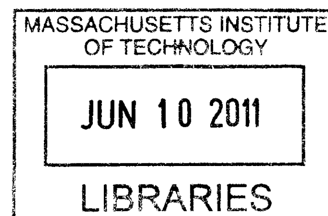
IMAGINATIVE PROCEDURAL MODELING
AUTOMATED 3D GENERATION AND RENDERING OF STYLIZED BUILDING DESIGNS


MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUN 10 2011
LIBRARIES
ARCHIVES

by

Anisha V. Deshmane

Submitted to the
Department of Architecture
in Partial Fulfillment of the Requirements for the Degree of

Bachelor of Science in Art and Design

at the

Massachusetts Institute of Technology

June 2011

© 2011 Deshmane
All Rights Reserved

The author hereby grants to MIT permission to reproduce and to
distribute publicly paper and electronic copies of this thesis document in whole or in part
in any medium now known or hereafter created.

Signature of Author _

Department of Architecture
May 20, 2011

Certified by __

Terry Knight, PhD
Professor of Design and Computation
Thesis Advisor

Accepted by ____________________

Meejin Yoon
Director of the Undergraduate Architecture Program

IMAGINATIVE PROCEDURAL MODELING
AUTOMATED 3D GENERATION AND RENDERING OF STYLIZED BUILDING DESIGNS

by

Anisha V. Deshmane

Submitted to the Department of Architecture
On May 20 2011 in Partial Fulfillment of the
Requirements for the Degree of Bachelor of Science in Art and Design in
Architecture, Computation

ABSTRACT

The entertainment industry relies fairly heavily on computer-generated imagery to depict built environments in current films, video games, and other forms of simulated reality. These often involve highly complex geometries that take a long time to hand-model and are too difficult or costly for many productions' rendering capacities, both in computational costs as well as time.

Procedural modeling and the automation of these geometries is one option to solve these problems. Many modeling programs involve a script or procedural modeling component. This thesis explores the use of CityEngine, a commercially available software that is specialized to generate geometries for buildings in urban environments. By using the CGA Shape Grammar built into CityEngine, this project seeks to generate geometries based on complex architectural precedents using a procedural modeling system based on shape grammar and computational design principles. Results are generated and discussed, as well as applications and future work.

Thesis Supervisor: Terry Knight, PhD.

Title: Professor of Design and Computation

TABLE OF CONTENTS

Introduction ... 5
  Motivation ... 5
  Objectives and Scope ... 5
Background ... 7
  Procedural Modeling and Computer Graphics ... 7
  CityEngine ... 8
Architectural Precedents ... 13
Imaginative Procedural Modeling: Methodology ... 17
  Walls ... 19
  Shapes ... 21
  Deformations ... 22
  Transformations ... 22
Imaginative Procedural Modeling: Results ... 25
Further work/Applications - Discussion ... 28
  Applications In Computer Graphics ... 28
  Applications in Urban Planning ... 28
Bibliographical References ... 29
Appendices ... 30
  Appendix A: CityEngine and CGA Shape ... 30
  Appendix B: Implemented Code ... 34

INTRODUCTION

MOTIVATION

The entertainment industry consists of film, video games, and other media that often uses computer graphics to tell a story or create a simulated reality for users to experience. For these forms of media, the creation of a successful story lies not only in the plot or characters, but also the crafting of a compelling, believable built environment for its characters to exist within. Because of the difficulties in filming or capturing some environments, from medieval or futuristic cities to outer space or under the ocean, computer graphics is commonly in these endeavors because of its capabilities to generate images of objects and scenarios that can't be feasibly shot or captured in reality.

One specific use of computer graphics is to generate architecture and environments for these scenarios, and its use has become widespread in the entertainment industry. All of Pixar's films are entirely computer-generated, and video games use computer graphics to create the environments that the user can explore and interact with. Depending on the scope of the computer graphics within a project, the modeling for the architecture can be done by hand in a 3D modeling software, which can be not only time-consuming and tedious, but also can occupy a computer processor's memory for long periods of time and is computationally very expensive.

These industry problems drive the motivation behind this project. For highly complex, imaginative architecture, generating the geometry in a 3D environment would take a very long time. If there is a way to automate the generation of geometries in a procedural modeling context, several person-years worth of work and computational power could be saved, allowing for production costs to decrease as well as increase the speed of generation of geometries.

This project delves not only into the capabilities of current 3D procedural renderers to realize these geometries but also to generate my own models using a procedural modeling system based on rules that are derived from current complex architecture.

OBJECTIVES AND SCOPE

This thesis involves the use and exploration of CityEngine software in order to create digital renderings and digital 3D models of complex, imaginative architecture using principles of procedural modeling and computational design in an automated computer graphics context. CityEngine is a commercially available software that specializes in the 3D rendering and digital geometric construction of urban environments. The capacities and full utilities of CityEngine will be discussed later in the paper.

By writing scripts within CityEngine using the CityEngine shape grammars, rules were generated that defined various traits observed from various architectural precedents. This thesis excludes the capabilities of CityEngine to import generated models from 3D modeling software such as Maya and 3D Studio Max, limiting the geometries to shapes that CityEngine can create and manipulate.

BACKGROUND

PROCEDURAL MODELING AND COMPUTER GRAPHICS

Procedural modeling is a process of iteratively creating designs based on a set of rules, which dictate how the design evolves. These rules can dictate shape generation as well as transformations on those shapes, and can be used in various combinations to By automating these rules and inserting them into 3D rendering software, models of designs can be easily generated to a high level of geometric detail that would otherwise consume a vast amount of time and memory resources to generate by hand. In applying this concept to cities and buildings, the production process of creating a set or an entire city is made drastically more efficient. [5]

The generation and modeling of large, complex cities has been frequently explored in the study of computer graphics. Many approaches address natural phenomena – for example, ecosystems and plant life. These renderings are helpful to visually describe the complexity of large-scale systems that consist of simpler elements [6], and can create highly detailed renderings of diverse plant life automatically. For example, a rainforest needs to have thousands of different plants, each at a different point in their growth and development, and placed in a seemingly random pattern throughout the scene. The problem of generating a city from scratch is a similarly complex one. CGI cities are very hard to construct because of the vast variety in architectural styles and building types that are necessary to make a city look realistic. Real cities grow and change over time, and as a result are more often than not an incredibly diverse landscape of different styles of architecture of different ages and functions. A procedurally-modeled city needs to take all of this into account as well as the social, historical, cultural, and economic reflections of the time passed in that scene. Thus the procedural modeling must take all of these aspects into account in order to create a realistic looking city[1]. [6]

