, 9].

One method to tackle multiple objectives is to combine the scores of each fitness criteria into one numerical value which represents the individuals fitness. In addition to simply summing the different fitness scores, weights are generally added to each fitness. By adding weights, the user can manually determine which of the multiple goals are more important. The formula for weighted sum is:

$$fitness = f_1 * w_1 + f_2 * w_2 + \ldots + f_n * w_n$$

where f represents one fitness score, and w represents a weight value.

This recasts the problem into a single objective problem, that of optimizing fitness. The disadvantage of using this method is that the chosen weights will have a drastic effect on the final results generated by the GP. Weights can greatly bias the GP into favoring solutions which optimize those fitness score while ignoring the other criteria.

Another multi-objective evaluation method is known as Pareto ranking. Pareto ranking keeps the multiple fitness scores separate, as opposed to summing them together. It uses the notion of dominance to compare the fitness scores between individuals within the population. One individual is said to dominate another individual, if the first is not inferior to the second individual in all fitness goals, and there is be at least one fitness which is better [30].

The formula for Pareto dominance is as follows.

$$A\,dominates\,B \Rightarrow (\forall_{obj} f_{obj}(A) \leq f_{obj}(B)) \bigwedge (\exists_{obj} f_{obj} A < f_{obj} B)$$

Where

f_{obj} = Fitness Objective

A, B = Individuals within the population

Therefore, all the undominated individuals in the population are given a rank of 1. The rest of the population is then compared until the next optimal is found and given a rank of 2. This process is repeated until every individual within the population has been evaluated and compared.

Pareto ranking creates a sets of individuals which are not dominated by any others. A disadvantage to Pareto ranking is that it ceases to be effective when the number of fitness criteria exceed five dimensions [3].

Another multiple objective evaluation method is known as summed rank [3]. Similar to Pareto, summed rank keeps all the fitness scores separate. Each rank is evaluated separately on an objective basis. Summed rank ranks an individual based on its fitness score in relation to all individuals fitness scores of the same criteria. Once an individual has a rank for each fitness score, the ranks are summed together to create one fitness score for the individual.

Given fitnesses for a k-objective problem:

$$[f_1, f_2, \ldots, f_k]$$

Each fitness f_i has its rank r_i determined by

$$[r_1, r_2, \ldots, r_k]\ (1 \leq r_i \leq N), N = populationSize,$$

Then the summed rank is:

$$fitness = \sum_{i=1}^{k} r_i$$

A variation is to normalize each rank before summation, for example $[\frac{r_i}{R_i}, \frac{r_2}{R_0}, \ldots, \frac{r_k}{R_k}]$ where R_i is the maximum rank in that objective. By normalizing the ranks before summation, any criteria which tends to obtain higher or lower values is equalized, creating a vector of unbiased raw fitness scores.

Table 2.1 shows a summary of how Pareto, summed rank, and normalized summed rank, rank the individuals within a population.

2.2 Grammars and Design

There are many factors an architect needs to consider when designing a building. This is due to the fact that every building is designed to accomplish

Table 2.1: Example Fitness Vectors

Fitness Vector	Pareto	Rank Vector	Sum	Rank	Norm. Sum	Rank
(1,9,5,4)	1	(2,1,2,2)	7	1	1.47	1
(2,100,4,8)	1	(3,2,1,3)	9	2	2.03	2
(10,9,9,10)	2	(4,1,4,4)	13	4	2.6	5
(16,100,8,4)	2	(5,2,3,2)	12	3	2.56	4
(16,9,500,0)	1	(5,1,5,1)	12	3	2.37	3
(0,999,999,999)	1	(1,3,6,5)	15	5	3.2	6

specific goals. Many of these goals can revolve around space, function, and form [5].

An important factor in the design of a building could be to maximize the space allocated for the building [5]. Often a plot of land is purchased and one of the tasks presented to an architect is to maximize the building to the size of the lot. This can be due to the fact that the building in design is meant to be a storage facility, where every additional square foot would mean additional profit.

Function refers to constructing the building such that it is able to accomplish its task [5]. For example, if the building in design is meant to function as a large office building where each floor is to be rented as a separate office. The value of an office would be its allocated floorspace represented by square feet. Therefore an office with 5000 square feet is more valuable then an office with 500 square feet. Moreover, the more floors the building has, the more offices can be sold.

Form is an exclusive term which has many different meanings. One particular meaning is that form can refer to the external recognizable appearance of objects. For example, certain objects can be identified as being a chair due to certain physical properties [5]. Certain building designs could benefit from having a complex form. One example could be of an architect designing a building which is meant to represent the wealth of a particular financial corporation: the final design would require a high level of complexity to achieve the correct form. Aspects that could make a design complex are things such as offset levels, spiral and circular designs, and multi-tiered sections. In this research, form is the main criteria with space also of consideration.

Grammars can be used in various ways, and formally introduced by Stiny [34]. They can describe the rules of spoken languages [7], model biological development [32], encode the structure of fractal images [31], buildings [29], household furniture [14], and more. Grammars are a method in which one can encode and model representations of many kinds.

Shape grammars are a generative grammatical re-writing series of rules which contain terminal symbols, non-terminal symbols, a series of production rules containing terminal and non-terminal symbols, as well as start symbol [27]. One definition of a simple shape grammar is the following.

- S, a finite set of shapes
- L, a finite set of symbols or labels
- R, a finite set of shape rules having the form $a \rightarrow \beta$, where a is a labelled shape in the set $(S, L)+$, and β is in the set $(S, L)^*$
- I the initial, nonempty labelled shape.

One popular method of creating interesting forms of architecture and other types of designs, is to create a grammar which describes the model of a two or three dimensional object. This particular type of grammar is called a shape grammar. It works by encoding a series of alterations in which an initial shape undergoes [11, 34]. The rules within the grammar can specify shape altering operations such as shape replacements, translations, rotations, scaling, repeating, adding shapes and moving shapes. Once the grammar is defined, it is executed sequentially and the actions specified within the grammar are performed on the shape. In the case of architectural design, the resulting three dimensional models represent buildings. Shape grammars can be extended into three dimensions by performing shape altering operations on three dimensional objects. When dealing with this level of shape grammars, shape altering operations such as splitting and extruding the model can be used along with standard three dimensional transformation operators such as translate, scale and rotate.

The benefit of using a grammar to encode the growth of a building is due to the fact that individual aspects of the building can be developed and reused multiple times during its construction. For example, a set of rules can be made which describe the structure of a floor within a building. This set

of rules can be called multiple times to create multiple floors of the same or similar design, saving the designer from having to create each floor explicitly.

One problem with using grammars for design is that they can be difficult and time-consuming to learn, create and modify. A minor change to any aspect of the grammar can create major differences in the resulting object. A second problem is that grammars have the ability to produce noteworthy results when detailed and complex. This is due to the fact that the language available to a grammar, as well as the multitude of production rules, can all be intertwined in a substantial amount of ways.

Some previous work exploring the use of shape grammars for design are as follows. A visually guided shape grammar system was designed by Tapia to let the user specify the grammar, while the system displays the results, giving the user an interface to explore the designs from the specified language [37]. The designer specifies an initial shape on a two dimensional grid and at least one production rule prior to getting, exploring, and refining the grammar. A system by Stiny creates objects from blocks based on spatial relationships, using hand-coded shape grammars to extrude, split, add blocks, remove blocks, and other basic commands [35]. In another study, Stiny uses shape grammars to create canvas paintings through the use of commands such as location, rotation and scale. This is done based on an informal principle that a painting can be explained by consisting of two dimensional shapes [36]. O'Neill used L-Systems, a specialized form of shape grammars, to evolve logo designs [26].

2.3 CityEngine

Buildings of high visual quality and geometric detail were created using shape grammars for the procedural modeling of CG architecture. The system, CityEngine, uses context-sensitive shape rules which allows the user to specify interactions between the different entities of the shapes present within the structure. Cities such as Pompeii have been virtually recreated using this program and method [25]. An example is shown in Figure 2.3.

CityEngine is a program which generates models of cities. It is capable of creating a city from the ground up through the use of grammars to create a roadwork [29], and shape grammars to model buildings [25], resulting in the creation of detailed three dimensional models. CityEngine is a well-developed and tested product. It provides a reliable method of creating

Figure 2.3: A complete cityscape created entirely from within CityEngine. The cityscape is made to resemble the ancient city of Pompeii.

and exporting detailed three dimensional building models. It differs from conventional modeling software due to the fact that it does not have manual model editing tools that would be found in other modeling programs such as 3D Studio Max [2] or Blender [10]. Its focus is on creating building architecture specifically through its built-in shape grammar system.

CityEngine's shape grammar has built-in automatic error correction, graphics rendering, model exporting, and texture importing [16]. CityEngine is capable of exporting the resulting three dimensional model created from a shape grammar in various formats such as Collada [1] (XML format), Autodesk, Renderman, and others. In addition to exporting the model, CityEngine has a custom reporting tool which exports valuable information about the building such as floor space, land-use, floor height, and more. An additional aspect which makes CityEngine powerful is its python scripting interface, which allows users to create their own custom tools and helper methods.

An example three dimensional shape grammar is as follows.

Rules: $Start$, $RuleA$

Functions: $Extrude(n)$, $Rotate(x, y, z)$,

$Split(axis)\{n : command\}[*optional\,repeat]$

Axiom: $Start \rightarrow Extrude(10)\ Split(z)\{5 : RuleA\}*$

P1: $RuleA \rightarrow Rotate(45, 0, 0)$

The example works by first extruding the initial shape as seen in Figure 2.4a (in this case the initial shape is the size of the lot), then splitting the model along it's z-axis into objects 5 units wide. On each of those sub-objects, the second rule is applied which rotates each sub-object 45 degrees along the objects x-axis. The final result is seen in Figure 2.4b.

Like other modeling programs, the disadvantage of using CityEngine's detailed shape grammar language and system is that the modeling process can be time-consuming [16]. In order to create a grammar, it would have to be coded and executed many times in a trail-and-error fashion, to see the progress of the building. Moreover, as mentioned above, any little change in the grammar has the possibility to create wild deviations in the final product. In addition it can be difficult for a user to imagine what the grammar would need to be in the first place. If the user has a building in mind which they would like to create via a shape grammar, they would need to dissect

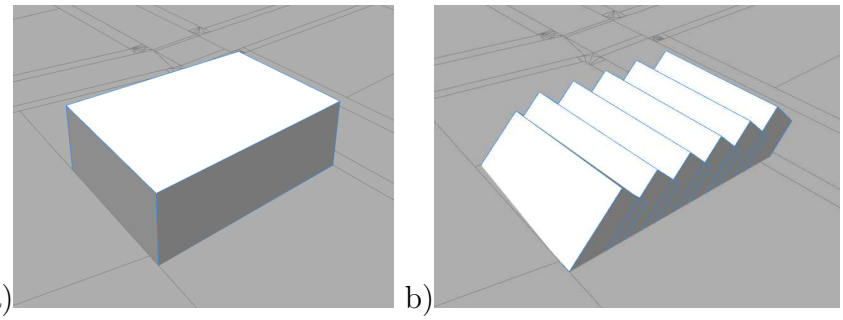


Figure 2.4: Image (a) is the model after the initial extrude. Image (b) is the completed model after the split and rotate commands.

the image into many small parts in an attempt to reverse-engineer the structure, and consider which combination of the vast amount of shape grammar functions would be needed create the model.

Shape grammars are a tested method which one can use to model and create various designs. Since developing detailed grammars to accomplish a specific goal can be time-consuming and tedious, many developers have used genetic programs to help them create their desired grammars. Due to the nature of artificial intelligence, many different areas of the vast solution space are explored in ways that a human might not have initially thought of, returning results which can be intriguing and unique.

2.4 Shape Grammars and Evolutionary Design

There are two main forms of evolutionary design. One form are interactive design systems which require the user to guide the system along its evolutionary path. The user can make personal judgment calls and assign fitness scores to individuals within the population. As such, the amount of individuals in which the system can generate is often limited to prevent the user from being overloaded with work. Also, the number of generations that the evolution evolves is also often reduced. The second form of evolutionary design systems are fully automated ones. These systems require no user intervention during the evolution stages. The user must correctly identify the goals to the evolution and rely on the evolution to return appropriate results.

One advantage of fully automated systems is that the evolution can benefit from having more individuals in the population, as well as having more generations.

Previous work involving an evolutionary approach to generating grammars are as follows.

Machado et al designed a graph-based evolutionary system to evolve grammars [23]. The system uses crossover and mutation operators to automatically evolve context-free grammars which design 2D artwork.

Gero et al make use of shape grammars with an automated genetic algorithm [11]. The system learns grammars which produce topologies of a beam section. Two areas of fitness are maximizing the moment of inertia of the beam section and minimizing the perimeter.

Gero and Sosa created an automated evolutionary system that is used for the design of automotive instrument panels that display situational information which adapt to traffic conditions and driving actions [12]. The fitness function uses a heuristic for "good design" principles that evaluates aspects such as layouts, use of text, and animations as well as evaluates the design in use of normal driving conditions as well as in emergency situations.

Shape grammars are used with grammatical genetic programming for application in automated evolutionary design in two dimensions from O'Neill et al [28]. A target two dimensional shape is given to the evolutionary system in hopes that the system can recreate the shape by creating an appropriate shape grammar consisting of basic functions.

Soddu used evolutionary software and shape grammars to evolve scenarios of possible medieval Italian architectural environments, using human-guided subjective and creative interpretation to guide the system to its final result [33]. The results created full three dimensional models of the buildings to represent a re-creation of the medieval time period.

Jackson evolves 2D L-systems, a specialized form of grammars, through genetic programming with a multi-objective fitness, a co-evolution fitness, and an interactive human-guided approach to creating building architecture [17]. In one example, two human subjects controlled the course of an evolutionary run creating an L-system. Both users were presented with identical starting L-systems containing 10 two dimensional line drawings. They were asked to select two designs which had the most architectural configurations after seeing each member for only five seconds. This was done for 10 generations and two drastically different architectural designs were reached. In other experiments, fitness evaluation based on spatial configurations evolved

two dimensional building architecture.

Evolutionary algorithms in this study by Buelow are used to aid designers of architectural structures [4]. The system was designed to evolve trusses which are structurally optimal for withstanding loads. The system allows the designer to guide the evolution, giving the designer the ability to set preferences to individuals within the population.

O'Neill et al presents a design tool that uses shape grammars and an interactive evolutionary computation to construct three dimensional architectures which represent possible shelters [27]. This tool was implemented as a plug-in for Blender modeling software and has the models evaluated based on user preference, symmetry and weight.

Hornby creates a system which uses L-systems and automated genetic algorithms to evolve three dimensional table designs [14]. The fitness used in this system considers balance, height, surface area, and the amount of material needed to create the design.

Hemberg et al created an interactive design tool known as Genr8, which is based on concepts from artificial life and evolutionary computation inspired by the growth of plants [13]. Genr8 was used to develop six different architectural projects: exploring double-curved self-intersecting surfaces, designing a pneumatic strawberry bar for an event, creating fibrous surfaces, creating surface envelopes which represent inhabitable spaces, and designing an environment though nested cubes.

Jacob et al created a nature-inspired genetic program called Inspirica [18, 19]. Inspirica uses interactive evolutionary breeding to create virtual sculptures and furniture designs though implicit surface modeling. Models such as containers have been developed using storage volume as a fitness, and other models such as chairs have also been created when evolving a base model. Flowers and plants have also been evolved though the same system and the use of L-systems.

Chapter 3

System Details

3.1 Architecture

The genetic programming system that met all the requirements for this research is RobGP. RobGP is an object-based genetic programming system made in C++ [8]. It can adeptly handle automatically defined functions (ADFs) as well as multi-object fitness evaluations, while maintaining a highly-customizable system.

Figure 3.1 shows an overview of the system architecture created using RobGP. The system uses GP to first create an initial population of complete grammars in the format CityEngine requires. As the GP creates each individual, it saves the grammar to the hard disk in the workspace folder which contains the current CityEngine project. Once the file is created and saved, the GP sends a command to CityEngine through a shared UDP port, notify-

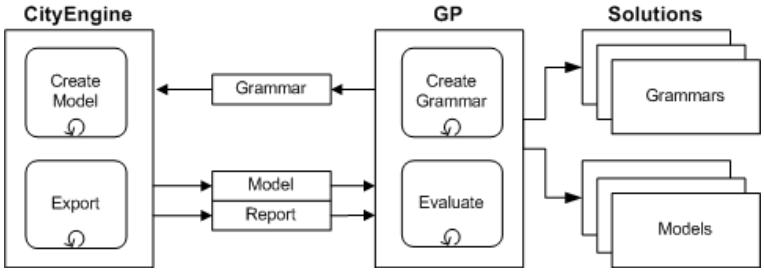


Figure 3.1: Program flow and architecture between the GP system and CityEngine.

ing CityEngine that the grammar is ready for use. The communication and UDP port connection is accomplished via a python script running within CityEngine. After the ready command is sent to CityEngine, CityEngine imports the grammar file into the current running project.

With the grammar now in CityEngine, it is executed to create the resulting building. Then the building model is exported back into the directory as a Collada model file [1]. Once the model file has been exported, CityEngine sends a command back to the GP through the UDP port, notifying the GP that the model is ready for evaluation. At this point, the GP reads in the model file, which is formatted in XML, and evaluates the structure of the model based on the provided fitness criteria. After that individual has been evaluated, the GP moves onto the next individual, repeating the same process.

3.2 Grammar

A genetic program lends itself perfectly to the creation of shape grammars since a shape grammar requires a language, an axiom, and a series of production rules. In this system, the production rules needed in the grammar are represented within the GP tree as ADFs, and the language is provided to the GP as a set of possible commands and terminals. Desired dimensions can be hard-corded to initialize specific values via assignment statements within the grammar. For example, if the building in design is meant to be a skyscraper, defining a high initial height can provide better results as as well reduce the search space. Since the grammar execution is handled entirely by CityEngine, the formatting of the language and assignment variables as well as their respective inputs must match the required structure set by CityEngine.

During the grammar creation, it is possible for the GP to create erroneous grammars or a series of commands which essentially do not accomplish anything. For example, the GP can place a series of size commands in a single rule in a sequence, in this case each new size command overrides the previous one. Another example is that an object can be created that is 100 units long, and following that a split command can state to divide that object into pieces 30 units long. The problem is that the object cannot be evenly divided into units of 30. CityEngine has built-in error checking and correction made to handle these common problems. In this case, the 100 unit long

object will be divided up into three sub-objects 30 units long, and one sub-object 10 units long. CityEngine handles these and other errors gracefully without crashing or stopping the model file from being created. Furthermore, GP can generate a grammar, filling many different production rules with detailed commands, and then not reference any of those production rules in the grammar. For example, the GP can create a production such as $R \rightarrow extrude(50)\, split(x)\{5 : rotate(0, 45, 0)\}$ though never have another production rule call it.

Although it is possible to program the GP to prevent these and other types of errors from happening, it would require a significant amount of post-computation, where the grammar file would need to be checked for a wide multitude of possible errors and bloat, then recreated and rechecked until the grammar file is finally error free. This would greatly increase the run time of the program. Instead, the GP leaves the error handling to CityEngine.

The functions and commands are shown and explained below. The language primitives shown are a subset of CityEngine's shape grammar [16].

$extrude(height)$**:** Extrudes the shape via a given value. Each face polygon of all the meshes in the geometry are taken and extruded along the face normal. This command accepts any integer value.

$split[axis]\{size\ :\ operation(s)\}[*, optional\, repeat]$**:** The split command works by splitting an object along a given axis and creating sub-objects. Each time a size value is specified, one or more operations must also be specified. Once the split command divides an object, the specified operations run on that newly created sub-object. The operations can be any command available to the language, including production rule references and nesting split commands. An optional asterisk can be added at the end of the split command. This informs CityEngine to repeat the entire split command until no additional sub-objects can be created. This thesis allows the split command to occur on the specified axis. The size terminal can be any integer value.