Some procedural modeling projects use aerial imagery to extract buildings and streets using computer vision, in order to rebuild cities, but are not designed to create new models without photographic data. Others use rules to construct cities, buildings, and houses, but are not formalized, so they cannot be used in the automated creation of an urban environment from scratch. The use of human behavioral data has also been used to generate procedures to build 3D renderings of cities, and from an architectural design approach, shape grammars developed by George Stiny have been used. These grammars consist of a set of shape rules, which define how an existing shape can be transformed, and a generation engine, which processes each rule and applies them to the shape. Shape grammars use production rules that operate directly on shapes rather than on digital strings of symbols, and have been used in interactive design applications varying from 2D to 3D patterns. One approach in particular, CityEngine, is a program capable of modeling a complete city using a very small amount of data. CityEngine creates environments from scratch, which have a high level of control from the user. [6]

[1] See Appendix A for technical details of CityEngine development.

CITYENGINE

CityEngine is a commercially-available procedural modeling software developed by the company Procedural, and is focused on the parametric 3D modeling of city and building geometries. CityEngine was chosen for this project for many reasons, each related to its capacity for generating architecture in urban environments.[7]

CityEngine can import GIS and OpenStreetMap data, which allows a user to recreate real cities or efficiently create an urban environment for a design. By downloading and importing either datatype, the user can model a city that has the look or feel of its real counterpart. These 'atmospherically correct' cities mimic the architectural styles of the buildings while maintaining the realistic street network based off of the data imported, but each building does not necessarily correspond to its real-world component. An example of this is the Venice workflow, seen in Figure 1. [6]

FIGURE 1 VENICE REBUILT IN CITYENGINE. LEFT TO RIGHT: SATELLITE IMAGE, OPENSTREETMAP DATA, GENERATED CITY. [7]

Users can also manipulate any generated street networks to customize their city blocks however they need to. Users can grow streets manually and then change the way that the blocks are generated, for example, beveling the corners of lots and changing where buildings are generated. Users can also define the level of detail required for the buildings generated, allowing for focus on any masses from the simple form of the building to a highly-detailed rendering with intruded windows and extruded balconies or other façade

elements. This allows a user to re-style the city they are modeling, which can be highly useful for films or video games that are set in reality but need customized components.

In a more detailed scope than just the look-and-feel and street data, users can customize the façade elements generated by using the façade wizard. This is useful when a user wants to generate individual buildings that will then have facades modeled to enable a "walk-through" of a city and see the building façades as it passes them. By implementing the façade wizard, Pocedural was able to create rules that are based on an image or textured mass model, resulting in a user-friendly workflow for detailing any façade elements. These uses are seen primarily in the Paris example provided by Procedural, as seen in Figures 2, 3, and 4.

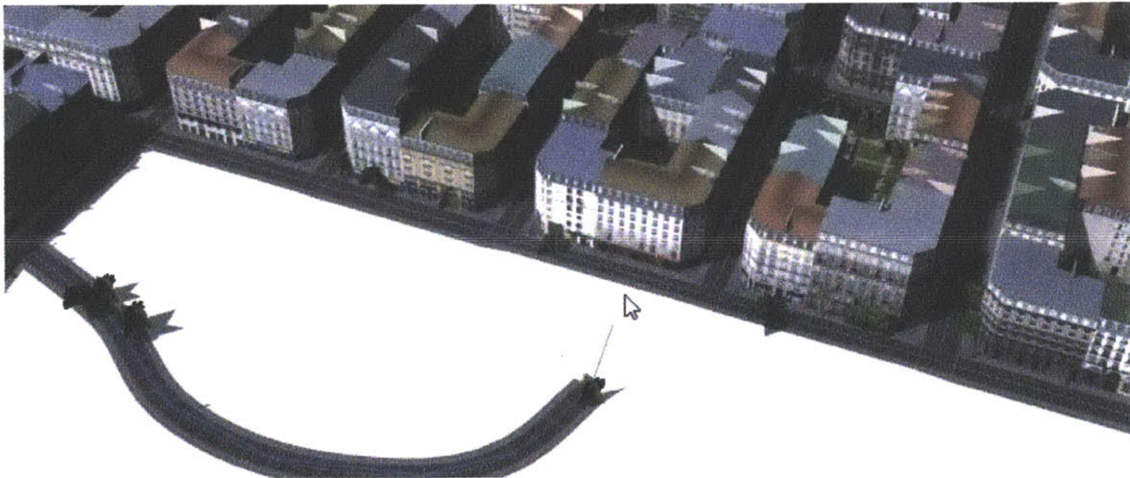


FIGURE 2 STREET MANIPULATION AND CUSTOMIZATION IN CITYENGINE [7]