$r(x, y, z)$**:** This rotate operation rotates the current shape within the constraints $x = z = 0;\ 0 \geq y \leq 360$ such that (x, y, z) are integer values and represents degrees along the specified axises. This research only allows rotations along the y-axis.

$r(x, y * split.index/split.total, z)$**:** This rotate operation differs from the standard rotation such that it splits the target object up along the y-axis

when coupled with the split command. When this rotate command is used inside of the split command, the model is split into slices, and each slice is rotated along a central pivot point, creating something similar to a winding staircase.

$r(scopeCenter, x, y * split.index/split.total, z)$**:** This rotate is similar to the previous rotation operator such that when coupled with a split command, the object is divided along the y-axis where each slice is rotated along the central point of the object.

$s('x,' y,' z)$**:** The size operation alters the size of the object relative to the given value. The values given to the operation are floats (x, y, z) within the constraints: $0 \geq x, y, z \leq 2$. For example, $s('0.5,' 1.5,' 0)$ would decrease the size in the x-axis by half, and increase the size in the y-axis by 1.5 times.

$i(object)$**:** Reads in a geometry asset (3D model, polygon mesh) from a file and inserts it into the scope of the current shape and given a bounding box. An inserted file can be split and extruded along the model's faces as well as have all other operations executed on it. The input used in this study provide the grammar with a model of a low-polygonal sphere.

[**and**]**:** The "[" operation pushes the current shape onto the top of the shape stack. It is matched by a succeeding "]" operation, which pops the shape on top of the shape stack and deletes the shape.

$baseHeight$: This is a custom defined terminal within the language. This terminal is an integer, encoded into the grammar, that defines the initial starting height of the model. This terminal can also be used within the grammar command which accepts integer values.

3.3 Genetic Programming Parameters

Table 3.1 summarizes the different parameters and their values used within the GP. The genetic program runs for 60 generations after the initial population is created, where each generation has a population size of 300 individuals. The individuals are pitted against each other using a tournament selection

method with a size of three. Each generation allows for approximately one individual, representing the elite of the generation, to be copied over unaltered into the next generation.

Table 3.1: Common GP Parameters

Parameter	Value
Crossover Rate	0.90
Mutation Rate	0.08
Elite Rate	0.02
ADFs	7
Generations	30 or 60
Population Size	300
Tournament	Size of 3
Fitness	Summed rank
Initial Tree Method	Grow
Diversity Penality	20

The GP is forced to make seven automatically defined functions (ADFs) for each individual. An ADF is an evolved portion of reusable code, which provides the GP with components that can be used multiple times in the evolved program tree [22]. This is similar to how production rules are referenced multiple times within a grammar. In order to simulate this, each ADF within the GP represents one production rule within the grammar. This allow each production rule to evolve as its own unit, yet still able to reference other production rules.

Early program runs initially allowed for 15 ADFs, however this proved difficult for the GP to evolve due to the high number of production rules it was forced to work with. Too many ADFs meant that the GP did not make use of them all, by creating many unreferenced production rules, as as well as leaving many of the production rules un-evolved. With fewer then seven ADFs, the GP often bloated one production rule with a very long chain of commands. One concern with extremely long production rules is that they tend to be mainly bloat. The advantage of grammars is the ability to reference multiple production rules several times within the grammar [15],

where just using one series of commands is more of a procedural encoding.

3.4 Fitness Evaluation

After CityEngine has exported the building in the Collada format the genetic program reads it in to assess the output for its fitness evaluation. The fitness evaluations require an inspection of the exported model. In this case, the exported model is formatted in the Collada standard and can easily be read in and evaluated due to the fact that Collada is a well-documented, open source model file, which is formatted in XML [1].

The model file is read in and parsed using the built-in parser native to RobGP. The XML file provides a series of lists: a list of vertices's, a list of polygon normals, and a list of which vertices's and normals belong to which polygon. The GP compiles the lists together and forms each polygon into a separate polygonal object which contains the vertices's of the polygon as well as its surface normal vector. The fitness then reads through and evaluates the polygons depending on the provided criteria, assigning the individual with the appropriate fitness scores.

In order to help keep the populations from grouping up and creating multiple copies of the same building, a diversity penalty is used. When two individuals have an identical score, one of the individuals is given a penalty of 20 points, making it less favorable of a solution. The formula for diversity is as follows.

$$fitness = rank + diversityFactor * identicalIndividuals$$

Chapter 4

Experiments: Basic

4.1 Height Matching to a Targeted Value

4.1.1 Experiment Setup and Parameters

The goal of this experiment is to achieve buildings which reach a target height. Table 4.1 displays the specific parameters used in this experiment. The model began with a unit height of 150 and the target height was 1500 units. The overall goal is to try a simple experiment with a single objective goal. See table 3.1 for other parameters used.

Table 4.1: Parameters - Height Matching to a Targeted Value

Parameter	Value
Targeted Height	1500
Initial Height	150
Generations	30
Tournament	Size of 3
Elites	1

4.1.2 Results

The following table, Table 4.2 summarizes all 10 runs of the experiment with the best result discovered by each. Figure 4.1 shows the performance graph

of the evolution, averaged over all 10 runs.

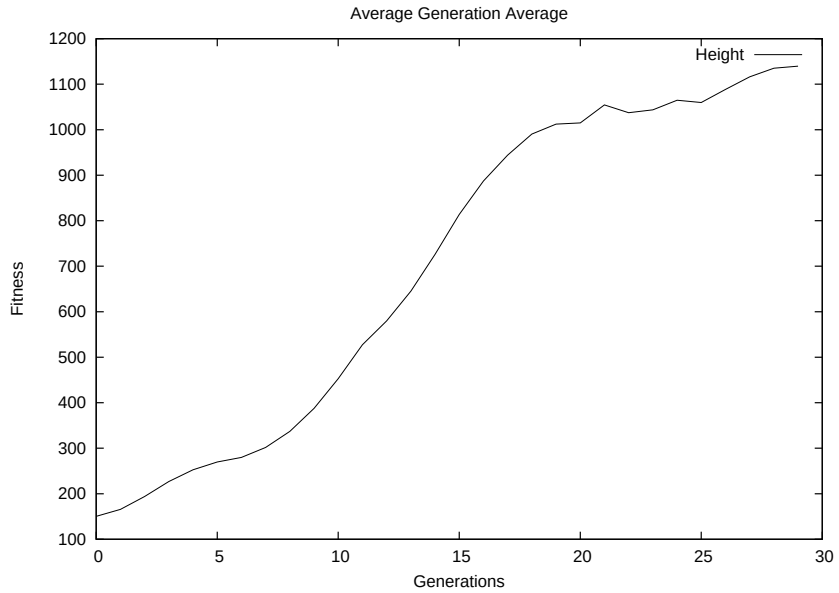
Table 4.2: Results - Height Matching to a Targeted Value

Run No	Final Best Height Distance	Final Pop. Avg. Height Distance
1	1449.0133	1280.48
2	1411.9368	1220.116
3	1500	1325.07
4	1491.12219	1305.476
5	1493.16174	1224.703
6	1466.9489	1257.986
7	295.51	278.14
8	1471.2106	1084.256
9	1484.4126	1306.049
10	1422.6489	1114.046
Average	1348.5965	1139.6322
Target	1500	

The target height experiment successfully grew the buildings to an average height of approximately 1350 units. The trend apparent in the results is that the buildings are generally thin and tall, with low polygon counts. This is due to the fact that the model does not need many polygons to create a taller building, as the grammar accomplishes height by stretching shapes along the y-axis, as seen in Figure 4.3. Table 4.3 shows the grammar that created the model in Figure 4.3.

Figure 4.2 (a) is the building with the highest fitness, a score of 1500. That model perfectly matches the target height, as requested by the fitness. It is interesting to note that the model achieves its height through the use of a rotation set along a fixed pivot point, as opposed to stretching a shape along the y-axis. The model in Figure 4.2 (b) is the building with the lowest fitness score, a height value of only 295.51. This model however does not represent the average results, though represents an outlier among the results.

a)



b)

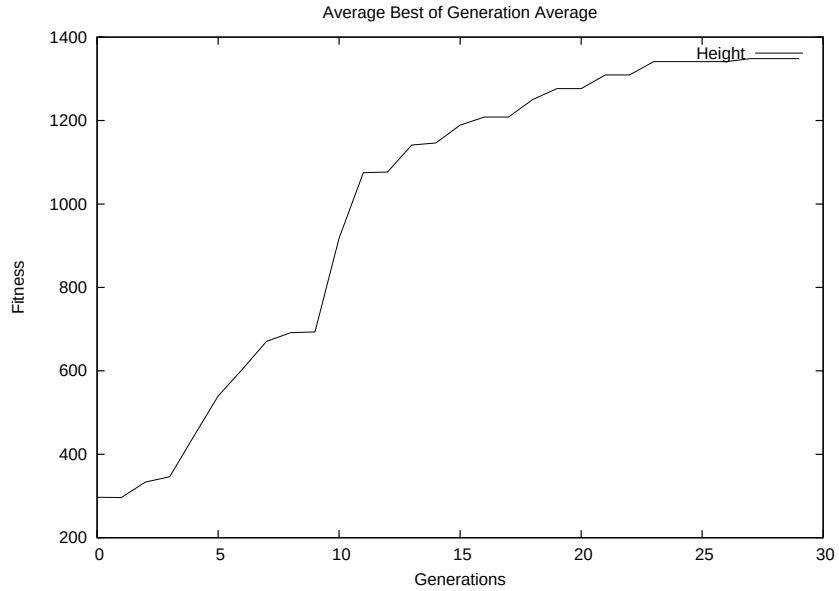
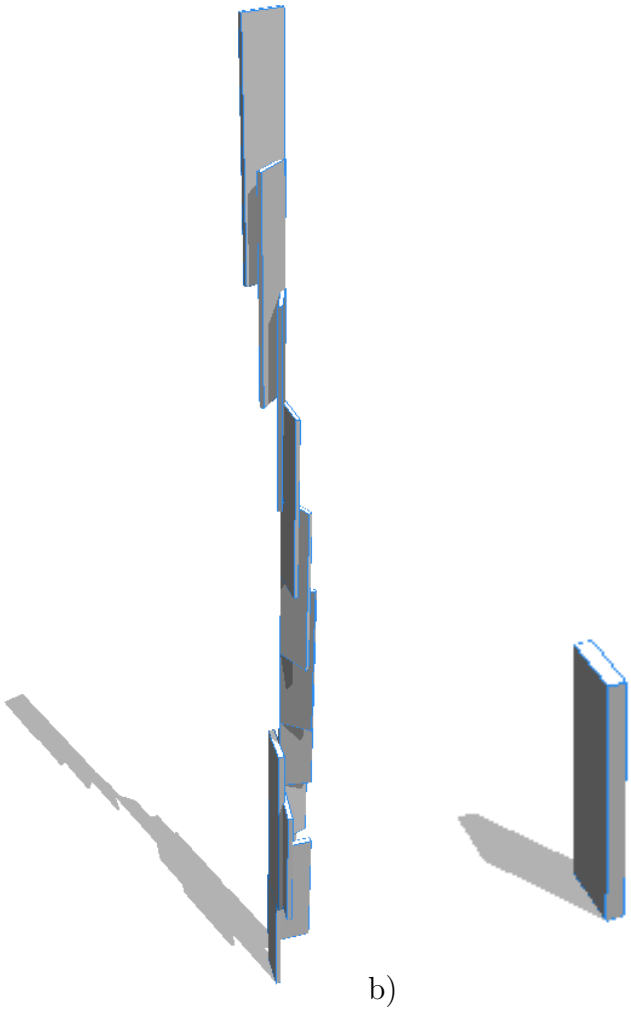
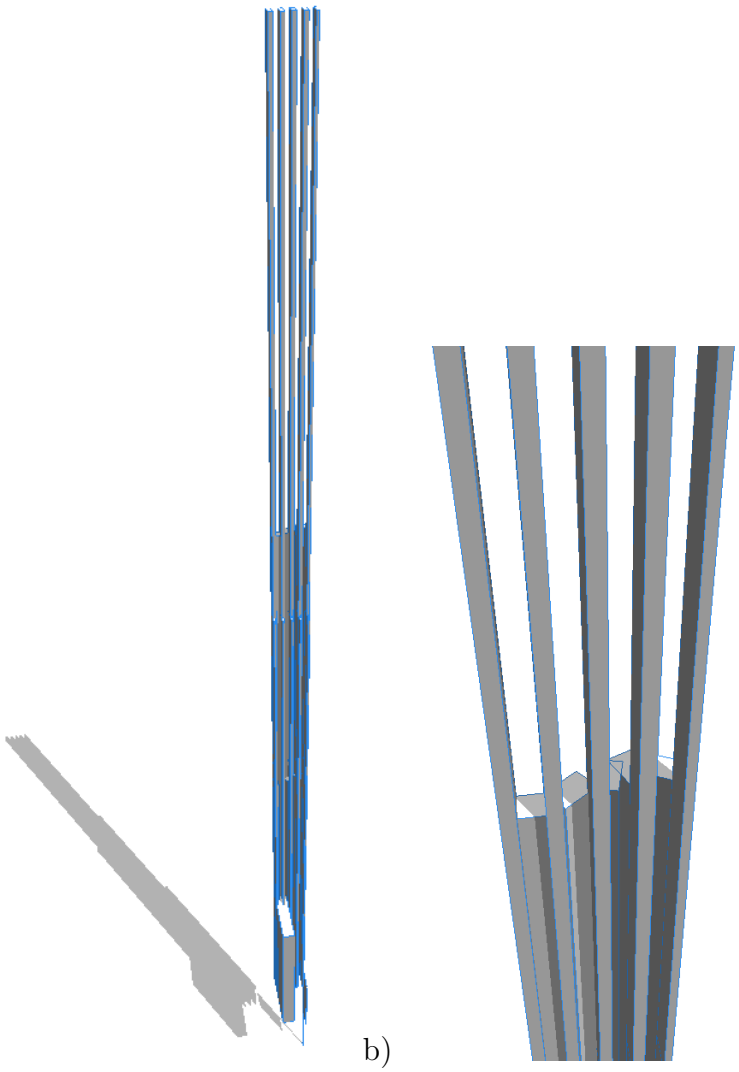


Figure 4.1: Image (a) shows the performance graph of the targeted height experiment. This graph shows the average population generation averaged over all 10 runs. Image (b) shows the average best of generation result though out the experiment.



a) b)

Figure 4.2: The model in image (a) is from the 3rd run of the height experiment, in which the target model height was 1500 units. This model is the highest ranked building, with a height of exactly 1500. Contrasting the best result, image (b) shows a different model with the worst fitness score, a height of only 295.52.



a) b)

Figure 4.3: This model is from the 6th run of the height experiment and has a height of 1466.9489 and displays interesting patterns. Image (a) is the entire view of the tower, and image (b) is a closeup detailed view.

Table 4.3: Grammar for tower in Figure 4.2

```
### predefined variables
attr baseHeight = 150

### generated grammar
Lot -->
   extrude(baseHeight) [ RuleC RuleE RuleC split(x){
   11 : [ RuleF ] }* ] extrude(55) s('0.352752,
   '1.95444, '0.780092) [ s('0.452398, '1.97907, '0.126987)
   [ RuleC ] RuleC ] RuleC

RuleA --> s('0.356402, '1.60921, '1.60921)

RuleB --> r(0, 190*split.index/split.total, 0)

RuleC --> s('0.00877033, '1.07162, '0.676101)

RuleD --> RuleA

RuleE --> RuleD

RuleF -->
   s('0.744515, '1.8519, '0.542972) s('0.744515, '1.8519,
   '0.542972) RuleC RuleE RuleE RuleB RuleE s('1.8519, '1.8519,
   '0.542972) RuleB RuleE RuleE

### unused rules
RuleG --> RuleE
```

4.2 Maximizing Unique Normals

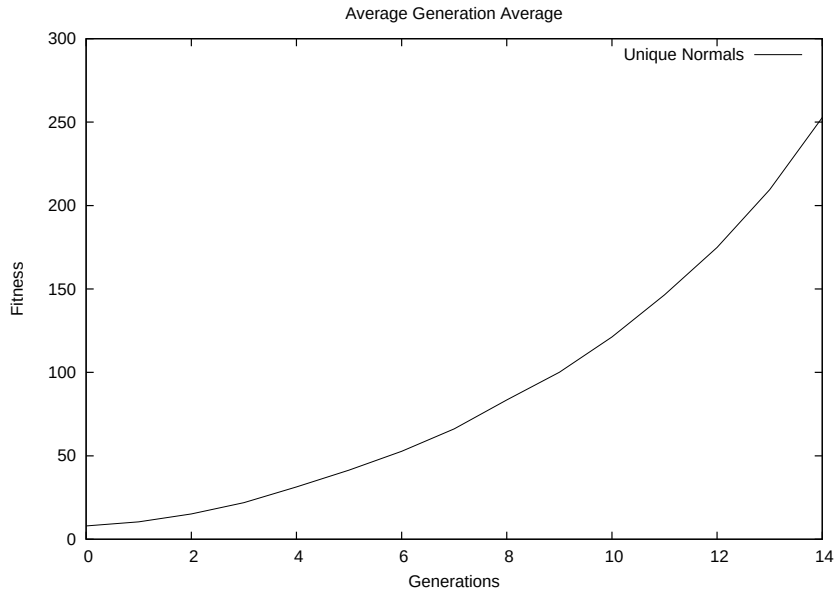
4.2.1 Experiment Setup and Parameters

Every surface, or polygon, in the building has a surface vector, also known as a surface normal. These vectors point out in a direction perpendicular to the polygon which represent the direction that the front face of the polygon is facing. In this experiment, evolution attempts to maximize the number of unique surface normals found in the building model. Table 4.4 shows the parameters used in this experiment.

4.2.2 Results

Table 4.5 shows the best results of all 10 runs, and Figure 4.4 contains the performance graph of the evolution, averaged out over all 10 experimental runs.

a)



b)

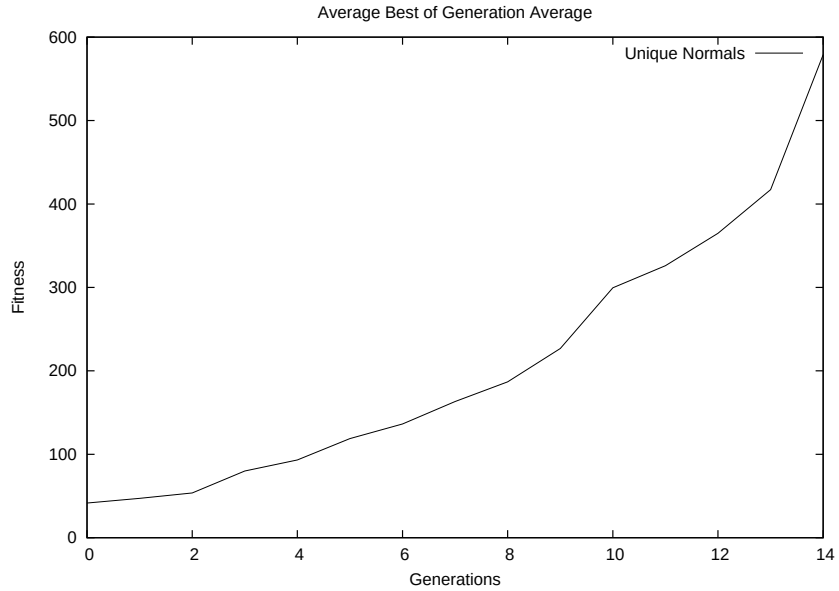
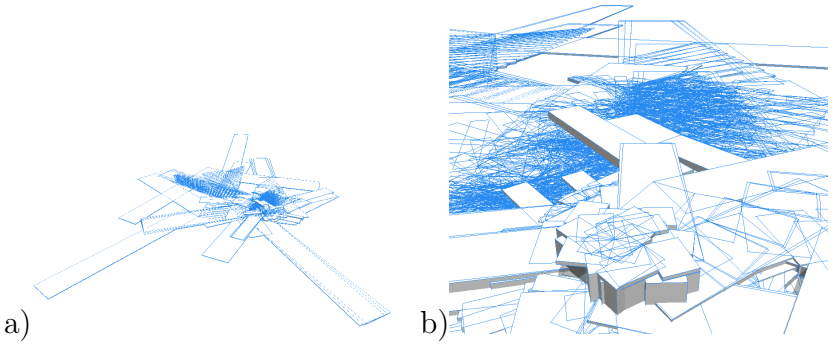
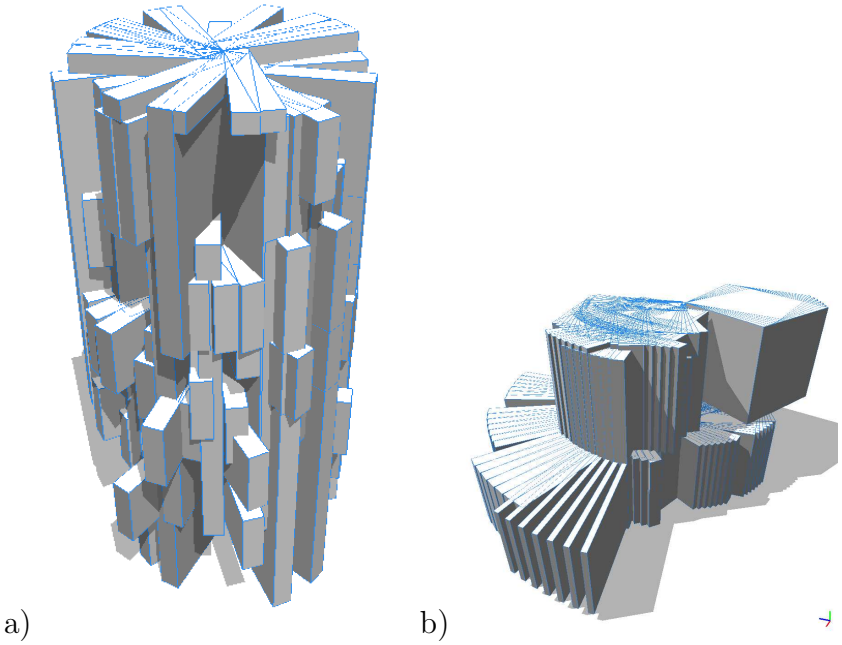


Figure 4.4: Image (a) shows the performance graph of the evolution of the maximizing unique surface normals experiment. This graph displays the average generation averaged over all 10 runs. Image (b) shows the average best of generation result during all 10 runs of the experiment.



a) b)

Figure 4.5: From the maximizing unique normals experiment, this model displays the best fitness, having a unique normal count of 2412. Image (a) is the building in its entirety, and image (b) is a detail view.



a) b)

Figure 4.6: Model (a) has 438 unique normals and model and is from the fourth run (b) has 542 unique normals and is from the fifth run. Both models show interesting aesthetic aspects.

Table 4.4: Parameters - Maximizing Unique Surface Normals

Parameter	Value
Targeted Unique Normals	Maximize
Initial Height	50
Generations	15
Tournament	Size of 3
Elites	1

Table 4.5: Results - Maximizing Unique Normals

Run No	Final Best Unique Normals	Final Pop. Avg. Unique Normals
1	2412	748.127
2	526	294.44
3	322	157.453
4	438	327.24
5	542	194.4
6	302	133.893
7	238	112.64
8	574	328.453
9	258	131.173
10	178	252.7499
Average	579	252.7499

The first thing to note about this experiment is that a minimal number of generations were used. The previous experiment, height matching, allowed for 30 generations, where this experiment only allows for 15 generations. This is due to preliminary experiments that lead to the observation that the higher the unique normals a model has, the greater the models polygon count. Models with a high polygon count create a couple of problems, one problem being increased memory consumption, and the second being greatly increased

Table 4.6: Grammar for tower in Figure 4.6 (b)

```
### predefined variables
attr baseHeight = 50

### generated grammar
Lot -->
   extrude(baseHeight) RuleG r(0, 3, 0) RuleG r(0, 3, 0)
   RuleG r(0, 3, 0) RuleG r(0, 3, 0) RuleG r(0, 3, 0)
   RuleD RuleG r(0, 3, 0) RuleG r(0, 3, 0) RuleB RuleD

RuleA --> r(0, 147*split.index/split.total, 0)

RuleB --> r(scopeCenter, 0, 191*split.index/split.total, 0)

RuleC --> split(y){ 40 : RuleB }*

RuleD --> r(scopeCenter, 0, 330*split.index/split.total, 0)

RuleE -->
   RuleA RuleC split(x){ 15 : [ [ r(scopeCenter, 0,
   104*split.index/split.total, 0) RuleB split(y){ 48 :
   RuleA } extrude(40) split(x){ baseHeight : [
   RuleB RuleB split(y){ baseHeight : RuleB
   s('0.390223, '0.872158, '1.99967) RuleC
   split(y){ baseHeight : [ extrude(24) r(0,
   272*split.index/split.total, 0) ] }* }* ] }*
   RuleB RuleA r(0, 114, 0) r(0, 85*split.index/split.total,
   0) split(x){ baseHeight : RuleA }* RuleA ] [ RuleB ]
   s('0.527875, '0.518108, '0.578314) split(x){ 12 : r(0,
   159*split.index/split.total, 0) }* ] }* r(0,
   132*split.index/split.total, 0)

RuleG --> [ RuleE ]

### unused rules
RuleF --> RuleE
```

rendering times. Therefore, to keep the evolution terminating within a timely fashion, the generations were reduced to 15.

The best result is shown in figure 4.5 and has 2412 unique normals. However, this model spans a massive area as well as contains 13,242 polygons. This model as shown in detail in image (b) of Figure 4.5, displays little ascetic value as it contains many areas of densely tangled overlapping shapes. However unappealing, it has achieved the greatest fitness score.

Taking a look at the two models in Figure 4.6, they both contain a good number of normals yet display two different types of buildings. Image (a) has 438 unique normals and is a taller building where image (b) has 542

unique normals and is a wider building. Moreover, both models display similar circular properties. Table 4.6 shows the grammar which constructs the building in Figure 4.6 (b).

4.3 Maximizing the Normal Distance

4.3.1 Experiment Setup and Parameters

This experiment attempts to maximize the sum of distance between nearest surface normals. The fitness works by comparing each surface normal with every other surface normal in the model to find the two normals which are closest in distance. It then computes the distance between those two normals and adds the distance score to the total. If two selected closest normals are identical, then they are pointing in the same direction, and thus have a distance of 0. The formula to compute this is done by computing the Euclidean distance and is as follows.

$$\text{Given } v_1 = (x_1, y_1, z_1) \text{ and } v_2 = (x_2, y_2, z_2)$$
$$\text{Distance } (v_1, v_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (x_1 - x_2)^2}$$

The fitness is the sum of all nearest distance for all normals. The goal is to maximize this sum. An example of a model with this property is the surface of a sphere. Table 4.7 displays the parameters used in this experiment.

Table 4.7: Parameters - Maximizing the Normal Distance

Parameter	Value
Targeted Normal Distance	Maximize
Initial Height	50
Generations	30
Tournament	Size of 3
Elites	1

4.3.2 Results

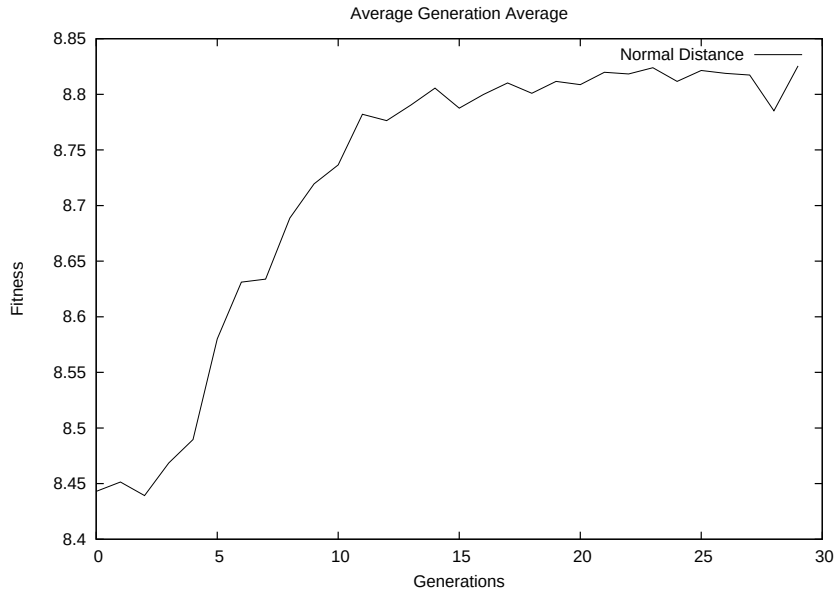
Figure 4.7 displays the average fitness for each generation, over the 10 experimental runs, showing the evolution of the measured criteria. Table 4.8 shows a summary of the top result of each run, as well as the average best of all combined runs. Figure 4.9 shows the building model which demonstrates the highest obtained fitness score over all 10 of the experiment runs, where figure 4.8 shows a model, which may not have the optimal fitness score, yet is particularly interesting to look at due to the curved structure of the model.

Table 4.8: Results - Maximizing the Normal Distance

Run No.	Final Best Normal Distance	Final Pop. avg. Normal Distance
1	8.95127	8.79815
2	8.86564	8.8517
3	8.93419	8.91418
4	8.82203	8.75167
5	8.95128	8.89019
6	8.9051	8.79102
7	8.95132	8.82788
8	8.69248	8.65342
9	8.95128	8.90475
10	8.88675	8.87203
Average	8.8911	8.8254

This experiment was to maximize the distance between surface normals, which would ideally create a curved surface such a sphere, or a spherical column, or several of them throughout the model. However the genetic program found it difficult to evolve as shown in figure 4.9. This might be due to the fact that a cube provides a decent fitness score to this criteria since a cube is essentially a simplified spherical column: in this case a cube is a column with 6 unique high distance normals, where a spherical column would have a high number of unique normals. Additionally, since the grammar begins by creating a 50 unit by 50 unit cube, the genetic program had little drive to evolve a larger structure. This may be due to the fact that any additional

a)



b)

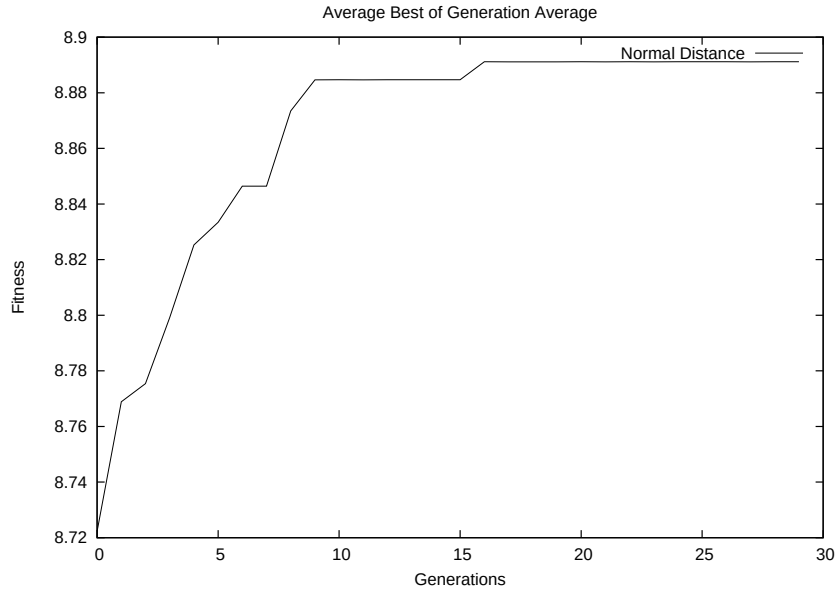


Figure 4.7: Image (a) shows the performance of the average generation, averaged over all 10 runs, from the maximizing normal distance experiment. Image (b) shows the performance graph for the average best result of each generation.

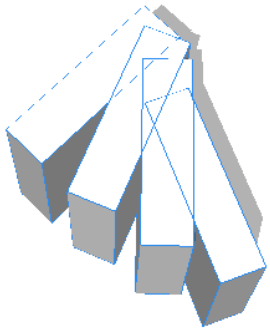


Figure 4.8: This model shows a result which demonstrates how polygons with a good normal distance can begin to take curved shapes. This result is from the second run and has a score of 8.95127.

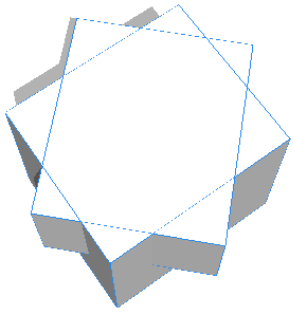


Figure 4.9: This model has the best fitness score over all ten runs, and was found as the highest ranked result in the seventh run. This model has a score of 8.95132.

Table 4.9: Grammar for model in Figure 4.8

```
### predefined variables
attr baseHeight = 50

### generated grammar
Lot -->  extrude(baseHeight) RuleB

RuleA --> r(scopeCenter, 0, 225*split.index/split.total, 0)

RuleB -->
   r(scopeCenter, 0, 52*split.index/split.total, 0)
   [ extrude(baseHeight) s('0.659809, '0.659809, '0.583543)
   [ r(0, 135, 0) RuleA split(y){ 56 : split(y){ 19 :
   RuleA } } ] split(x){ 17 : r(0,
   4*split.index/split.total, 0) } r(scopeCenter, 0,
   52*split.index/split.total, 0) ]

### unused rules
RuleC --> r(0, 90*split.index/split.total, 0)

RuleD -->
   extrude(baseHeight) extrude(51) split(x){ 17 :
   split(x){ 32 : extrude(57) } }* split(x){ baseHeight
   : extrude(51) }* extrude(51) split(x){ 23 :
   split(x){ 23 : r(scopeCenter, 0, 131*split.index/split.total,
   0) } } split(x){ baseHeight : r(0, 104, 0) }*

RuleE --> extrude(35)

RuleF --> RuleE

RuleG --> RuleA
```

structure, unless rotated, would decrease the normal distance fitness score since two normals facing the same direction has a distance of 0. Figure 4.8 shows some aspects of having a high normal distance due to the repeating structure offset by a rotation. The grammar which created this model is found in Table 4.9.

Chapter 5

Experiments: Multi-Objective

5.1 Maximizing Unique Normals Using Spheres while Keeping to a Boundary

5.1.1 Experiment Setup and Parameters

Maximizing the number of unique normals within a building model has the possibility to create very interesting results, as shown in the previous experiments. This experiment expands off of the other experiments by allowing evolution to insert low-polygonal spheres into the model. The criteria for this experiment is to maximize the number of unique normals in the model, while constraining the model to a particular size as measured by the foundation area of the footprint. As shown in the other experiments in Section 4.5, if there is no size constraint, the model can become a confusing massive structure with little aesthetic value. Table 5.1 shows the parameters used within this experiment.

In order to compute the boundary, the function takes the coordinates of the extreme vertices's found on the model, and computes the distance between those points. For example, if the model extends 100 units along the positive x-axis, and 50 units along the negative x-axis, then the models distance along the x-axis is 150 units. If the boundary limit imposed by the fitness is 175 units along the x-axis, then the model is within the allowed boundary by 25 units. The boundary is only computed in the x- and z-axis, and total length that the model surpasses the boundaries are summed together to get the final boundary score of the model. In the case of the

example provided, the boundary score is 0, meaning that the model is entirely inside the boundary.

Table 5.1: Parameters - Maximizing Unique Normals Using Spheres while Keeping to a Boundary

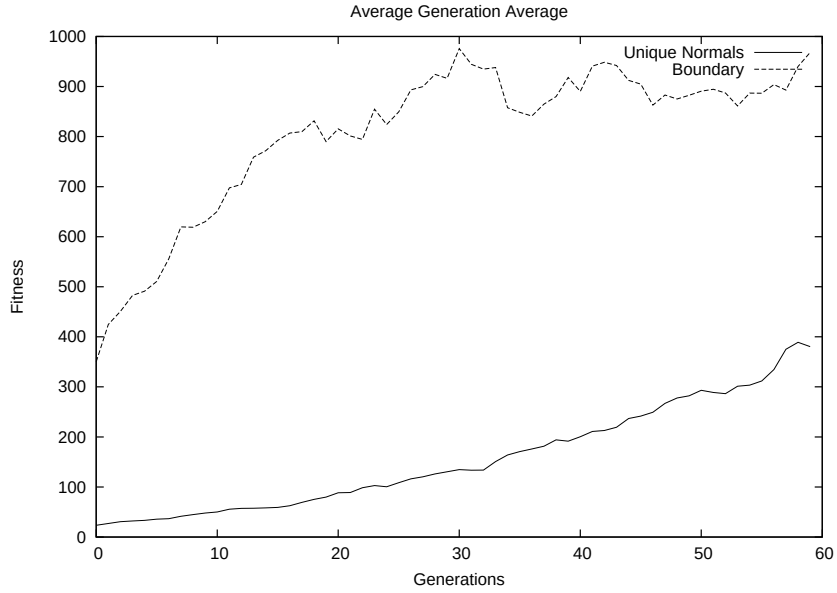
Parameter	Value
Boundary Limit	150
Initial Height	50
Sphere Model	144 polygons
Generations	60
Tournament	Size of 3
Elites	1
Individual Ranking	Summed Rank
Grammar Notes	Allows for the insert(sphere) command

5.1.2 Results

Figure 5.1 displays the average fitness for each generation, over the 10 different experimental runs, showing the evolution of the unique normals and the boundary. The higher the number of unique normals, the better the fitness score, where the higher the boundary number, the more the model is outside of the boundary limits. Table 5.2 shows a summary of the top result of each run, as well as the average. Figure 5.2 shows the building model which demonstrates the highest obtained fitness score over all 10 of the experiment runs, where figures 5.3 and 5.4 show models which demonstrate interesting merit.