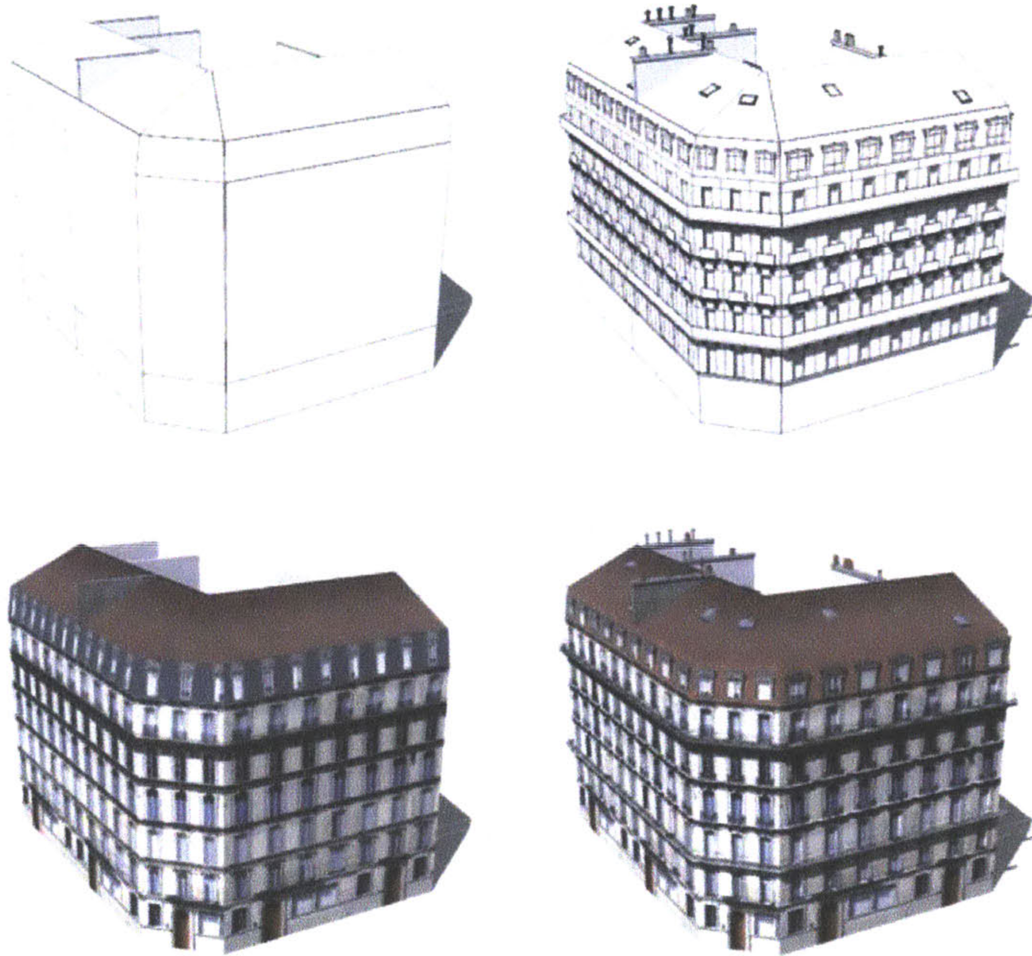


FIGURE 3 LOW TO HIGH LEVEL OF DETAIL ON PARISIAN BUILDING GEOMETRY IN CITYENGINE [7]

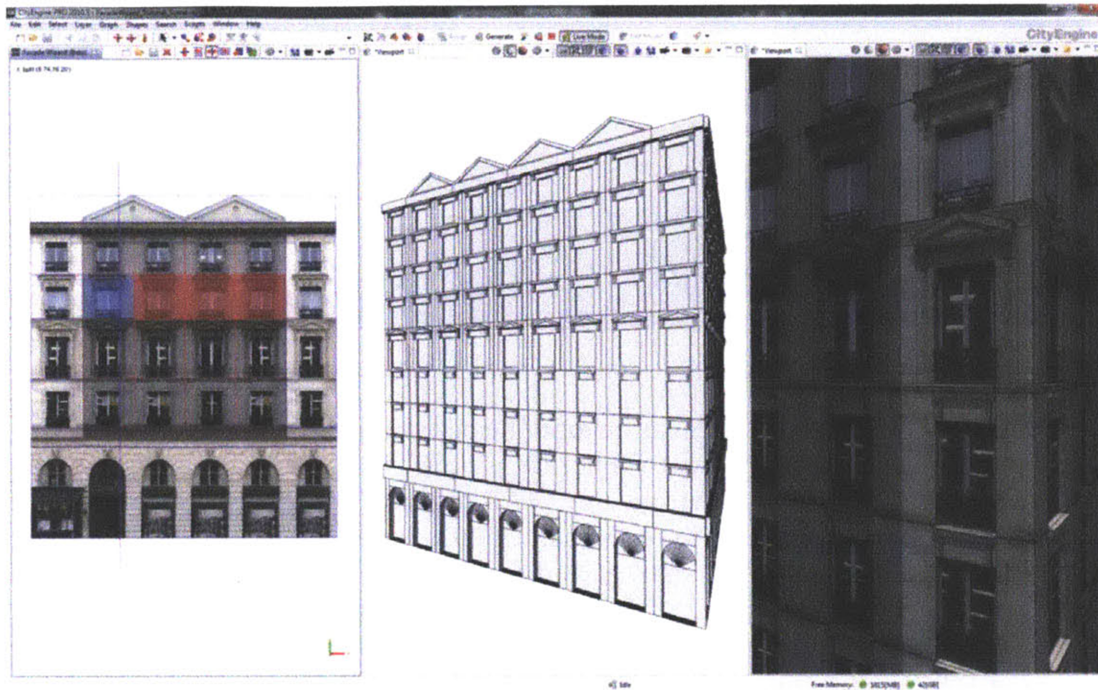


FIGURE 4 FACADE WIZARD SCREENSHOT IN CITYENGINE[7]

CityEngine needs no imports, however, to generate geometry. The program can create street networks and models from scratch and built-in modeling capabilities that the user can manipulate using the CGA Shape Grammar. This shape grammar was developed by Pascal Muller[2] was implemented in this program to use the CGA shape grammar and workflow in order to create rules in a scripting environment that can then be translated to generate geometries. Because of the wide array of customizable features, CityEngine is becoming a commonly-used program in the entertainment industry and has been used on various projects that display its capabilities, for example the upcoming Pixar release, Cars 2, uses CityEngine extensively to model the urban environments of Tokyo and London, but with a distinctly Pixar stylization (figure 5).

[2] See Appendix A

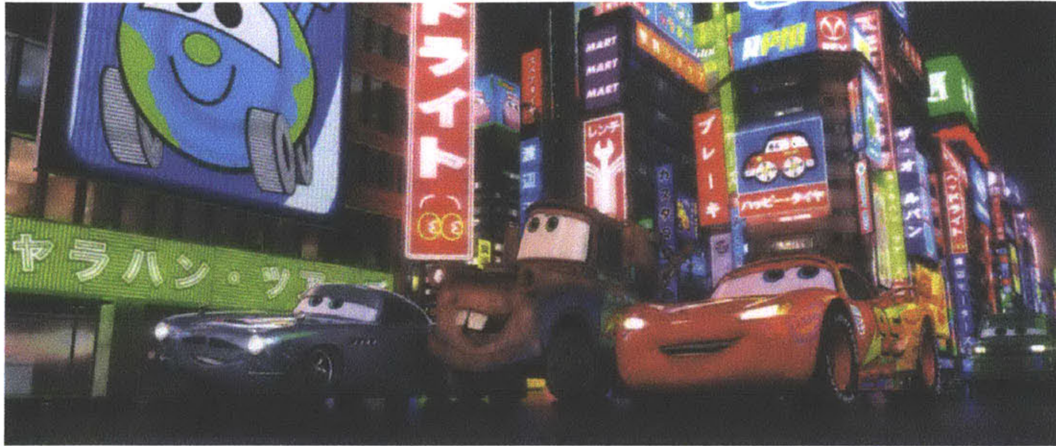


FIGURE 5 CITYENGINE USED TO MODEL TOKYO IN CARS 2(PIXAR ANIMATION STUDIOS, 2011)[3]

The applications of CityEngine range far beyond the entertainment industry, however. CityEngine has been used extensively to generate various historical cities, including Pompeii, the ancient Roman city destroyed in 79 AD (figure 6). [4,6] City Engine modelers used data gained by archaeologists to re-map and generate selected building types to model the city, including placement of trees and streets. This not only allowed many studies to being verifying their results but also begin looking at the way the city was organized from a higher-level, urban planning perspective.

FIGURE 6 POMPEII REGENERATED IN CITYENGINE [4]

---

[3] *http://pixarblog.blogspot.com/2011/04/cars-2-scene-progression-lights-of.html*

ARCHITECTURAL PRECEDENTS

The original goal of this project was to look at fantasy architecture and attempt to recreate architecture from films such as those directed and heavily artistically influenced by Tim Burton, who is noted for his architecturally warped geometries. These sets, although hand-made, would be very computationally expensive for many computer programs to render because of the high complexity and level of detail needed to create such irregular shapes.

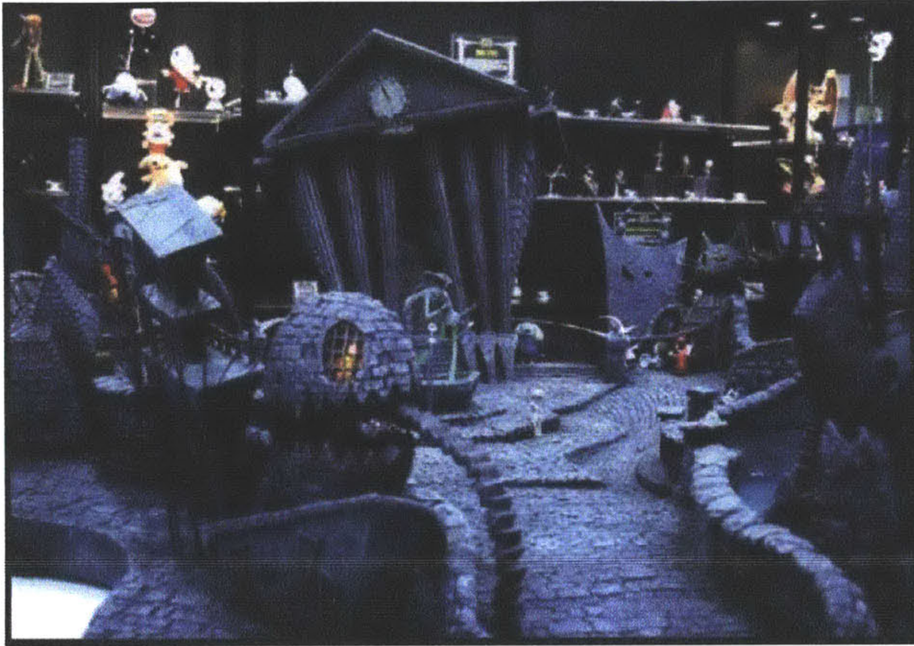


FIGURE 7 SET DESIGN FOR *THE NIGHTMARE BEOFRE CHRISTMAS*[4]

Once I had looked at CityEngine's built in capabilities, I quickly realized that creating these geometries is out of the scope of the shapes that could be created without importing models from other programs. With this in mind, I began looking at architectural precedents that are either concept works or built pieces of architecture around the world that could be analyzed to derive rules about the way their masses were generated. The following images are precedents and rules were derived from their forms that were applied to the geometries generated for this project.

The stacking levels in Figure 8 were particularly intriguing, as well as the offset of the top floor. The skewing of the walls was very interesting as well but the CityEngine capabilities were only able to achieve this in a wall-by-wall basis, which posed its own problems as will be seen later in this document. Figures 9 and 10 show not only vertical stacking but also horizontal stacking, with offsets and skewing as well. The stacked blocks in figure 10 are also translated along one another, and rotated in along another axis. This is a compelling rule to create geometric complexity with a very simple series of rule applications.

The most applicable precedent is that of Allard Architect's Matchbox, a commercial concept design for Amsterdam. This design clearly includes an aggregation rules as well as translation and rotation

[4] http://www.core77.com/gallery/new-york-toy-fair-2008/31.asp

rules. Each of the modular additions to the structure follow the same rules in various combinations. These rules will be implemented in the CityEngine shape grammar rule script for this thesis.

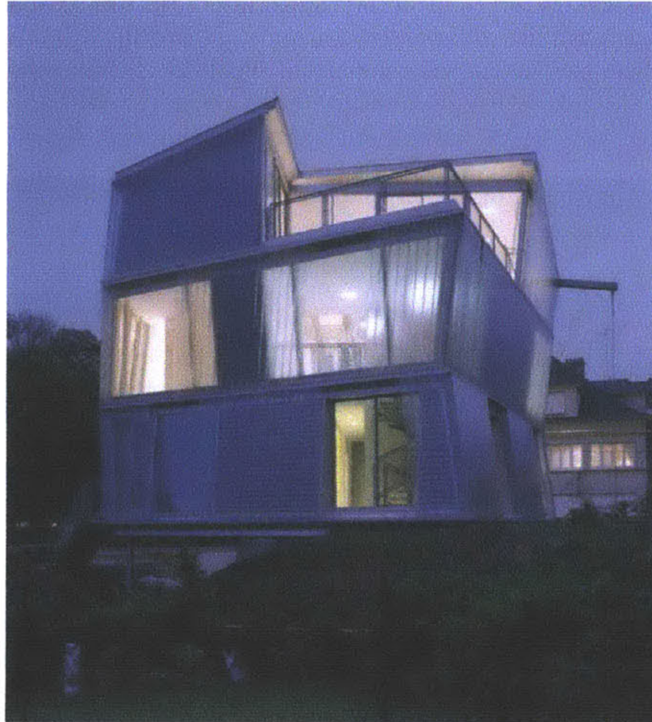


FIGURE 8 MAISON GO, PERIPHERIQUES ARCHITECTES, 2006, THIONVILLE, FRANCE[5].