By reviewing figure 5.1, it is apparent that in order for the genetic program to evolve structures which display a high count of unique normals, that it needs to expand the overall size of the model. This is shown throughout the generations as when the normal count increases, the amount of the building which remains inside of the boundaries decreases. As compared to the other experiments which only allowed for cubes, the genetic program quickly

a)



b)

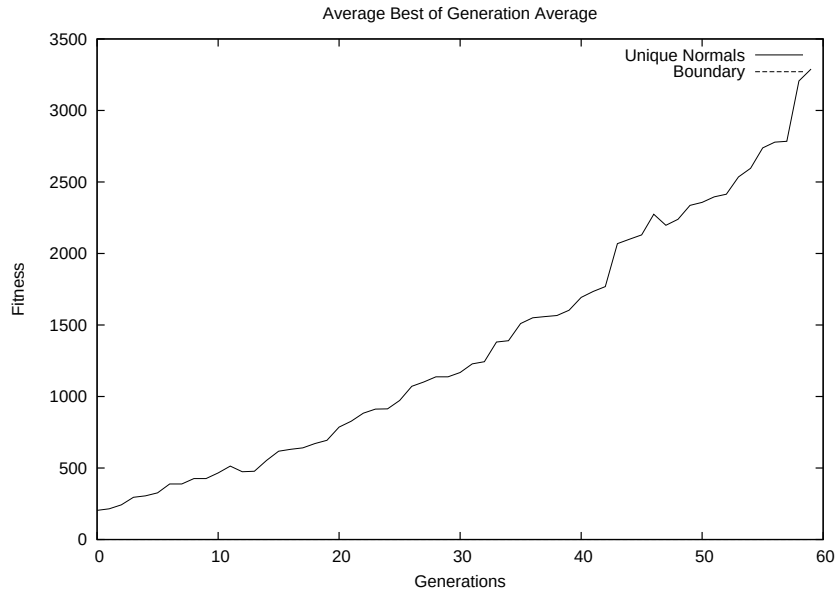
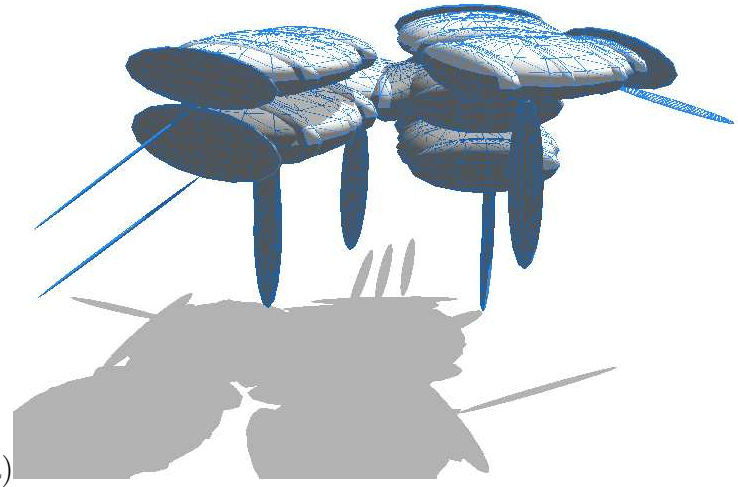


Figure 5.1: Image (a) shows the average fitness score evolution of each generation over the 10 experimental runs for the maximizing unique normals experiment which allows for the inclusion of low-polygonal spheres. Image (b) shows the average best result throughout the generations and all runs of the experiment. Image (b) does not show the boundary curve as it's score is constantly 0 for the best individual.

a)



b)

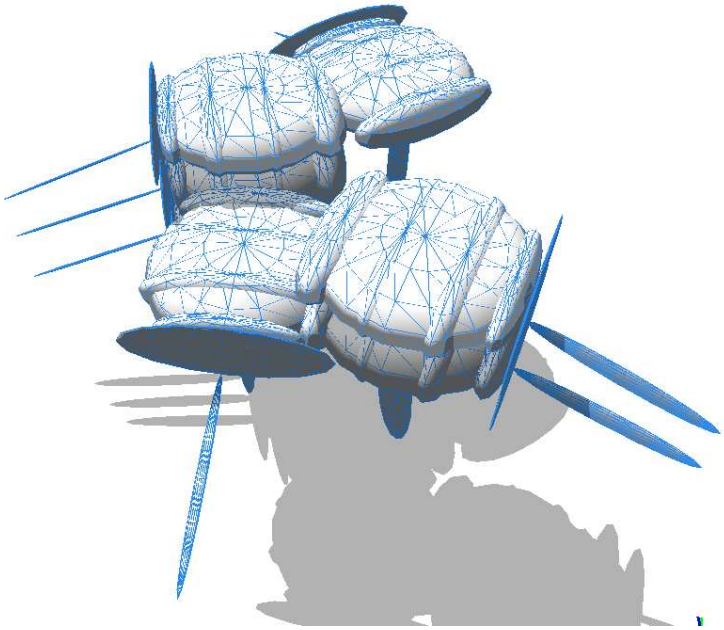
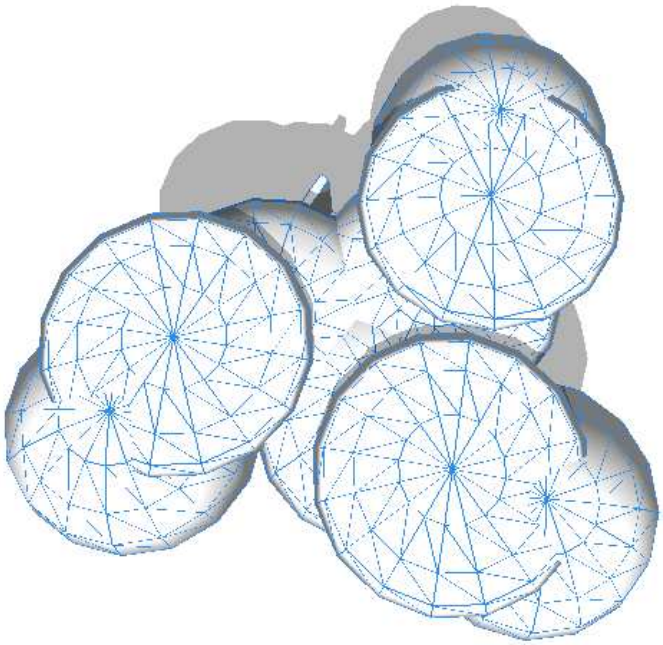
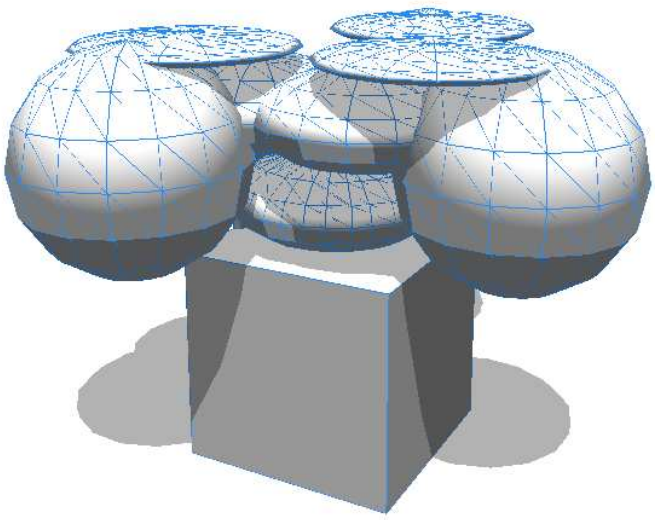


Figure 5.2: The model created from the grammar which returned the best fitness score over all 10 experiment runs. The result shows two different angles of the building which has 6240 unique normals.



a)



b)

Figure 5.3: This model displays interesting symmetry. This model is the best of the second experiment run and has 1330 unique normals. Image (a) and (b) are two different views of the same model.

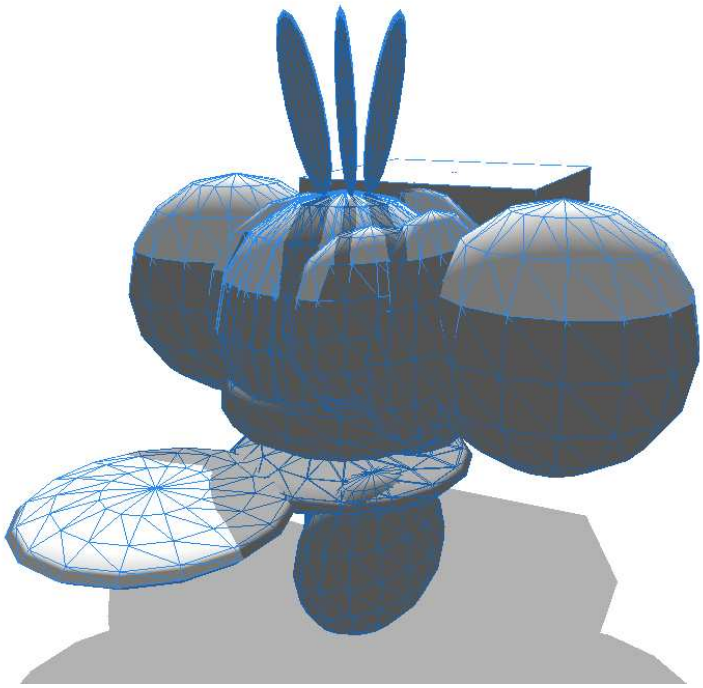


Figure 5.4: A futuristic model which could possible represent a space-colony or habitat. This model is the best of the sixth experiment run and has 4760 unique normals.

Table 5.2: Results - Maximizing Unique Normals Using Spheres while Keeping to a Boundary

	Final Best		Final Pop. Avg.	
Run No.	Normals	Boundary	Normals	Boundary
1	4184	0	380.367	1060.09
2	1330	0	261.693	777.927
3	2688	0	304.49	910.923
4	4485	0	400.023	418.577
5	1046	0	149.753	1208.65
6	4760	0	527.89	516.71
7	6240	0	844.297	975.31
8	3428	0	286.857	1510.82
9	3786	0	450.117	1369.99
10	950	0	199.023	270.5847
Average	3289.7	0	380.4510	270.5847

learns that spheres create the most number of unique normals, and begins to heavily favor them instead of cubes. This appears like an simple choice since a cube only contributes 6 normals, with only 4 of them unique, where a sphere can contribute 144 unique normals.

Figure 5.2 shows the model with the best result out of all 10 runs of the experiment. That model shows an excellent use of spheres as the genetic program evolved spheres with different proportions and angles to create a structure with a very high number of unique normals. Taking a deeper look at the model, each of the four main sections have spheres which have been compressed and layered on top of each other, as well as spheres compressed to such an extent that they form column-like objects, or even elongated spikes. Each one of the four structures have their own set of unique normals, and when the structures are roughly duplicated at a different angles, each new structure contributes their own set of unique normals to the overall fitness score. The different angles of rotation is important, this is due to the fact that if two spheres were beside each other and had an identical size, only one of the spheres would have unique normals, as the second sphere would have normals in the same direction as the first. However, by resizing and

compressing the sphere, it changes the normals along each polygon on the sphere.

Another result as shown in Figure 5.3 deserves a highlighted focus due to the symmetrical nature of the structure. This model shows an excellent example of how encoding a model into a grammar can create aesthetically pleasing results, due to the ability to reuse objects and components. In this case, one object was created then duplicated 3 times along a single pivot point which was perched atop a cube.

A second noteworthy result is displayed in Figure 5.4, where the model has a futuristic appeal to it. This model could represent a city encased within a biosphere, where in each of the three main spheres hundreds of levels could exist, each with its own network of streets and living space. The heavily compressed outlying sphere could be used for many things such as a landing pad, or perhaps a giant park which the futuristic inhabitants can take leisurely strolls in, or perhaps vacation in.

5.2 Maximizing Unique Normals and Height Matching: Random Search versus Evolution

5.2.1 Experiment Setup and Parameters

Since the results of the height matching and maximizing unique normals experiments produced both interesting and accurate results, this experiment was created to make use of both of those fitness criteria. This experiment first used the summed rank method with a tournament selection of 3 with 1 elite, and then was ran a second time using a tournament selection of 1 and 0 elites, creating a random search environment. These two different experiments are then compared. Table 5.3 shows the parameters used for the experiment.

5.2.2 Results

Table 5.4 and Figure 5.5 show the results of the experiment using a tournament size of 3. Table 5.5 and Figure 5.6 show the results with the same fitness criteria ran using a tournament size of 1 and 0 elites to simulate

Table 5.3: Parameters - Maximizing Unique Normals and Height Matching

Parameter	Value
Unique Normals	Maximum
Initial Height	50
Target Height	750
Generations	60
Tournament	Size of 3 and 1
Elites	1
Individual Ranking	Summed Rank

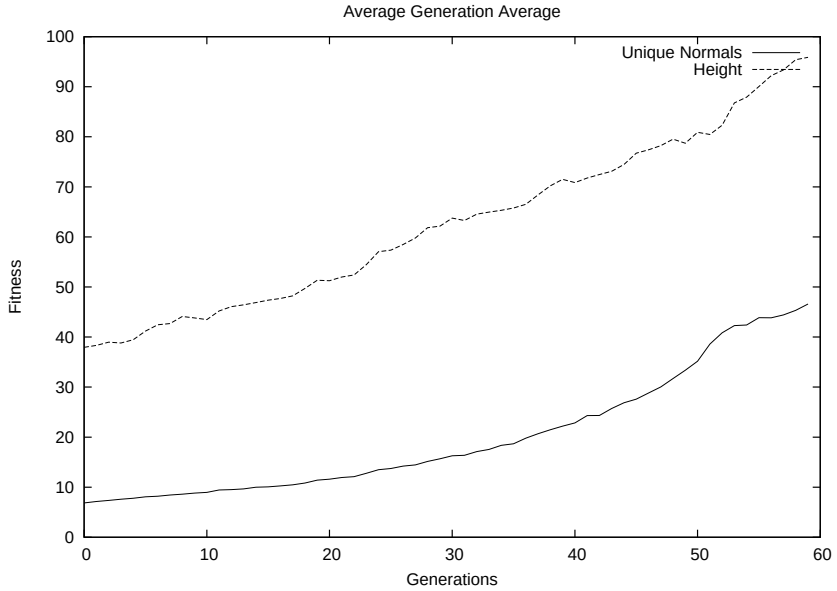
random search.

Table 5.4: Results - Maximizing Unique Normals and Height Matching Using a Tournament Size of 3

	Final Best		Final Pop. Avg.	
Run No	Unique Normals	Height	Unique Normals	Height
1	218	135.893	40.4933	74.747
2	114	152.448	20.24	74.526
3	130	321.987	39.2	70.119
4	150	672.7477	38.0533	113.605
5	470	318.467	76.6133	119.587
6	247	164.464	39.1	87.837
7	510	345.109	67.93	104.782
8	150	99.172	32.0933	58.731
9	558	747.48663	42.6	151.747
10	710	168.071	69.6533	103.025
Average	325.7000	312.5845	46.5976	95.8706

As shown in the above results and tables, evolution trumps random search. The final results of the evolution over all ten runs had an aver-

a)



b)

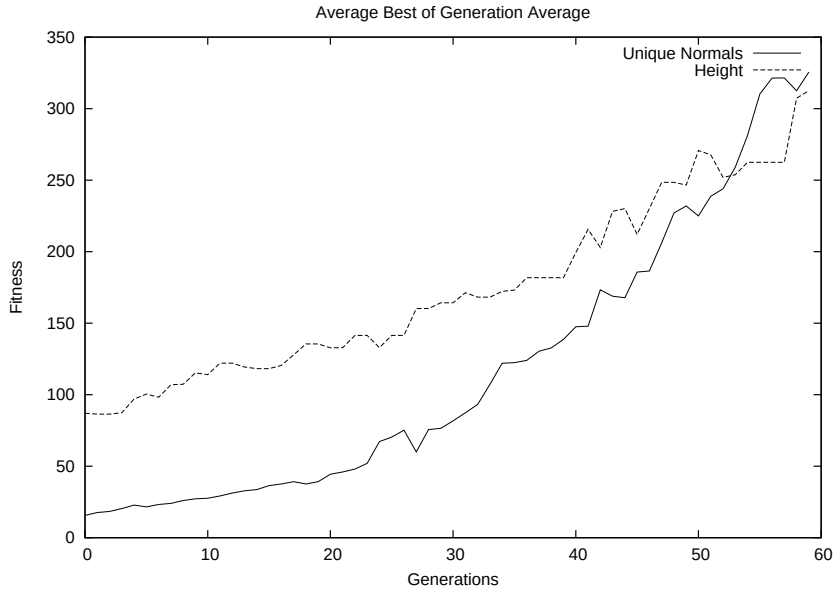
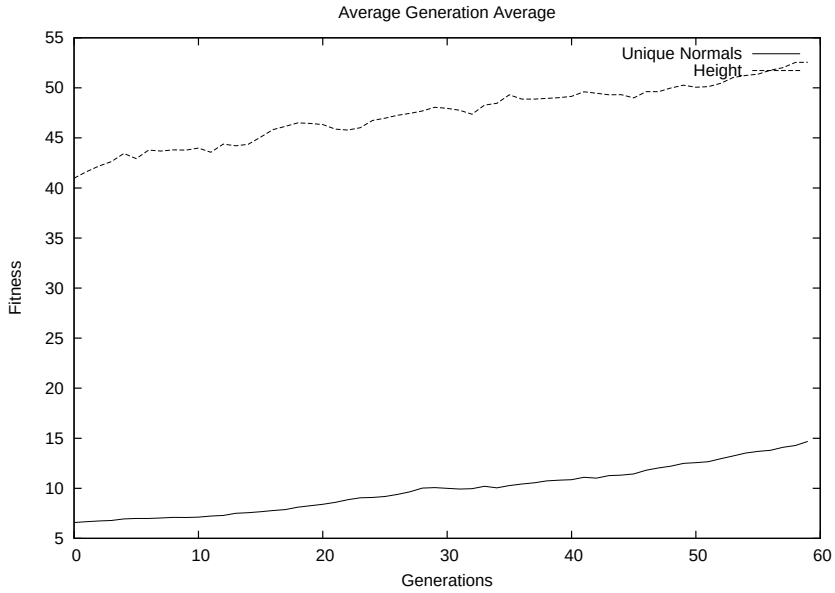


Figure 5.5: Image (a) is the average fitness score evolution of each generation over the 10 experimental runs from the height matching and maximizing normals experiment with a tournament size of 3. Image (b) is the average best of generation.

a)



b)

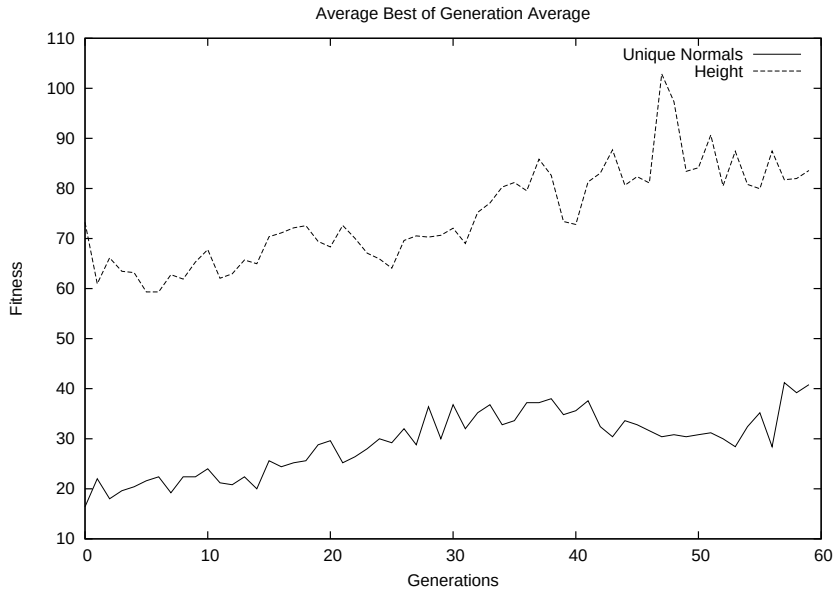


Figure 5.6: Image (a) shows the average fitness score evolution of each generation over the 10 experimental runs from the height matching and maximizing normals experiment with a tournament size of 1 and no elites, simulating random search. Image (b) shows the average best of generation.

Table 5.5: Results - Maximizing Unique Normals and Height Matching Using Random Search

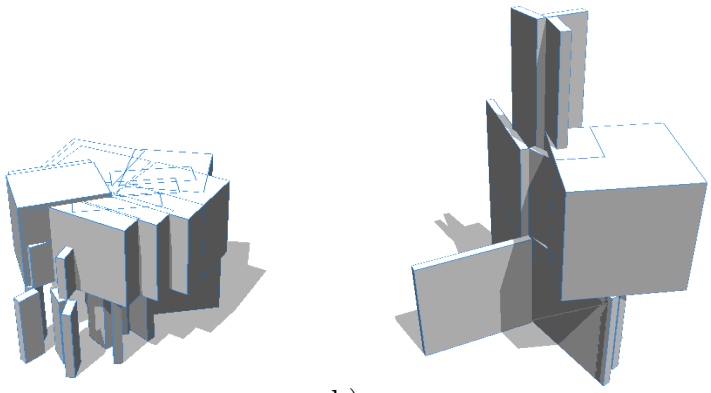
	Final Best		Final Pop. Avg.	
Run No	Unique Normals	Height	Unique Normals	Height
1	106	52.589	29.67	50.972
2	70	50	24.0133	51.179
3	54	98.826	15.5067	50.766
4	22	50	7.82667	45.411
5	26	86.011	13.3067	55.826
6	30	75.061	10.4	46.322
7	46	50	16.4667	47.257
8	10	192.945	7.45	66.201
9	34	95.826	13.76	51.023
10	10	84.648	8.46667	60.543
Average	40.8000	83.5906	14.6866	52.5500

Table 5.6: Results - Summary and T-Test Confidence Percentages

	Final Best Avg.		Final Pop. Avg.	
	Unique Normals	Height	Unique Normals	Height
k=3	325.7	312.5845	46.5976	95.8706
k=1	40.8	83.5906	14.6866	52.55
Conf.	99.97	98.92	99.97	99.92

age unique normal count of 325.7 and an average height of 312.5845. When looking at the results of the same experiment though with no evolution (random search), the final results when averaged over all ten runs achieved a score of 40.8 unique normals, and an average height of 83.5906.