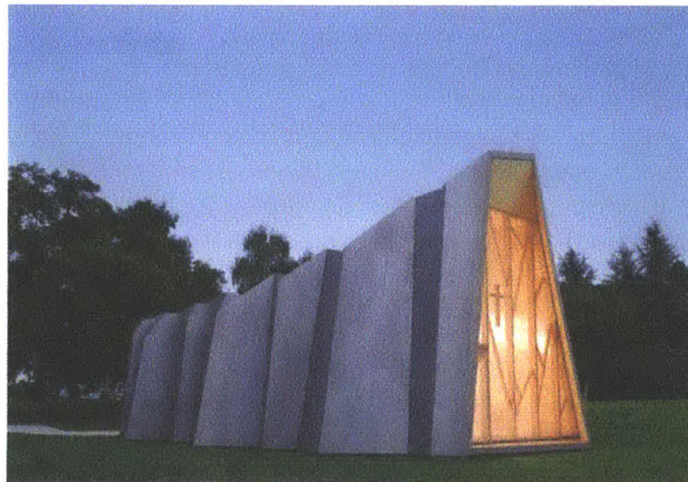


FIGURE 9 HÔPITAL DE ST-LOUP, LOCALARCHITECTURE & DANILO MONDADA, 2008, SWITZERLAND[6]

[5] http://www.archithings.net/a-skewed-view-of-maison-go-home-designed-by-peripheriques-architectes
[6] http://www.archithings.net/a-skewed-view-of-maison-go-home-designed-by-peripheriques-architectes

FIGURE 10 GUARDIAN TOWERS, LAB ARCHITECTURE STUDIO, 2007, ABU DHABI, UAE[7]

FIGURE 11 MATCHBOX, ALLARD ARCHITECTURE, 2010, AMSTERDAM, NETHERLANDS[8]

[7] http://www.arcspace.com/architects/Lab/guardian_towers/gt.html
[8] http://www.architecture-page.com/go/projects/matchbox

All of the examples previously exhibited by CityEngine use imported Maya models in some capacity to generate the geometries shown earlier in this document. As mentioned earlier, this thesis is exploring the capabilities of CityEngine to use its built-in modeling and scripting capabilities to generate geometry rather than importing models in order to create models that mimic some properties of the precedents in the previous section. The focus here is also based on the massing of geometry rather than any façade detailing, maintaining a very low level of detail that will mimic the massing of the aforementioned precedents. The rules that are generated in the following few sections are based simply on the observed transformation and construction of the massing of these precedents.

The CityEngine workflow exists in two panels – one is the scripting panel, in which the rules can be typed out like a normal computer program, and the other is a visual workflow in which the rules are displayed as a flowchart-like diagram that lets the user know how rules are interacting with each other and in what order they're being performed. A viewport can be added that will show the geometries in real-time, allowing you to see any geometries you've generated (figure 12).

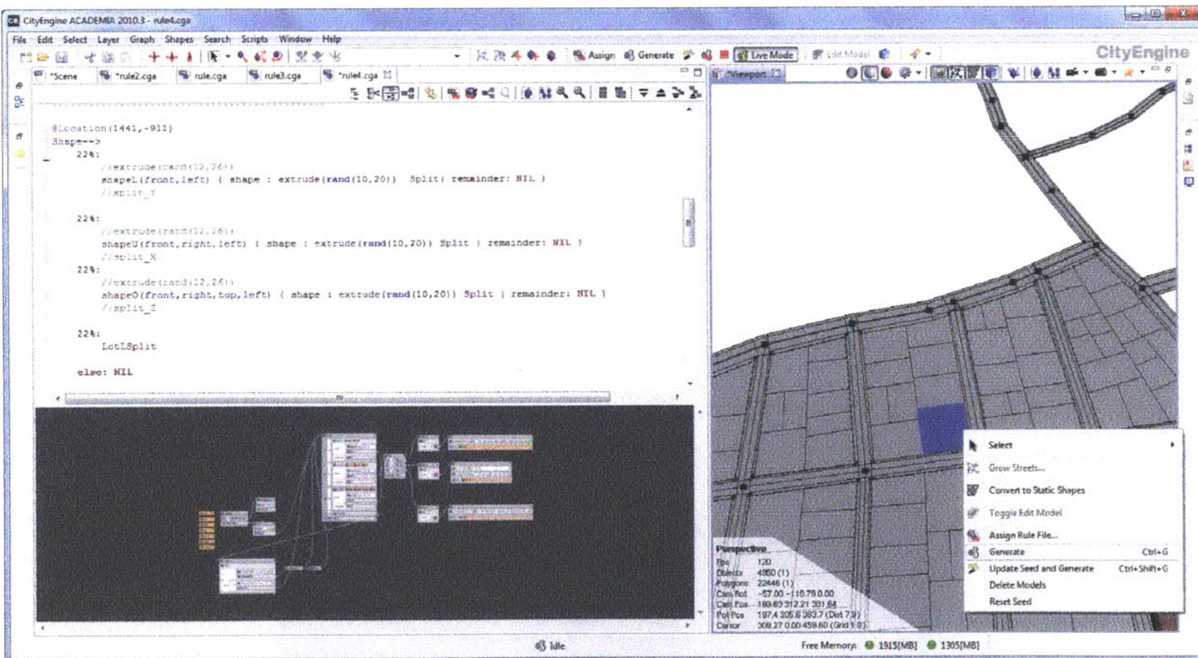
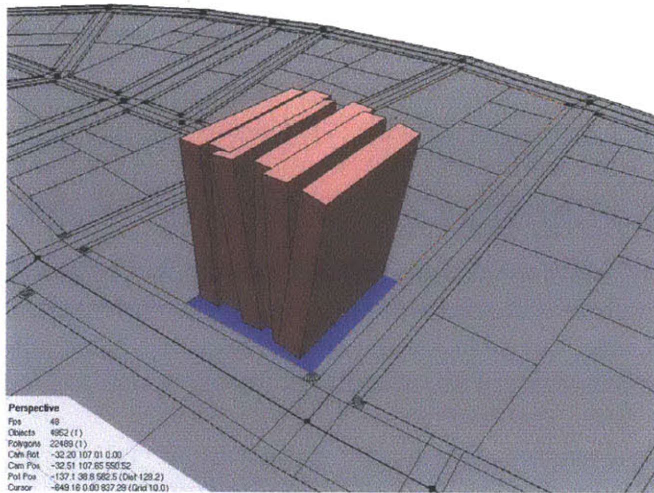
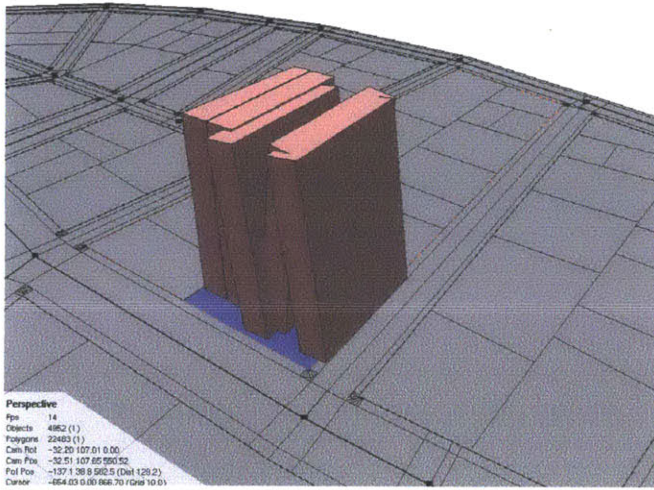


FIGURE 12 SCREEN CAPTURE SHOWING SCRIPTING PLANE, VISUAL WORKFLOW, AND VIEWPORT PANELS

By selecting a lot in the viewport, the user can specify the rule they wish to assign to that piece of land, and then generate innumerably varied geometries based on the rule they applied to that lot. Figure 13 shows three different geometries based off of one rule scripted by using the 'update seed' button.





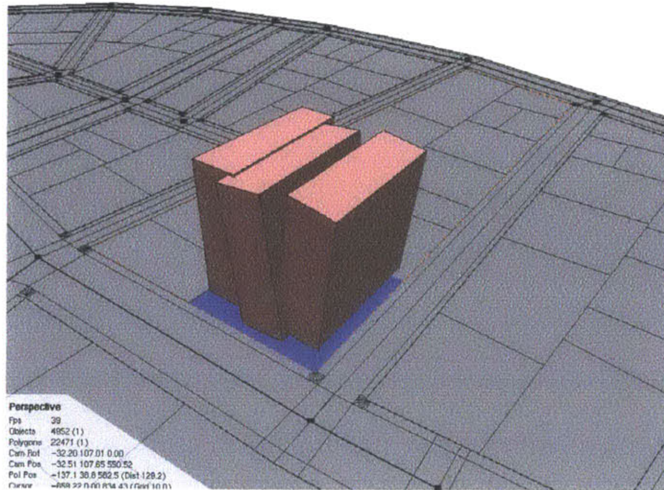


FIGURE 13 VARIOUS DESIGNS GENERATED FROM ONE RULE IN CITYENGINE

In many scripting softwares, it is necessary to rebuild the script if a parameter needs to be changed, but CityEngine allows a user to regenerate geometries easily and instantaneously by changing code and re-updating the seed for the design. In Figure 14, the only parameters that were changed were an axis split direction and a color change, generating the pictured geometry.

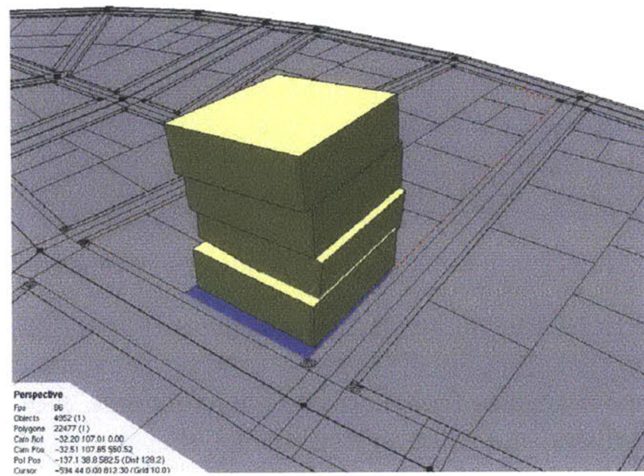


FIGURE 14 MODIFIED GEOMETRY FROM ALTERED SCRIPT

Based on these user-interface capabilities, I began to write scripts that would manipulate individual walls and shapes within CityEngine.

WALLS

CityEngine typically just extrudes the shape of the lot to a user-defined height, but I began by writing a script[9] that would extrude individual walls of the exterior of the lot rather than a solid form, and then perform rotation and translation transformations on them. Figure 15 displays some of my initial results. Tilting the walls backward and forward was easily done.