When looking at the random search results, Figure 5.7 (a) shows the model which has the highest count of 106 unique normals, however has a height of only 52.589 units. When comparing that to the model from the



a) b)

Figure 5.7: These two different models display some of the best fitness scores found throughout the ten runs of the maximizing normals, and height matching though a random search experiment. Image (a) has 106 unique normals and a height of 52.589. And image (b) has 34 unique normals and a height of 95.826 units.

evolved results which has the highest count of unique normals, the model in Figure 5.9 has 558 unique normals and an almost perfect height of 747.48663.

An interesting thing to note is that the random search does appear to improve, even if only slightly. One possible reason for this is that as the evolution progresses, the grammars naturally become larger. This is because the fitness is to evaluate the height of the building as well as the number of unique normals. The larger the grammar is, the larger the resulting buildings tend to be. As such, a larger building would naturally have more unique normals and be taller as well.

One misinterpretation of these results might lead to the conclusion that bigger or longer grammars rules create better buildings. However this is not the case. Table 5.7 shows the grammar for the model in Figure 5.7, and Table 5.8 shows the grammar for the model in Figure 5.9. When comparing the two grammars, the grammar generated by the random search makes use of every production rule, as well as has more commands present in each production rule then the grammar generated from the evolved results. The evolved grammar is more compact and generates a significantly better building. One conclusion which can be made from this observation is that for a user to create a good building from a grammar, they cannot randomly chain commands and

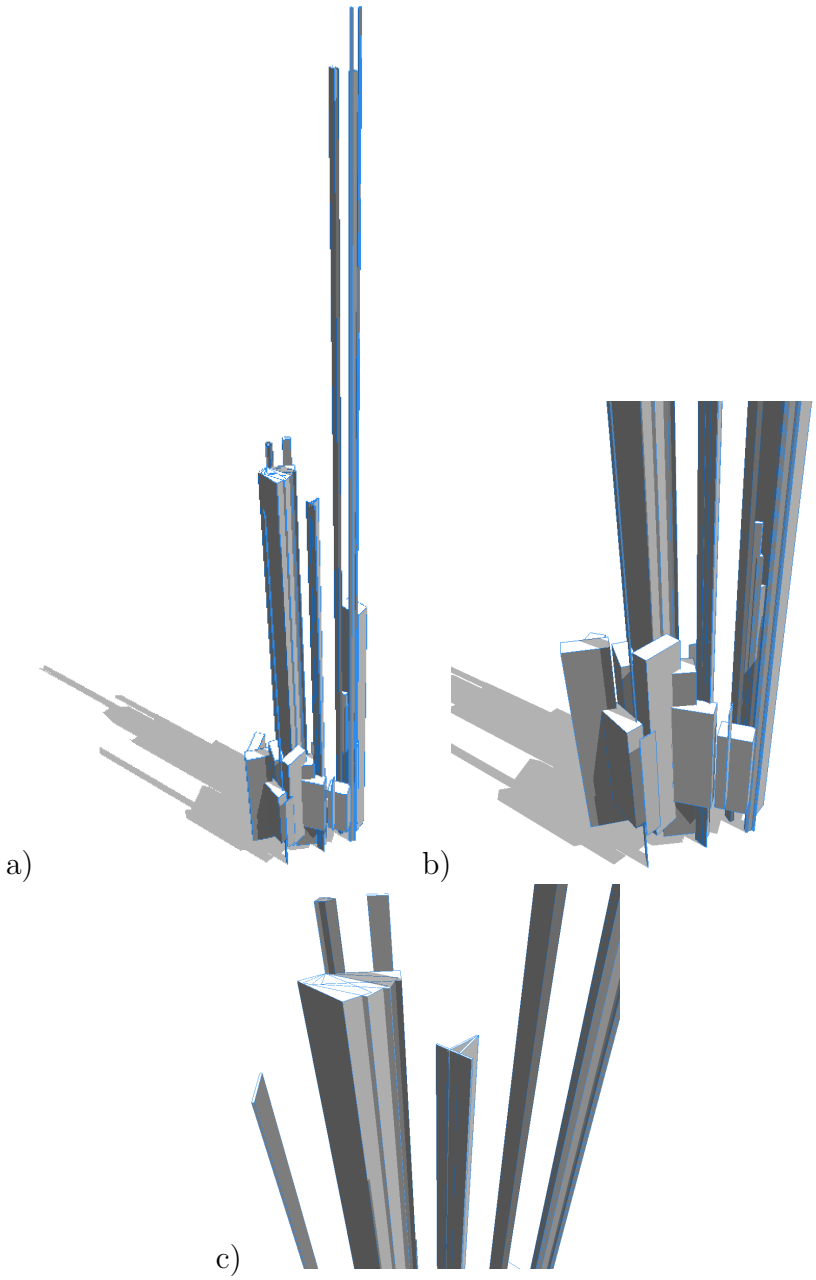


Figure 5.8: These results are all of the same model, taken from the best result of the forth run. This model has 150 unique normals and a height of 672.7477 units. Image (a) is the full view of the model, where images (b) and (c) are detailed views.

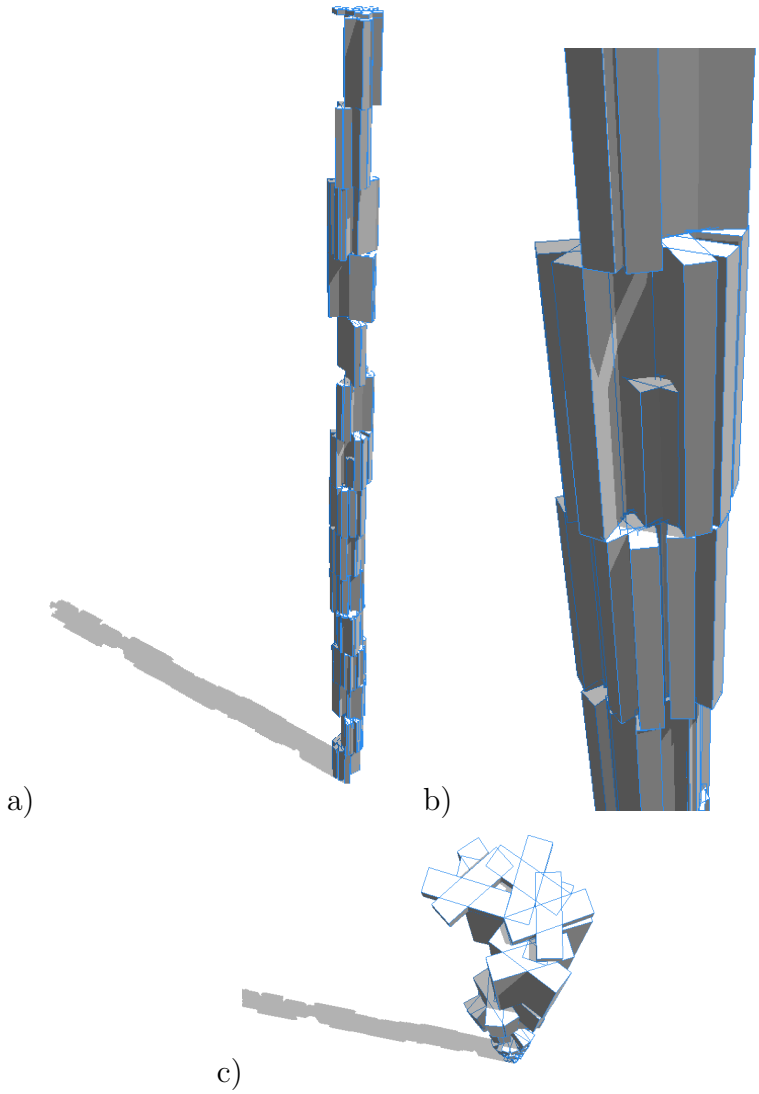


Figure 5.9: These results are all of the same model, taken from the best result of the ninth run, and also displayed the all-around best combined fitness scores for the experiment. This model has 558 unique normals and is 747.48663 units tall. Image (a) is the full view of the model, where images (b) and (c) are detailed views.

Table 5.7: Grammar for tower in Figure 5.7 (a)

```
### predefined variables
attr baseHeight = 50

### generated grammar
Lot -->
   extrude(baseHeight) [ RuleD r(0, 266, 0) RuleF
   [ RuleC RuleF ] RuleG [ RuleD RuleF RuleF
   RuleD RuleG RuleC RuleF r(scopeCenter, 0,
   21*split.index/split.total, 0) RuleD ] r(0,
   17*split.index/split.total, 0) r(scopeCenter, 0,
   21*split.index/split.total, 0) [ RuleE RuleB ] ]

RuleA -->
   split(y){ 54 : [ [ split(y){ 22 : r(0,
   349*split.index/split.total, 0) extrude(baseHeight)
   r(0, 208, 0) split(y){ baseHeight : r(0,
   129*split.index/split.total, 0) split(x){ 54 :
   extrude(baseHeight) }* extrude(baseHeight) split(y){
   58 : extrude(52) }* extrude(baseHeight) r(0,
   137*split.index/split.total, 0) } }* ] ] }

RuleB -->
   r(0, 282*split.index/split.total, 0) split(x){
   48 : RuleA }* r(0, 282*split.index/split.total, 0)
   [ extrude(45) r(0, 168, 0) RuleA r(0, 145, 0)
   r(0, 28*split.index/split.total, 0) RuleA extrude(33) ]

RuleC -->  s('1.04537, '1.05178, '0.821831)

RuleD -->  r(0, 221, 0)

RuleE -->
   split(x){ 18 : split(x){ baseHeight : split(x){
   baseHeight : r(scopeCenter, 0, 355*split.index/split.total,
   0) } } RuleD [ s('0.904042, '0.553391, '0.103944)
   r(0, 66, 0) r(0, 182*split.index/split.total, 0)
   r(0, 279*split.index/split.total, 0) split(x){
   44 : RuleB } ] split(x){ 16 : split(x){ baseHeight :
   s('0.832202, '1.53894, '0.000411292) } } }*

RuleF -->
   [ r(scopeCenter, 0, 183*split.index/split.total, 0)
   r(0, 29, 0) r(0, 216*split.index/split.total, 0) ]

RuleG -->   RuleE
```

production rules, though must know how to properly weave them together.

Table 5.8: Grammar for tower in Figure 5.9

```
### predefined variables
attr baseHeight = 50

### generated grammar
Lot -->
   extrude(baseHeight) s('0.245968, '1.74324, '0.418107)
   split(x){ baseHeight : [ s('0.910534, '1.69943, '0.665899)
   s('1.03508, '1.69943, '0.779976) RuleG split(x){
   42 : split(x){ baseHeight : [ s('0.910534,
   '1.69943, '0.665899) RuleA RuleG [ s('0.910534,
   '1.69943, '0.665899) s('1.69943, '1.03508, '1.03551)
   RuleG split(x){ 42 : r(0, 15, 0) } ] ] } } ] }

RuleA -->  r(scopeCenter, 0, 268*split.index/split.total, 0)

RuleB -->  r(0, 13*split.index/split.total, 0)

RuleC -->
   split(y){ baseHeight : r(0, 304, 0) r(0,
   306*split.index/split.total, 0) r(0, 304, 0) r(0,
   306*split.index/split.total, 0) [ RuleB RuleA RuleB ]
   [ RuleB s('0.585351, '0.960822, '0.479374) RuleB ]
   r(0, 302*split.index/split.total, 0) [ split(x){
   29 : RuleB }* RuleA ] r(0, 306, 0) [ split(x){
   29 : RuleB }* RuleA ] r(scopeCenter, 0,
   131*split.index/split.total, 0) }*

RuleG --> split(x){ 43 : RuleC }*

### unused rules
RuleD --> RuleC

RuleE -->
   r(scopeCenter, 0, 63*split.index/split.total, 0)
   RuleA RuleC RuleB

RuleF -->
   [ RuleB RuleC split(y){ baseHeight : r(0,
   208*split.index/split.total, 0) } r(0,
   346*split.index/split.total, 0) r(scopeCenter, 0,
   346*split.index/split.total, 0) ]
```

5.3 Comparing Summed Rank, Normalized Summed Rank and Pareto Evaluation Methods

5.3.1 Experiment Setup and Parameters

As demonstrated in previous experiments, interesting building models can be made when multiple fitness criteria are combined. In this experiment the

criteria is to maximize the number of unique normals, match the height of the model to 750 units, all while attempting to constrain the model within a particular boundary of 175x175 units. Table 5.9 shows the parameters used for the experiment.

This set of fitness criteria is ran in three different experiments to compare different popular evaluation methods. The three methods used are summed rank, normalized summed rank, and Pareto.

Table 5.9: Parameters - Maximizing Unique Normals, Maximizing Height, and Constraining to a Boundary

Parameter	Value
Boundary Limit	175
Target Height	750
Unique Normals	Maximize
Initial Height	50
Generations	30
Population Size	300
Tournament	Size of 3
Elites	1
Individual Ranking	summed rank, normalized summed rank, and Pareto (varies)

5.3.2 Results

Table 5.11 summaries the three different experiments ran, displaying the average result of each fitness as well as the standard deviation for each fitness.

Taking a look at the scatter plot in Figure 5.10, it is shown that Pareto evaluation creates a population which is very diverse. The positive side to this is that there are many different and highly unique results generated. The negative side is that many of those results are extreme cases and rank terribly in one or more fitness areas.

An interesting set of results to note is shown in Table 5.12, which shows

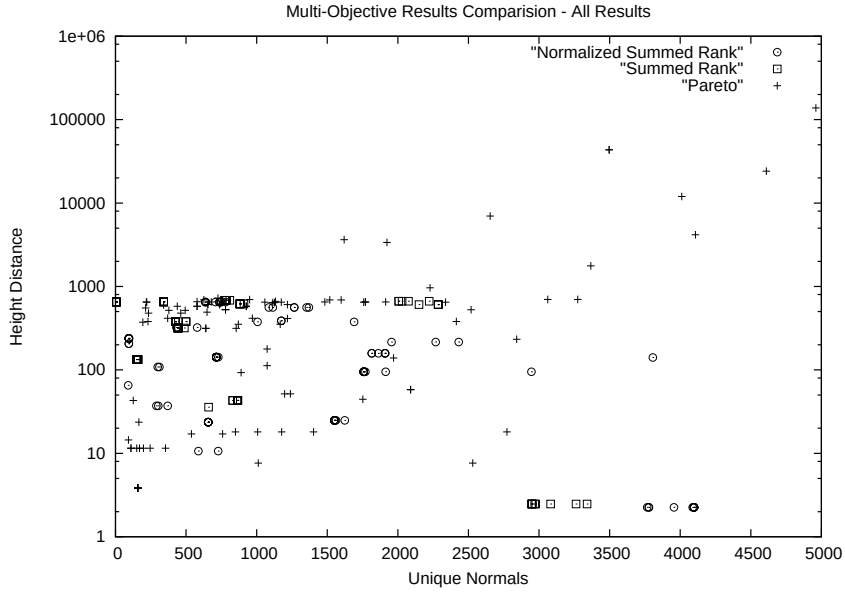
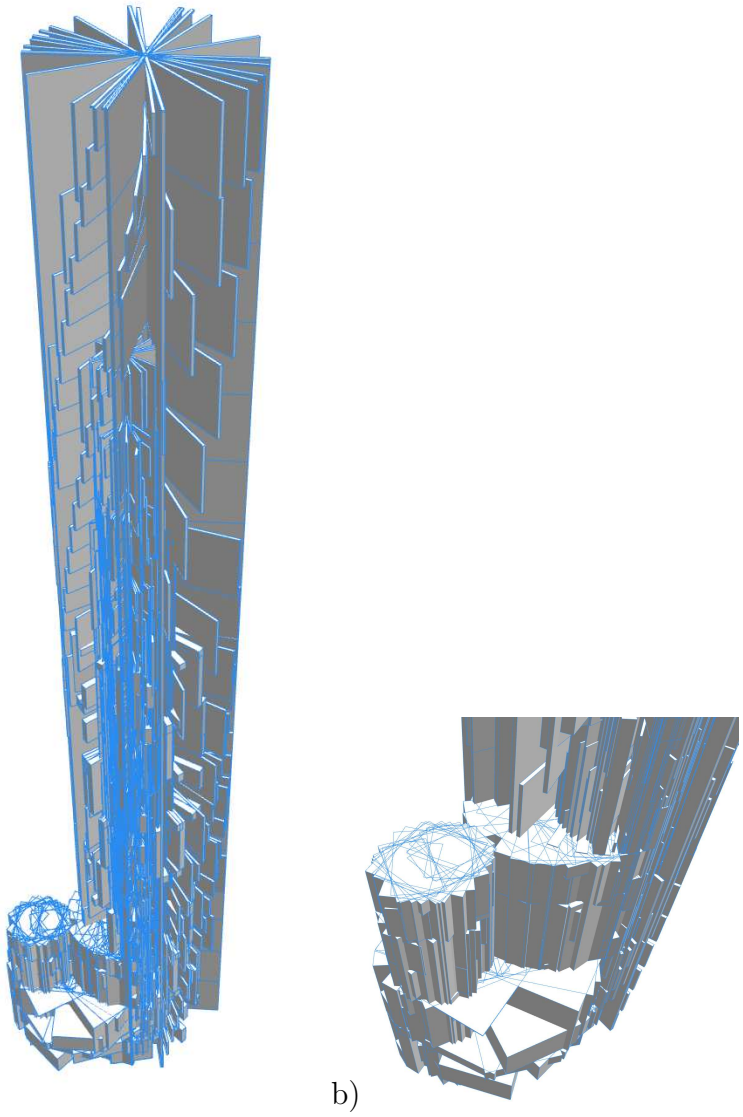
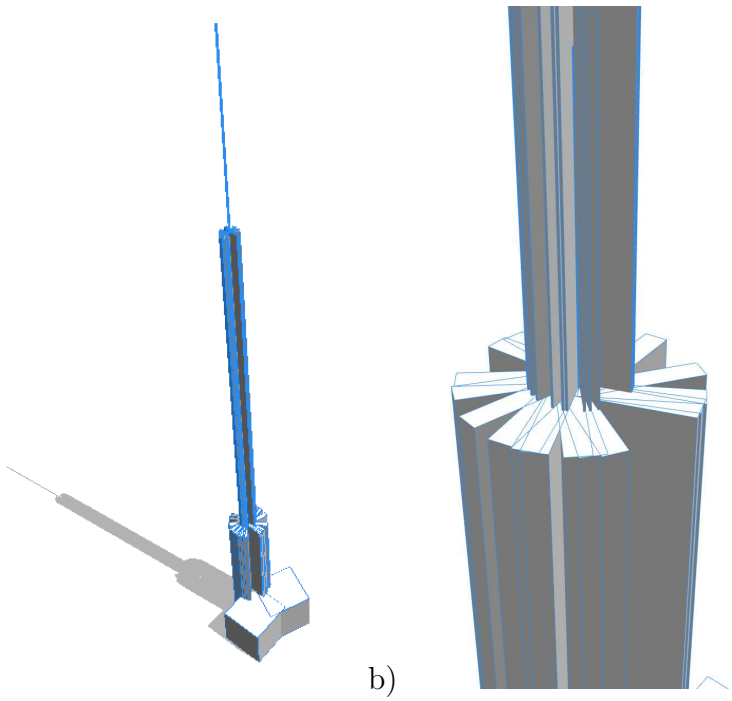


Figure 5.10: This scatter plot shows two fitness scores (height, and count of unique normals) plotted over all 100 results generated by each experiment, where each run of the experiment returns ten results. This plot compares the three different experiments, Normalized Summed Rank, Summed Rank, and Pareto evaluation methods. The target value for the height is 750.