[9] see Rule 1, Appendix B

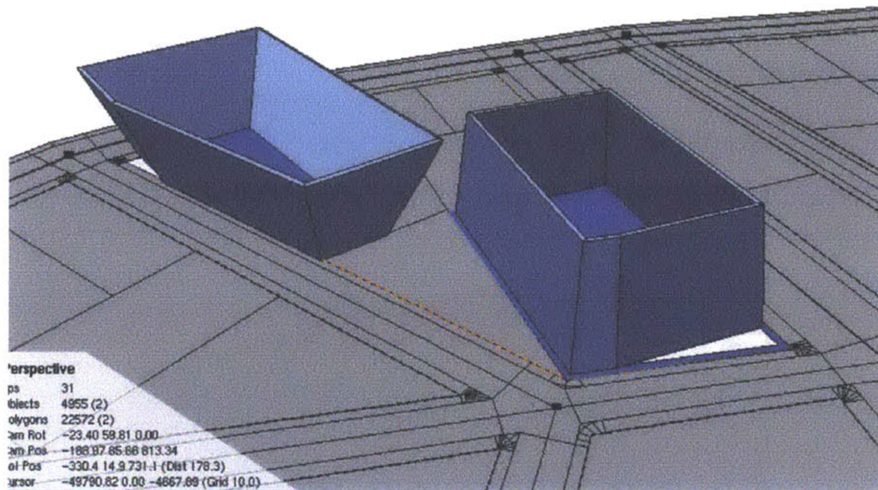


FIGURE 15 TILTING WALLS INWARD AND OUTWARD

By rotating in two directions, the joining of the walls became very difficult. CityEngine has a built-in trim capability that allows walls to know to trim the excess if they intersect, but the walls are not aware of the walls that surround it beyond that. If two walls barely do not meet based on their extrusion and rotation from the lot, they do not know to expand and meet each other, as seen in Figure 16 below.

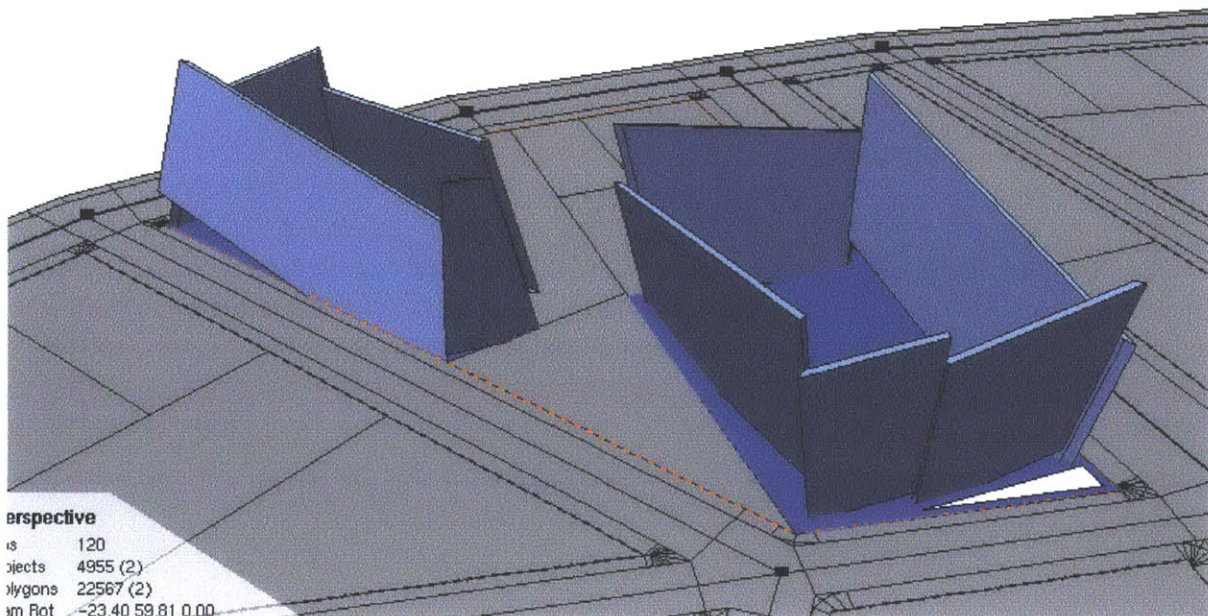


FIGURE 16 ROTATED AND SKEWED WALL MANIPULATIONS

Rotations and skewing also created the problem of holes at the ground level and the joining of the roof section at the top of the geometry. It was clear after these explorations that individual wall manipulation was outside the scope of the project and would not give the desired results, so further exploration in the extrusion and manipulation of solid forms was needed.

SHAPES

CityEngine contains built-in lot manipulations that can be coded into the rules that extrude the lot in various shapes: Lot, L, U, and O, as seen in figure 17 below.

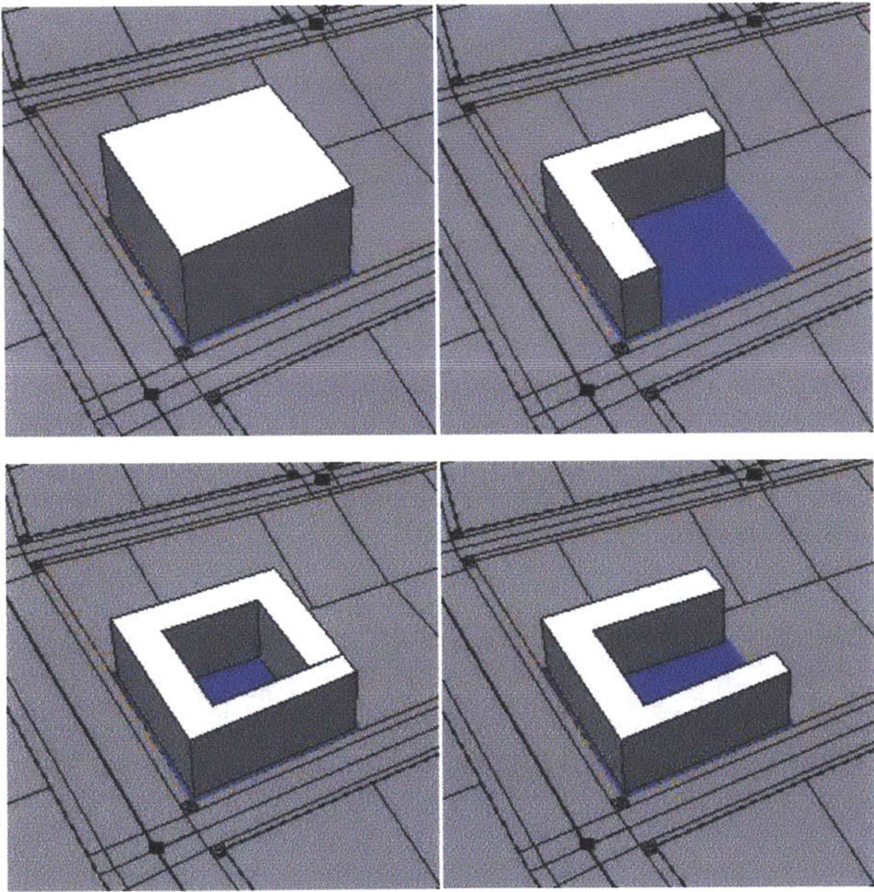


FIGURE 17 LOT EXTRUSIONS IN CITYENGINE. CLOCKWISE FROM TOP LEFT: LOT, L, U, AND O.