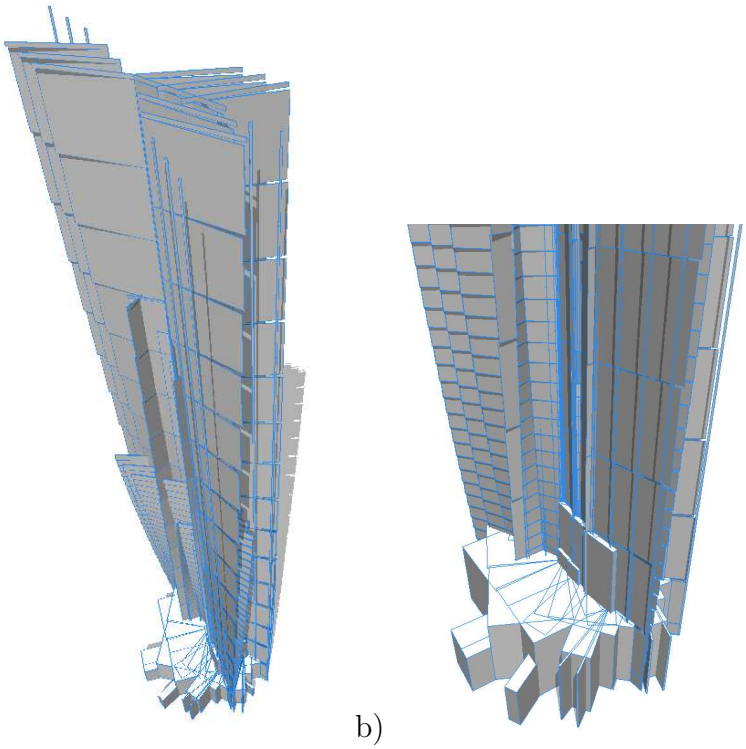
a) b)

Figure 5.11: Images (a) and (b) are both from the best individual produced at the end of the evolution in the ninth run under the normalized summed rank experiment. Image (b) is a closeup of the same model in image (a). This model stays within the boundary, while having 1624 unique normals, and a height of 1475.2029.



a) b)

Figure 5.12: Images (a) and (b) are both from the best individual produced at the end of the evolution in the fifth run under the normalized summed rank experiment. Image (b) is a closeup of the same model in image (a). This model stays within the boundary, while having 4088 unique normals, and a height of 1497.7514.



a) b)

Figure 5.13: Images (a) and (b) are both from the best individual produced at the end of the evolution in the fifth run under the regular summed rank experiment. Image (b) is a closeup of the same model in image (a). This model stays within the boundary, while having 3340 unique normals, and a height of 1497.5269.

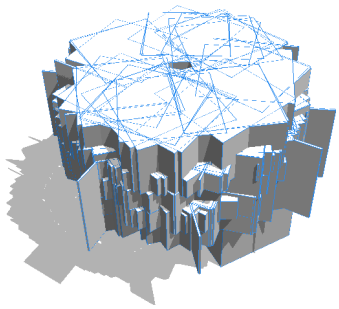
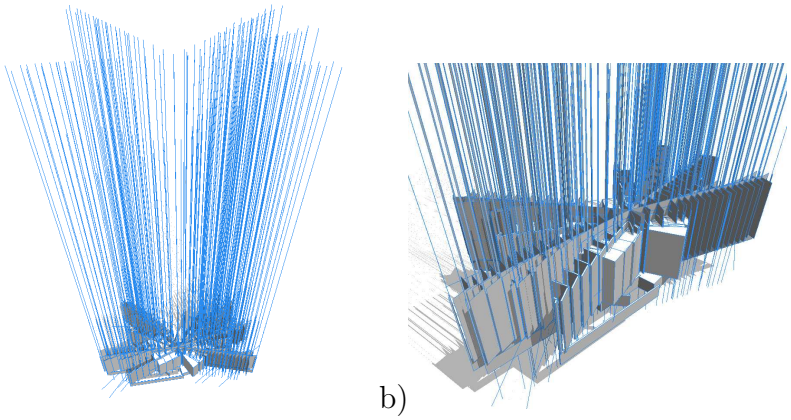
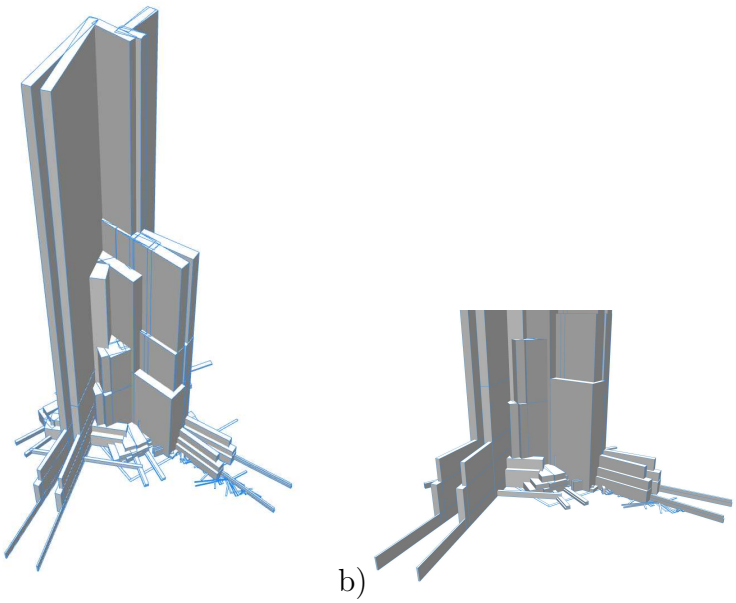


Figure 5.14: This image is the best result from the tenth run of the summed rank experiment. This model had a particular low fitness score when compared to the average. It maintained the limits of the boundary, however it only has 810 unique normals and a height of exactly 817.



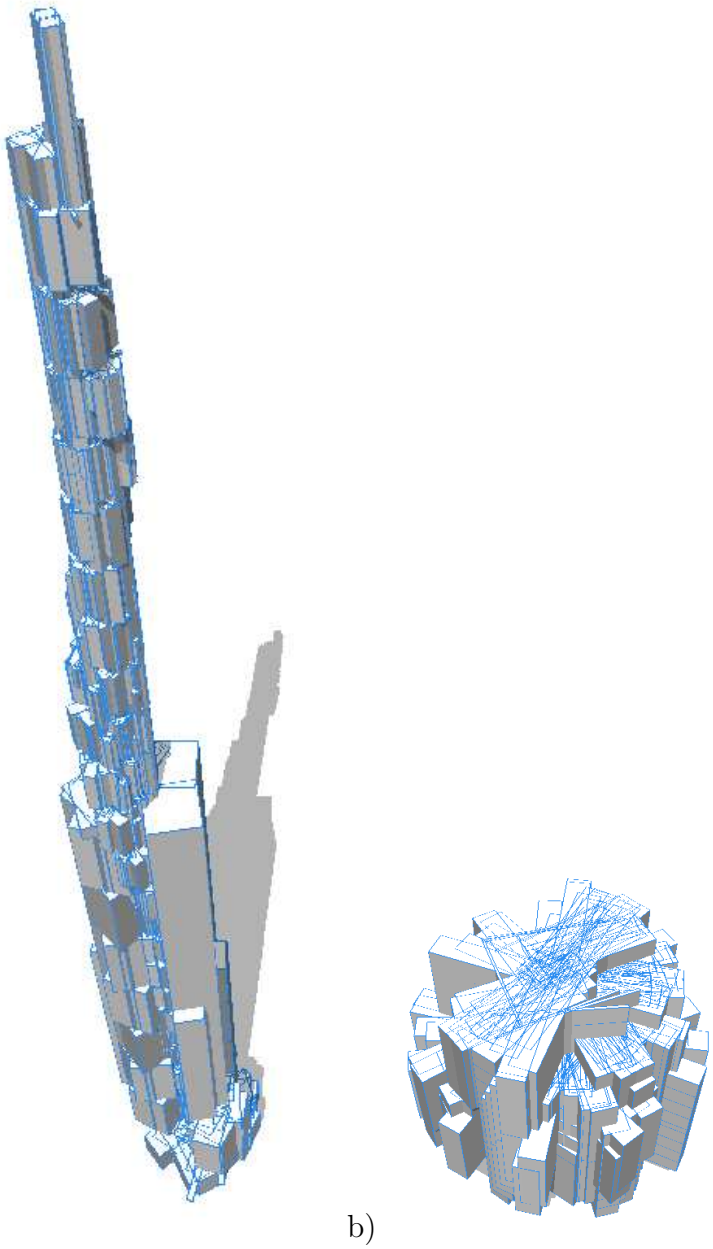
a) b)

Figure 5.15: Images (a) and (b) are both from the best individual produced at the end of the evolution in the second run under the Pareto experiment. Image (b) is a closeup of the same model in image (a). This model breaks the boundary by 545.07 units, has 2531 unique normals, and a height of 1492.3526.



a) b)

Figure 5.16: Images (a) and (b) are both from the best individual produced at the end of the evolution in the seventh run under the Pareto experiment. Image (b) is a closeup of the same model in image (a). This model breaks the boundary by 550.637 units, while having 855 unique normals, and a height of 1184.786.



a) b)

Figure 5.17: Image (a) is from the best result found in run 9 of the experiment using Pareto. It manages to stay within the boundary, have 2090 unique normals, as well as a height of exactly 1442. Image (b) is the outlier of the results from run number 8. That model breaks the boundary limits by 372.859 units, has 1758 normals, and is only 851 units tall.

Table 5.10: Multi-objective Results Comparison: 100 Solutions Each

	Pareto		Summed Rank		Norm. Sum. Rank	
	AVG	STD	AVG	STD	AVG	STD
Boundary	202.2204	219.2640	0.0000	0.0000	0.0000	0.0000
Normals	1137.5000	927.1608	988.1000	1037.2596	1653.3000	1447.9569
Height	368.6579	328.2944	369.1578	295.4449	562.9862	231.2281

Table 5.11: Multi-Objective Results Comparison (Continued)- Duplicates Removed

	Pareto		Summed Rank		Norm. Summed Rank	
	Height	Normals	Height	Normals	Height	Normals
Average	3165.5298	1295.1667	342.7781	1332.6315	254.4790	1439.3396
StDev	15901.3017	1108.5484	272.6226	1087.8604	239.3546	1179.7872
Min	3.8414	92	2.4731	6	2.2486	90
Max	138215	4961	683	3340	650.049	4098
Entries	84 Unique		38 Unique		53 Unique	

Table 5.12: Multi-Objective Results Comparison - Duplicates Removed

	Pareto Boundary	Summed Rank Boundary	Norm. Summed Rank Boundary
Average	316.8260	0	11.2199
StDev	337.7530	0	46.3947
Min	0	0	0
Max	1715.29	0	218.289

the boundary results over all 100 results generated by each of the three evaluation methods, with duplicate results removed. The Pareto evaluation created boundary scores with an average of approximately 316 and a standard deviation of approximately 337, where the normalized summed rank generated averages of about 11 with a standard deviation of about 46. However,

Table 5.13: Multi-Objective T-Test Confidence Percentages

	Boundary	Normals	Height
NSR v SR	91.58	34.27	88.65
P v NSR	100	52.25	90.28
P V SR	100	13.85	89.23

the regular summed rank results generated an average boundary score of 0, creating all results that fit within the provided boundary limitations. This might be due to the fact that Pareto tends to create outliers where regular summed rank tries to please each fitness criteria.

Table 5.11 shows the averages, standard deviation, minimum, and maximum fitness scores for each evaluation method. This table, as well as the scatter plot in Figure 5.10, shows that Pareto has created outliers. In a particular run, the final building height was 138215 units, when the target height is only 1500 units. The regular summed rank and the normalized summed rank experiments generated a maximum height of 650 and 683 respectfully.

The experiment which returned the best results is the normalized summed rank experiment. Figure 5.11 shows an excellent building which has a height of 1475.2 units, a unique normal count of 1624, while staying within the boundary of 175x175 units. The grammar for Figure 5.11 is shown in Table 5.14. Figure 5.12 shows the model which is the best ranked out of the experiment and has a height of 1487.8 unit, 4088 unique normals, and also stays within the boundary. Moreover, this model displays an ascetically pleasing building of a practical skyscraper, complete with a cylindrical tower which tapers in three levels to a point.

In the regular summed rank experiment, the majority of the results fall within acceptable fitness scores. However, a few outliers do exist. Figure 5.13 shows a great model which has a height of 1497.5 units, 3340 unique normals, and stays within the boundary, as well as shows an interesting structure for a possible skyscraper. However the result shown in Figure 5.14 shows an outlier model which has a height of 817 units, only 810 unique normals, though stays within the limits of the boundary. The model is not aesthetically pleasing as it is a short, round building made from a jumbled mess of overlapping objects.

Pareto results returned many good results as well as outliers. In order

Table 5.14: Grammar for tower in Figure 5.11

```
### predefined variables
attr baseHeight = 50

### generated grammar
Lot -->
   extrude(baseHeight) [ s('1.72999, '1.72999, '0.742476) [
   r(scopeCenter, 0, 157*split.index/split.total, 0) [
   split(y){ baseHeight : RuleC }* ] s('0.362006,
   '1.72999, '1.72999) RuleC s('0.362006, '1.72999, '0.742476)
   s('0.362006, '1.72999, '1.72999) [ split(y){ baseHeight :
   [ RuleB s('0.362006, '1.72999, '0.742476) s('0.362006,
   '1.72999, '0.742476) RuleC [ RuleB RuleC s('1.72999,
   '1.72999, '0.742476) RuleC RuleC ] ] }* ] s('0.362006,
   '1.72999, '1.26481) RuleC RuleC RuleC ] s('0.362006,
   '1.72999, '0.742476) RuleC RuleC ]

RuleA -->  r(scopeCenter, 0, 183*split.index/split.total, 0)

RuleB -->
   [ [ r(scopeCenter, 0, 283*split.index/split.total, 0)
   split(x){ baseHeight : r(0, 108*split.index/split.total, 0)
   }* [ [ r(scopeCenter, 0, 283*split.index/split.total,
   0) split(x){ baseHeight : r(0, 108*split.index/split.total,
   0) }* split(y){ 18 : RuleA }* r(scopeCenter, 0,
   279*split.index/split.total, 0) RuleA ] split(x){ baseHeight
   : RuleA } split(y){ 43 : [ RuleA split(y){ baseHeight : RuleA }
   ] }* ] r(scopeCenter, 0, 279*split.index/split.total, 0)
   RuleA ] split(x){ baseHeight : RuleA } split(y){ 43 : [
   r(scopeCenter, 0, 279*split.index/split.total, 0) RuleA
   split(y){ baseHeight : RuleA } ] }* ]

RuleC -->  RuleB split(x){ 16 : RuleB }* split(x){ baseHeight : RuleA }*

### unused rules
RuleD -->  r(scopeCenter, 0, 281*split.index/split.total, 0)

RuleE -->  r(scopeCenter, 0, 252*split.index/split.total, 0)

RuleF -->  s('1.13894, '0.433669, '0.554051)

RuleG -->  s('0.284231, '1.81872, '0.815832)
```

to determine which result generated is "better" then the other results, a summed rank was performed on the Pareto results to determine a ranking. Figure 5.15 shows a building which is 1492.4 units tall, has 2531 unique normals, though breaks the 175x175 unit boundary with a score of 545.1 units. It appears that this building achieves its unique normal count due to the massive amount of narrow spires. The model in Figure 5.16 is the best

result found in the Pareto experiment and has a height of 1184.8 units, 855 unique normals, and breaks the boundary by 550.6 units. Figure 5.17 (a) shows a very interesting model which has a futuristic appeal to it, in which a tower is composed of many block along a slightly winding cylindrical tower. This building has a height of 1442 units, contains 2090 unique normals, and manages to stay within the boundary. However the result in Figure 5.17 (b) shows an outlier which is the opposite of the previous model. It has a height of 851 units, contains a 1758 unique normals, though breaks the boundary with a score of 372.9. That model, although not having a good height or staying within the boundary, still demonstrates an interesting architectural concept from its blocky yet cylindrical pattern.

In conclusion, Pareto shows a wide range of diversity with some good solutions, but often with many outliers. Summed rank has less diversity then Pareto, but has more consistent good solutions. Normalized summed rank created results with the best overall scores, but again, has less diverse results than Pareto.

Chapter 6

Experiments: Advanced Multi-Objective

6.1 Top-Down Shape Matching

6.1.1 Experiment Setup and Parameters

This experiment uses a target shape as a target for the building silouette. Previous experiments made use of a boundary limitation, which essentially created a square area around the model and calculated any error by computing the distance of the furthest points outside of the boundary. This experiment differs such that it allows for a more detailed boundary in the form of shape to match.

The image file that the fitness uses is a black and white image, in which the black area defines the shape for the building to grow within, and the white area defines the out-of-boundary areas. In order for the fitness to compare the building's shape to that of the requested shape, the program creates a 2D vertical projection (orthographic projection) of the model, highlighting the model in black, and the unused space as white. Fitness evaluation then performs a pixel-by-pixel comparison of the 2D projection image and the target shape image. A perfect score would be a score of 1, in which case every rendered pixel from the building matches that of the target image. This fitness evaluation attempts to maximize the building to "fill" the target area, while minimizing the error. Table 6.1 shows the parameters used in this experiment.

Table 6.1: Parameters - Shape Matching

Parameter	Value
Initial Height	50
Generations	60
Tournament	Size of 3
Elites	1
Individual Ranking	Summed Rank

6.1.2 Results

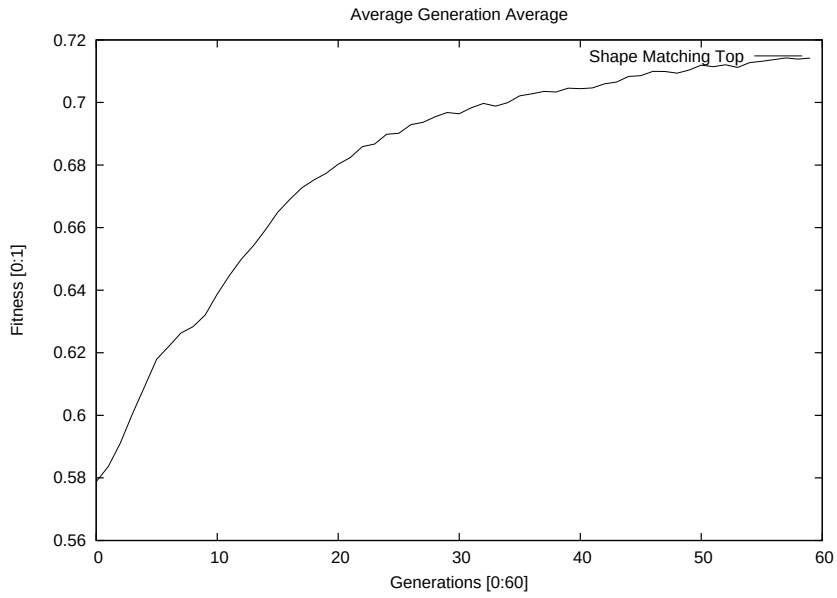
Table 6.2 shows the final fitness scores for this experiment as well as the final generation average for each run.

Table 6.2: Results - Vertical Projection Shape Matching

Run No	Final Best Top Match	Final Pop. Avg. Top Match
1	0.9132	0.9090
2	0.6993	0.6696
3	0.6993	0.6665
4	0.9000	0.8891
5	0.8884	0.8741
6	0.9048	0.8856
7	0.8742	0.8681
8	0.9285	0.9203
9	0.9052	0.9017
10	0.7685	0.7678
Average	0.8481	0.8351

The target image is shown in Figure 6.2 (b). The evolution had a bit of a difficult time providing accurate matches during a couple runs. As shown in Table 6.2, runs 2 and 3 have an identical fitness score of 0.6993. Their

a)



b)

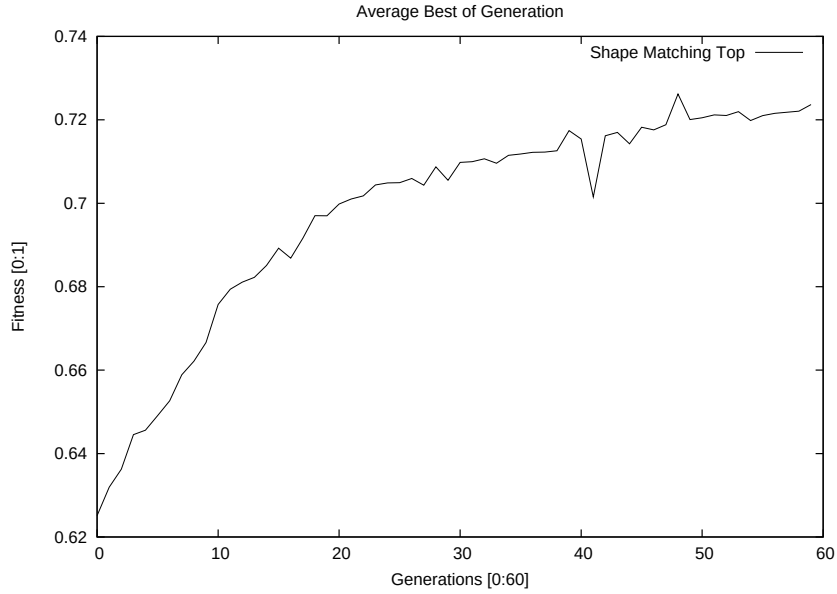
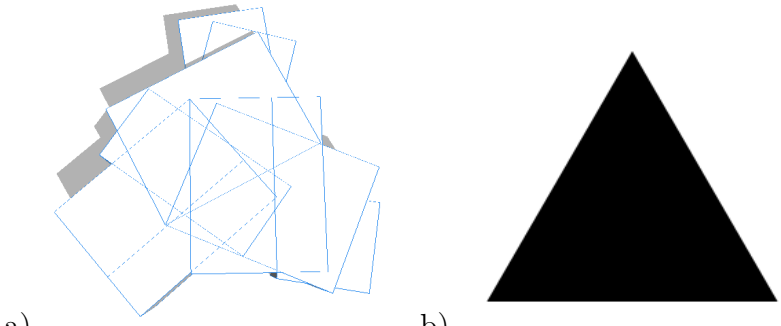
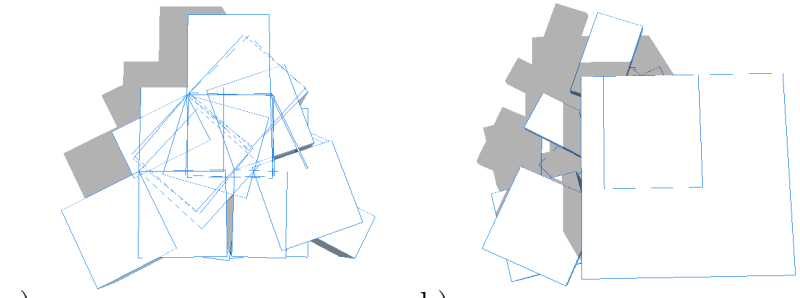


Figure 6.1: Images (a) shows the performance graph of the population over all ten runs, and image (b) shows the performance graph of the average best population over all ten runs.



a) b)

Figure 6.2: Images (a) is from the sixth run and has a score of 0.9048. Image (b) is the target shape for this experiment (not to scale).



a) b)

Figure 6.3: Images (a) is from the eighth run and has the highest score found in the experiment with 0.9285. Image (b) is from the ninth run and has a score of 0.9052.

scores are identical because the final building models are the original starting model provided from the default grammar, which is a 50x50 unit cube.

A more promising result is shown in Figure 6.2 (a) in which the final building has a score of 0.9048 and does a respectful job of resembling the triangle target shape. Even better, the model in Figure 6.3 (a) has a score of 0.9285 and more accurately fills the target shape. A result which deserves a focus is shown in Figure 6.3 (b). This model has a high score of 0.9052 though also makes an interesting pattern with the shadow casted from the building.

6.2 Top-Down and Front-View Shape Matching

6.2.1 Experiment Setup and Parameters

This experiment is similar to the previous one. The fitness attempts to match the vertical projection of the model to the provided shape. However, it also attempts to match the horizontal projection of the model to a different target shape, as shown in Figure 6.5 (b). The parameters used in this experiment, is the same as previous one, and is shown in Table 6.1.

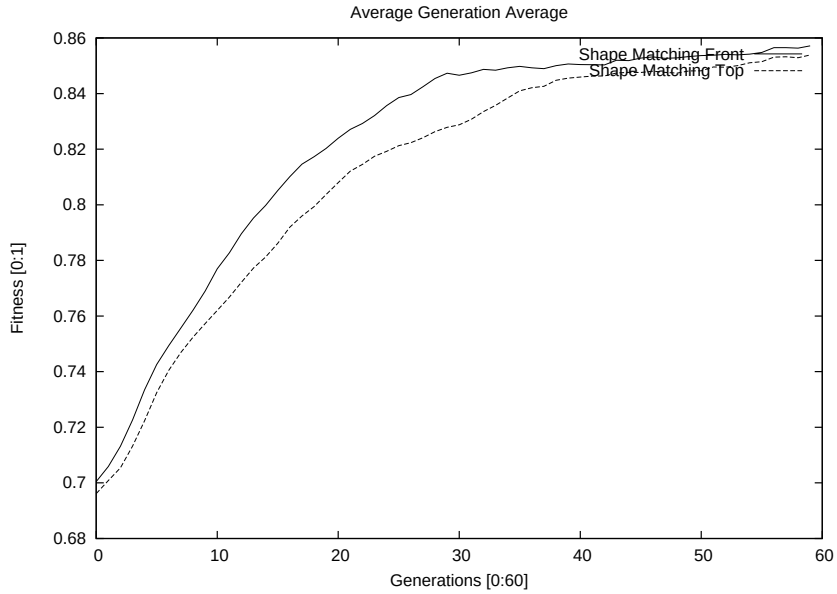
6.2.2 Results

Table 6.3 shows the fitness scores from the best of each run as well as the final generation average.

Shape matching in both the vertical projection as well as the horizontal projection, as shown in Table 6.3. It returns comparable results to the previous experiment. The previous experiment had an average vertical projection matching score of 0.8481, where this experiment has the average score of the vertical matching as 0.8653 and a horizontal projection matching of 0.8918. Figure 6.5 shows the two targeted shapes used in this experiment.

Figure 6.5 shows the model from the third run of the experiment. Image (c) shows the vertical view and image (d) of the same figure shows the horizontal view. This building has a vertical projection score of 0.8878 and a horizontal projection score of 0.8502. The model in Figure 6.5 has a vertical score of 0.8662 and a horizontal score of 0.8738. Image (d) of the same figure shows a decent triangular shape.

a)



b)

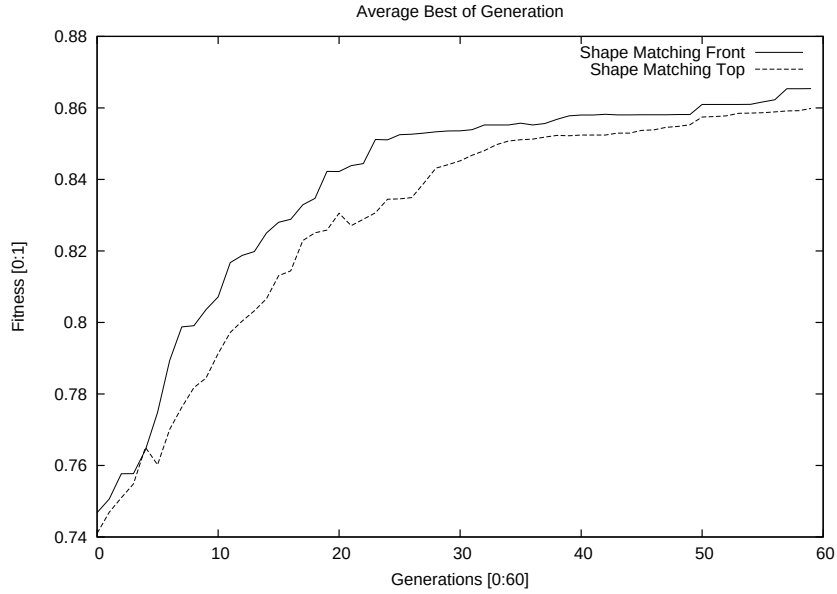


Figure 6.4: Images (a) shows the performance graph of the average generation over all ten runs, and image (b) shows the performance graph of the average best of generation over all ten runs.

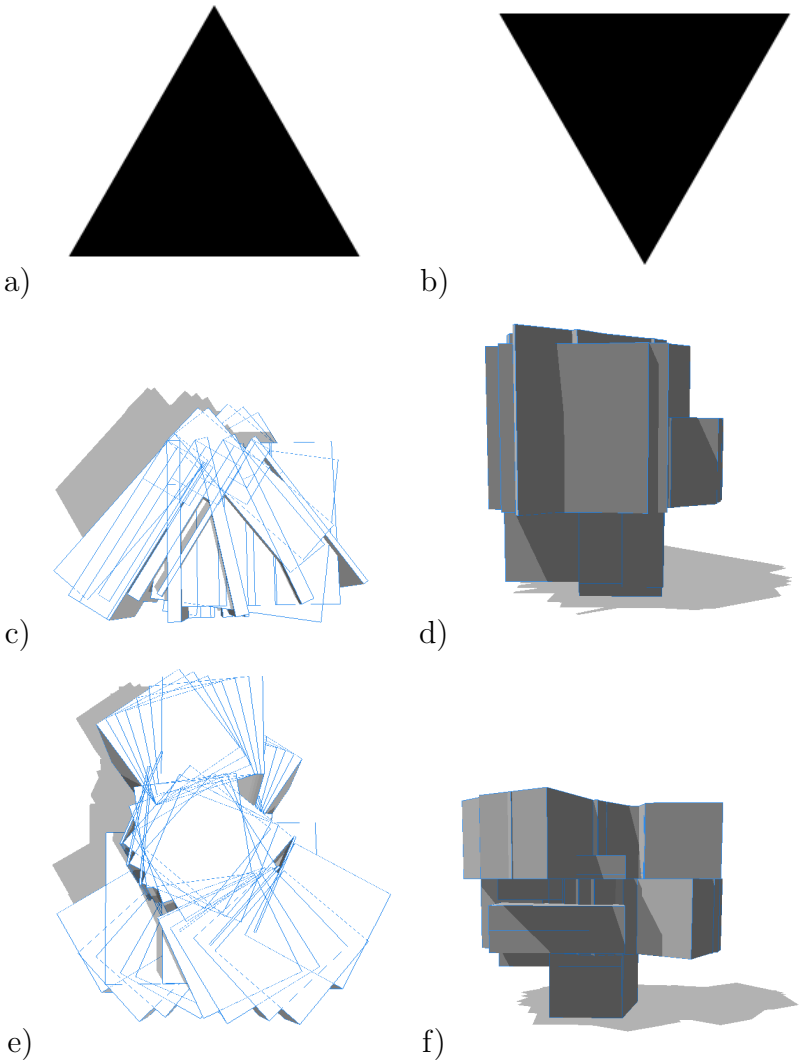


Figure 6.5: Image (a) shows the target vertical shape, and image (b) shows the target horizontal shape. Image (c) shows the vertical view of the model from the third run and has a score of 0.8878 (vertical) and 0.8502 (horizontal). Image (d) is the same model, showing the horizontal view. Image (e) shows the vertical view of the model from the sixth run and has a score of 0.8662 (vertical) and 0.8738 (horizontal). Image (f) is the same model, showing the horizontal view.

Table 6.3: Results - Vertical and Horizontal Projection Shape Matching

	Final Best		Final Pop. Avg.	
Run No	Vert. Match	Horiz. Match	Vert. Match	Horiz. Match
1	0.7717	0.7717	0.7717	0.7717
2	0.8900	0.8919	0.8866	0.8827
3	0.8878	0.8502	0.8861	0.8480
4	0.9157	0.8567	0.9115	0.8533
5	0.8824	0.8871	0.8784	0.8760
6	0.8662	0.8738	0.8559	0.8621
7	0.8832	0.8852	0.8743	0.8757
8	0.8690	0.8318	0.8593	0.8297
9	0.8283	0.8584	0.7980	0.8561
10	0.8596	0.8918	0.8499	0.8837
Average	0.8653	0.8598	0.8571	0.8538

6.3 Top-Down Shape Matching and Maximizing Normals while Targeting Height

6.3.1 Experiment Setup and Parameters

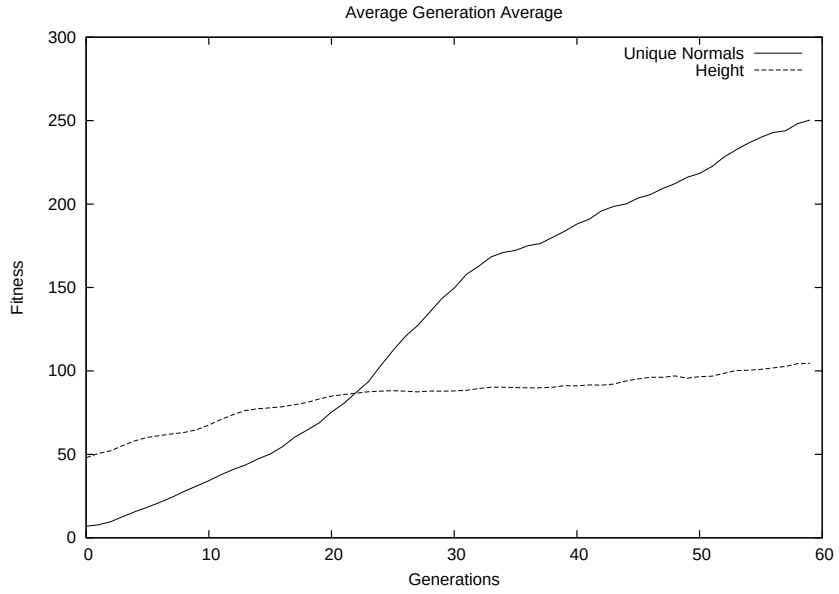
As shown in the previous experiments, two different methods of shape matching can be obtained simultaneously. With this experiment three different fitness functions are used: maximizing the number of unique normals, target the buildings height, as well as match the shape of the vertical projection. Table 6.4 shows the parameters used in this experiment.

6.3.2 Results

Table 6.5 shows the top-ranked results for each run in the experiment, as well as the final generation average.

The results generated by this experiment are decent. As shown in Table 6.5, the average number of unique normals is only 281.8, and the average

a)



b)

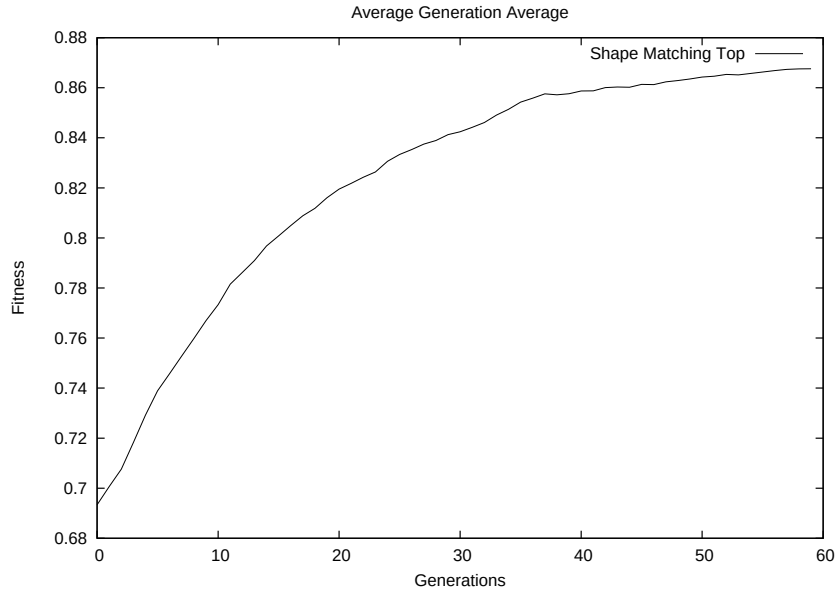
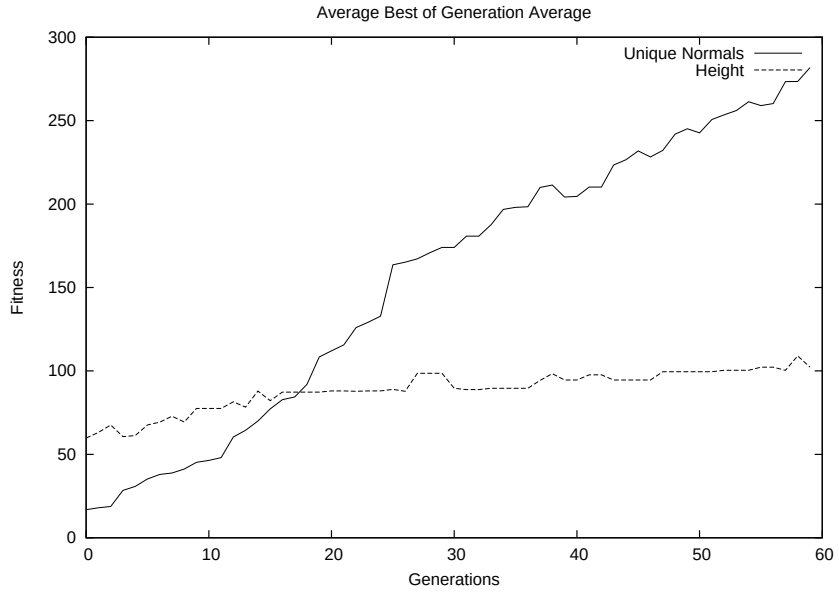


Figure 6.6: Image (a) and (b) show the performance graphs displaying the population averaged over all ten runs.

a)



b)

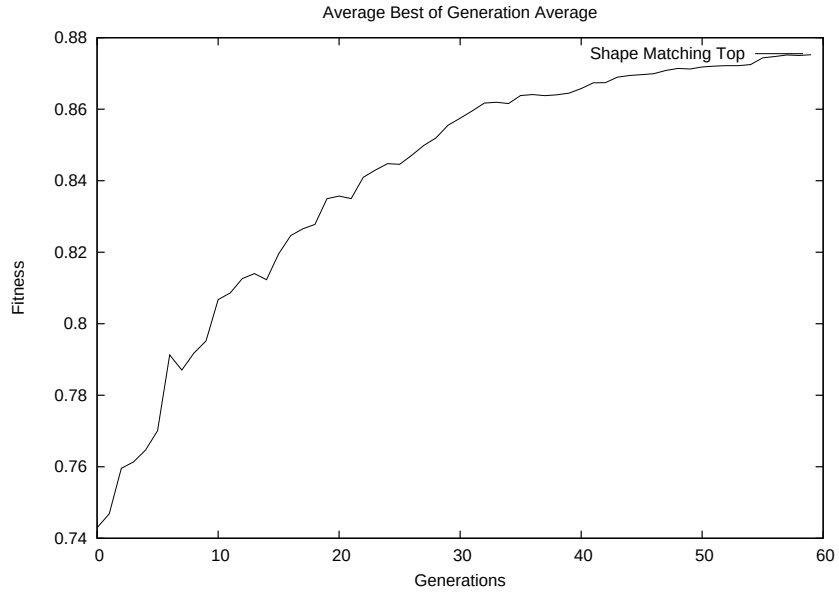
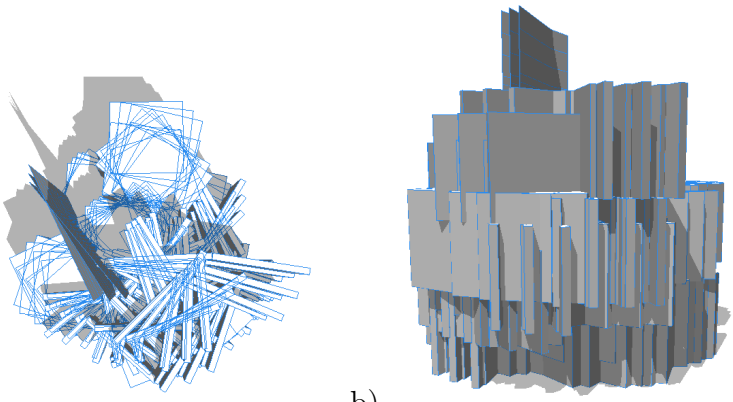
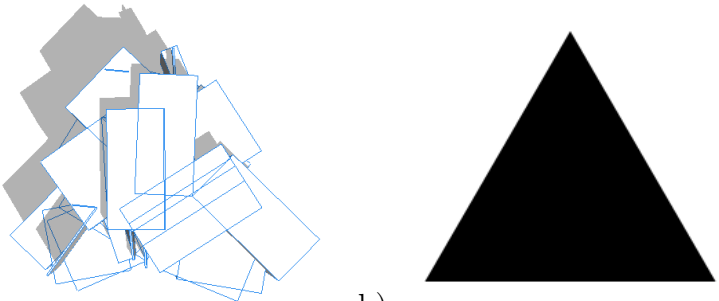


Figure 6.7: Image (a) and (b) show the performance graphs displaying the population averaged over all ten runs.



a) b)

Figure 6.8: Image (a) shows a top-view of the model from the first run of the experiment. This model contains 820 unique normals, a height of 148.652 units, and has a shape matching score of 0.8612. Image (b) is the same model however viewed from the front.



a) b)

Figure 6.9: Image (a) shows the model from the fifth run of the experiment. This model contains 502 unique normals, a height of 68.554 units, and has a shape matching score of 0.9077. Image (b) shows the target shape.

Table 6.4: Parameters - Maximizing Unique Normals, Height Matching, and Shape Matching

Parameter	Value
Unique Normals	Maximize
Target Height	750
Initial Height	50
Generations	60
Tournament	Size of 3
Elites	1
Individual Ranking	Summed Rank

Table 6.5: Results - Maximize Unique Normals, Match Height, and Vertical Projection Shape Matching

	Final Best			Final Pop. Avg.		
Run No	Normals	Height	Shape	Normals	Height	Shape
1	820	148.652	0.8612	727.383	156.576	0.8545
2	274	96.532	0.8927	248.16	96.005	0.8865
3	230	50	0.8778	218.733	50	0.8769
4	6	97.817	0.7760	6	97.456	0.7754
5	502	68.554	0.9077	429.96	53.05	0.8810
6	290	98.514	0.9013	240.147	98.208	0.8918
7	42	94.518	0.8911	40.6133	93.906	0.8838
8	270	114.59	0.8782	239.947	113.986	0.8736
9	82	142.265	0.8786	81.1733	144.426	0.8717
10	302	110.895	0.8878	271.853	142.183	0.8808
Average	281.8000	102.2337	0.8752	250.3969	104.5796	0.8675

height is only 102.2337 units. However the average shape matching portion has a good score of 0.8675.

The model in Figure 6.8 is an interesting building which displays the high-

est count of unique normals achieved in this experiment. This model has 820 unique normals, has a height of 148.652 units, and a vertical shape matching score of 0.8612. Image (a) shows the top down view of the model and image (b) shows the horizontal view. This model has an interesting arrangement of "arms" and a significant amount of intertwining internal squares to account for its high unique normal count.

Figure 6.9 shows the model with the best shape matching score found in this experiment. The vertical projection score is 0.9077, and the model also has a decent 520 unique normals, however has a very low height of only 68.554 units.

An observation based on the previous set of experiments, is that it can be suggested that using a normalized summed rank might provide better results then the regular summed rank method used. With normalized summed rank, the results might improve over all fitness criteria. With the regular summed rank, it appears that the primary focus of the evolution appears to be on the vertical shape matching.

Chapter 7

Discussion

7.1 Constraining the Grammar

CityEngine has a large grammar set. Only a small number of the available commands were used in this study. Initial experimental runs of the evolution system allowed for a much larger grammar set, in which some of the commands included taper, translate, and component splitting. However, these grammar commands, although useful, provided unnecessary complications in the grammar, and cause difficulty in evolving accurate models.

For example, the translate command created many portions of the buildings which were not connected to the main model. The taper command helped create a surplus of unique normals in the models, however was used too often and created non-aesthetically pleasing models. Although the component split is a powerful command, the evolution often used it incorrectly and created holes within the model, or random surfaces which had a thickness of 1 unit. Restricting the available commands also helps minimize the search space and focus the evolution.

In order to evolve accurate results, the grammar had to be tailored. For example, when height is one of the goals of the experiment, the grammar was changed in such a way to allow the evolution to create height easier. One way in which this was done, was to increase the base size of the initial model. The base model normally is a cube of 50x50 units. However, in the height experiment, the base height of the model was increased to 150 units and 50 units wide. This helps the grammar create and find higher models due to the nature of the size command present in the grammar. The size command

re-sizes the model based on the current size of the model.

If the interest is to evolve different styles of skyscrapers, then it would be advantageous to constrain the shape grammar to generate just skyscrapers. Then the GP will search the solution space which constrains only skyscrapers. Otherwise the GP may have to look for skyscrapers in a much larger space of general shapes. The main reason to tailor the grammar is to reduce the size of the search space. By using all the grammar commands at their default settings, it creates a massive search space in which the vast majority of the results are not acceptable to the fitness. However once the grammar is refined, the possible results are also limited, therefore limiting the search space and allowing the evolution to find better results faster.

Another way in which the grammar was tailored was the addition of one variable, known as the $baseHeight$. This variable is an integer number that defines the initial height of the model, as well as is available to be used within the grammar as an integer parameter. This tailors the grammar in such a way that if the base height is tall, the grammar will have a larger number to use within the commands.

If a grammar was to be used in its entirety, without any tailoring, it is capable of creating a multitude of objects, not just buildings. The same commands which were used in this study, could also be used in another study which attempts to create model cars, for example. It is the tailoring of the grammar, combined with fitness scores which focuses the evolution to create the desired object: in this case, buildings.

7.2 Limitations

One of the main limitations of the system is the time it takes to execute one run of the GP. Since the evolutionary system uses an external program to handle the three dimensional model creation and exportation, CityEngine becomes a choke-point. It takes approximately one second for CityEngine to create and export a basic low-polygon model. As the GP runs and provides CityEngine with more detailed grammar files, it has been observed to take up to five seconds to create and export high polygon models. Therefore one run of the GP which consists of 60 generations each of 300 individuals, creates a total of 18000 individuals, all which need to be processed through CityEngine. Taking a look at the average case scenario: 18000 individuals, each being processed at 2 seconds, consumes 10 hours of processing time.

However there are other factors which also lead to more time consumption.

CityEngine unfortunately has a recognized memory leak. Each time a model is created and exported, a proportionate size of system memory is consumed based on the size of the model. It was observed that when a low polygon model was created and exported, that approximately half to one megabyte of memory is used. Where larger polygon models could take up to five megabytes. The CityEngine version used is 2009.2 for Linux 32-bit, released on September 18 2009. The 32-bit version has a memory cap on it such that CityEngine allows up to two gigabytes of memory, and at the time no 64-bit versions for Linux were available. This memory leak caused a major issue in the program run time due to the fact as the individuals were processed and the available memory dwindled away. The time it would take CityEngine to process a model drastically increased to an upwards of 15 to 30 seconds as the available memory disappeared. Once the memory was entirely consumed, the program would grind to a halt. For normal every-day applications of CityEngine this issue would not surface, however for this system one run of the genetic program would require CityEngine to process 18000 models. Taking a look at the average case scenario: 18000 individuals, each consuming an average of 2 megabytes of memory, consumes approximately 36 gigabytes of memory over the run time. Considering the fact that CityEngine only holds two gigabytes of system memory, once 600 individuals, or two generations, have been processed, the majority of the available memory has already been consumed and the program would have already began to lag.

In order to solve this issue, a script was created which would kill the system process running CityEngine, therefore freeing up the consumed memory, after which the script would then restart CityEngine. Due to the fact that the time it takes to close, re-open the program, and connect to the UDP port for communication varies, generous time delays were needed in the script. The total time this fix takes is approximately one minute. In order for this fix to be effective and minimize any slow downs in the processing of the model files, this script would run every 300 individuals, or at the end of each generation. This proved to minimize the negative effects of the memory leak, at the expense of increasing the overall run time of the system. This fix added one minute per generation, adding a total of one hour to the run time of the system, bringing the approximate time of one run to 11 hours.

Moreover, it is not feasible to review the results of one run and make a conclusion on the success of the experiment, so multiple runs needed to be

completed. In order to preform proper statistical analysis on the success of an experiment, a minimum of 30 runs would need to be executed. However, due to the time limitations the system presented, 30 runs would take approximately 330 hours or 14 days to complete, which did not fit in the available time line, considering the number of different experiments performed and that only one computer with a CityEngine license was used.

Chapter 8

Conclusion and Future Work

In this study we were able to successfully evolve building models that fit their respectful fitness requirements. Three dimensional building models were evolved that satisfied constraints such as targeting a specific height, maximizing the number of unique normals present in the model, maximizing the distance between surface normals, matching the building to resemble targeted shapes in both vertical projections as well as horizontal projections, as well as using multiple fitness criteria in combination. The system is capable of exporting those building models to an open-source 3D model file. This model file is ready for use as-in or for use in future development in many different fields of study. The models can be used in fields such as architectural design, video game design, or even for use in animated films.

These models were created though the evolution of their defining shape grammars. Shape grammars, as discussed earlier, are time-consuming and heavily knowledge-based. They can be complex and challenging to development according to specific architectural ideas in which the programmer or designer has in mind. This study successfully evolved those shape grammars that defines building models according to specific architectural ideas though the use of fitness functions and grammar tailoring.

Another accomplishment of this study is the creation of an interesting and innovated design and exploration tool that aids an architect or designer in discovering unique and new architectural ideas and concepts. The program developed explores many possible architectural designs that might not have been conceptualized by a human. This is due to the nature of the program in that is it not a human-guided evolution, but a fully automated one. This allows the evolution to run unbiased and return a wide variety of results.

One way in which the results could be improved is to add additional grammar tailoring and constraints. For example, as mentioned earlier, the component split command was removed from the grammar due to the evolution using that command incorrectly. However, with enough tailoring and restrictions, that command might become a valuable asset to the grammar. Many other commands can also be added to the grammar language in a similar fashion.

There are many grammar enhancements which can be made as well. Since CityEngine allows for custom models to be imported, as shown in the experiment with the low-polygon spheres, models such as columns, pyramids, gargoyles, busts of Abraham Lincoln, or any other models can be used. They can be used to help the genetic program evolve faster, or can be used to give an added flare to the building models, or to even create buildings representative of a specific time period. If one of the target criteria of the genetic program is to create column-like structures, it could be beneficial to provide the genetic program with a column model. This way the genetic program can use the column model instead of wasting generations and processing time in an attempt to evolve them.

The main handicap encountered was the time required to complete one run of the program, as discussed in the previous chapter. Future work can improve on this time issue by using a custom modeling or rendering program, instead of CityEngine. Although CityEngine is a powerful and useful design tool, it was not created for a mass automated production of models as performed in this study. As such, runs were limited to a single computer, on a single-use license of CityEngine. If multiple computers were used, each with a copy of CityEngine on it, the number of runs possible would increase. Due to the time constraints, only 10 runs of each experiment were possible.

There are many new opportunities to explore by adding new fitness criteria. An example of new fitness functions could be evaluating the models based on the standard deviation of the vertices's along a particular axis, such as the y-axis. There are many geometric properties which can be analyzed. The fitness could be to maximize the standard deviation, which might create models with different levels of roofs. Similar criteria could also be done on the other two axes, or could be done to minimize the deviation instead of maximize. There are other areas which can afford statistical analysis as a fitness criteria, such as evaluating the deviation of the area and volume.

Furthermore, another level of evolution can be done on the final building model to enhance it. After the model has been evolved, a separate phase of

evolution can development the exterior design of the model. Such as adding windows, facades, doors, and even textures to the model. The CityEngine environment allows for custom textures which can be implemented directly in the grammar. As such, custom textures can be used in the genetic program as a pure aesthetic addition, or as something new in which to evolve. This can further develop this system by creating models using textures of a specific time period or of a fantasy environment, which can allow the program to evolve even more-specific buildings to match the creative needs of the user.

By adding new and unique fitness criteria and new grammar constraints, buildings of immense uniqueness can be created, discovering wild and unforeseen architectural ideas and possibilities: the experiments completed above merely scratch the surface of the realm of possibilities.

Bibliography

[1] R. Arnaud and M. Barnes. *COLLADA: Sailing the Gulf of 3D Digital Content Creation*. A K Peters, Ltd., 2006.

[2] Autodesk. 3ds max - 3d modeling, animation, and rendering software. http://usa.autodesk.com/3ds-max/, Mar. 2011.

[3] P. J. Bently and J. P. Wakefield. Finding acceptable solutions in the pareto-optimal range using multiobjective genetic algorithms. In *Soft Computing in Engineering Design and Manufacturing*. Spinger Verlag, 1997.

[4] P. Buelow. *Creative Evolutionary Systems*, chapter Using Evolutionary Algorithms to Aid Designers of Architectural Structures. Academic Press, Inc., 2002.

[5] F. D. K. Ching. *Architecture - Form, Space, and Order*, chapter Introduction. Wiley, 2007.

[6] Coello, C. A. Coello, Lamont, G. B., and Veldhuizen. *Evolutionary Algorithms for Solving Multi-Objective Problems*. Kluwer, 2 edition, 2007.

[7] B. David. *A grammar of speech*. Oxford University Press, 1995.

[8] R. Flack. Robgp - robust object based genetic programming system. http://robgp.sourceforge.net/about.php, Nov. 2010.

[9] C. M. Fonseca and P. J. Fleming. An overview of evolutionary algorithms in multiobjective optimization. In *Evolutionary Computation*, pages 3(1):1–16, 1996.

[10] Blender Foundation. Blender. http://www.blender.org/, Mar. 2011.

[11] J. S. Gero, S. J. Louis, and S. Kundu. Evolutionary learning of novel grammars for design improvement. *AIEDAM*, 8:83–94, 1994.

[12] J. S Gero and R. Sosa. Complexity measures as a basis for mass customization of novel designs. Technical Report 1, Dept. of Computer Science, Stanford University, January 2008.

[13] M. Hemberg, U. O'Reilly, A. Menges, K. Jones, M. Goncalves, and S. R. Fuchs. Genr8: Architects' experience with an emergent design tool. In *The Art of Artificial Evolution*, chapter Genr8: Architects' Experience with an Emergent Design Tool. Springer, 2008.

[14] G. S. Hornby. Functional scalability through generative representations: the evolution of table designs. *Environment and Planning B: Planning and Design*, 31(4):569–587, Jan. 2005.

[15] G. S. Hornby and J. B. Pollack. The advantages of generative grammatical encodings for physical design. In *In Congress on Evolutionary Computation*, pages 600–607. IEEE Press, 2001.

[16] Procedural Inc. Cityengine. http://www.procedural.com/, Mar. 2011.

[17] H. Jackson. *Creative Evolutionary Systems*, chapter Toward a Symbiotic Coevolutionary Approach to Architecture. Academic Press, Inc., 2002.

[18] C. Jacob. Genetic l-system programming: Breeding and evolving artificial flowers with mathematica. In *In Proceedings of the First International Mathematica Symposium*, volume 33976, pages 215–222, 1995.

[19] C. Jacob and G. Hushlak. *The Art of Artificial Evolution*, chapter Evolutionary and Swam Design in Science, Art, and Music. Springer, 2008.

[20] J. R. Koza. Hierarchical genetic algorithms operating on populations of computer programs. In N. S. Sridharan, editor, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence IJCAI-89*, volume 1, pages 768–774, Detroit, MI, USA, 20-25 1989. Morgan Kaufmann.

[21] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[22] J. R. Koza. *Genetic Programming II: Automatic Discovery or Reusable Programs*. MIT Press, 1994.

[23] P. Machado, H. Nunes, and J. Romero. Graph-based evolution of visual languages. In *Proc. EvoMusArt*, volume 2, pages 271–280. Springer, 2010.

[24] D. J. Montana. Strongly Typed Genetic Programming. *Evolutionary Computation*, 3(2):199–230, 1995.

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

[26] M. O'Neill and A. Brabazon. Evolving a logo design using lindenmayer systems. In *Evolutionary Computation*, pages 3788 – 3794, June 2008.

[27] M. O'Neill, J. McDermott, J. M. Swafford, J. Byrne, E. Hemberg, and A. Brabazon. Evolutionary design using grammatical evolution and shape grammars: designing a shelter. In *International Journal of Design Engineering*, volume 3, pages 4–24, 2010.

[28] M. O'Neill, J. M. Swafford, J. McDermott, J. Byrne, A. Brabazon, E. Shotton, C. McNally, and M. Hemberg. Shape grammars and grammatical evolution for evolutionary design. *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1035–1042, 2009.

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

[30] R. Poli, W. B. Langdon, and N. F. McPhee. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008.

[31] P. Prusinkiewicz and A. Lindenmayer. *The Algorithmic Beauty of Plants*. Springer, 1990.

[32] G. Rozenberg and A. Salomaa. *Lindenmayer Systems: Impacts on Theoretical Computer Science, Computer Graphics, and Developmental Biology*. Springer, 1 edition, August 1992.

[33] C. Soddu. *Creative Evolutionary Systems*, chapter Recognizability of the Idea: The Evolutionary Process of Argenia. Academic Press, Inc., 2002.

[34] G. Stiny. Introduction to shape and shape grammars. *Environment and Planning B: Planning and Design*, 7(3):343–351, May 1980.

[35] G. Stiny. Kindergarten grammars: designing with froebel's building gifts. *Environment and Planning B: Planning and Design*, 7(4):409–462, July 1980.

[36] G. Stiny and J. Gips. Shape grammars and the generative specification of painting and sculpture. In C. V. Friedman, editor, *Information Processing '71*, pages 1460–1465, Amsterdam, 1972.

[37] M. Tapia. A visual implementation of a shape grammar system. *Environment and Planning B: Planning and Design*, 26:59–73, 1999.

[38] Vitruvius. *Ten Books on Architecture*. BiblioLife, 2009.

Appendix A

Additional Multi-Objective Scores

The following tables display the best result of each of the 10 runs for the three different evaluation methods. Table A.1 shows the results from the normalized summed rank experiment, Table A.2 shows the results from the regular summed rank experiment, and Table A.3 shows the results from the Pareto experiment.

Table A.1: Normalized Summed Rank Results

Run No.	Boundary	Unique Norms	Height Distance
1	0	94	1293.797
2	0	2946	1404.975
3	0	3806	1359
4	0	727	1489.361
5	0	4088	1497.7514
6	0	780	849.951
7	0	370	1462.9896
8	0	1370	939.107
9	0	1624	1475.2029
10	0	728	1357.727
Average	0.0000	1653.3000	1312.9861

Table A.2: Summed Rank Results

Run No.	Boundary	Unique Norms	Height Distance
1	0	342.00	844.667
2	0	659.00	1464.2408
3	0	6.00	847.409
4	0	159.00	1367.077
5	0	3340.00	1497.5269
6	0	2286.00	894.4
7	0	868.00	1457.0161
8	0	501.00	1121.241
9	0	910.00	881
10	0	810.00	817
Average	0.0000	988.1000	1119.1578

Table A.3: Pareto Results

Run No.	Boundary	Unique Norms	Height Distance
1	178.854	462	1020.852
2	545.07	2531	1492.3526
3	181.277	126	1456.9598
4	0	222	848.936
5	0	2228	535.18
6	193.507	890	1407.1536
7	550.637	855	1184.786
8	372.859	1758	851
9	0	2090	1442
10	0	213	947.359
Average	202.2204	1137.5000	1118.6579