These shapes were generated using an early version of Rule 3[10] and, in future versions of the code, were manipulated with the following deformations and transformations.

[10] See Appendix B

DEFORMATIONS

Rules 2 and 3 in Appendix B are the rules that define splitting of not only lots but also the extruded geometry. By assigning a directional axis (X,Y,Z) in the coordinate system, the user can define the direction they want their extruded shape to be split. Parameters exist that can define the minimum width of a split, or the number of divisions required. Figure 18 shows the three splitting directions along the X, Y, and Z axes.

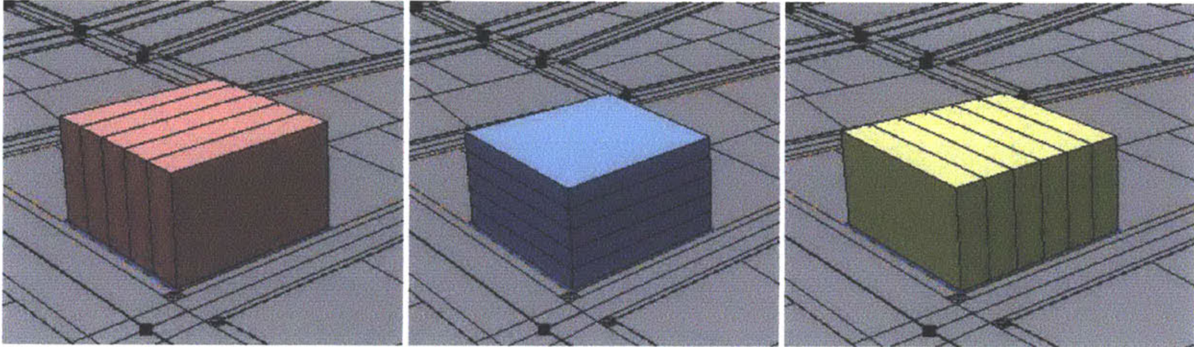


FIGURE 18 SPLITTING OF EXTRUSIONS IN CITYENGINE. LEFT TO RIGHT: X AXIS, Y AXIS, Z AXIS.

TRANSFORMATIONS

Rules 3 and 4 in Appendix B further continue the transformations, including rotations and translations. Once split, each individual piece of the geometry can be manipulated by these two simple transformations.

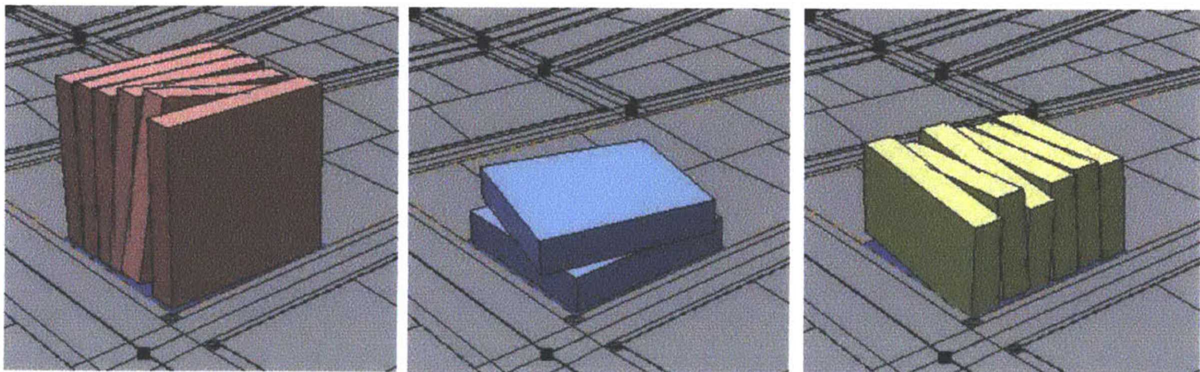


FIGURE 19 ROTATIONS IN X, Y, AND Z

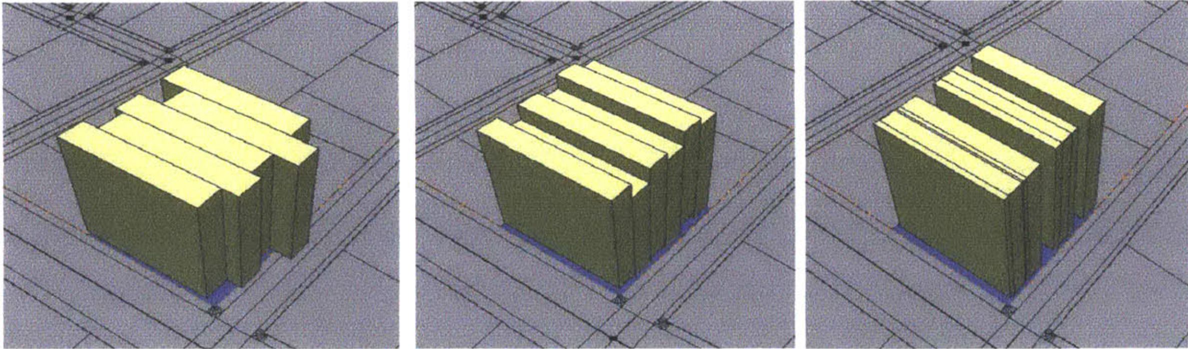


FIGURE 20 TRANSLATIONS IN X, Y, AND Z

By linking these transformations in various combinations in a shape grammar with splitting and color shifting, complex architectural aggregations can be generated, as seen in figure 21. This image was generated by splitting along the Y axis and rotating and translating by a random parameter along the X and Z axes, we can generate the following geometries, one with the simple lot extrusion and the other in an O-shape.

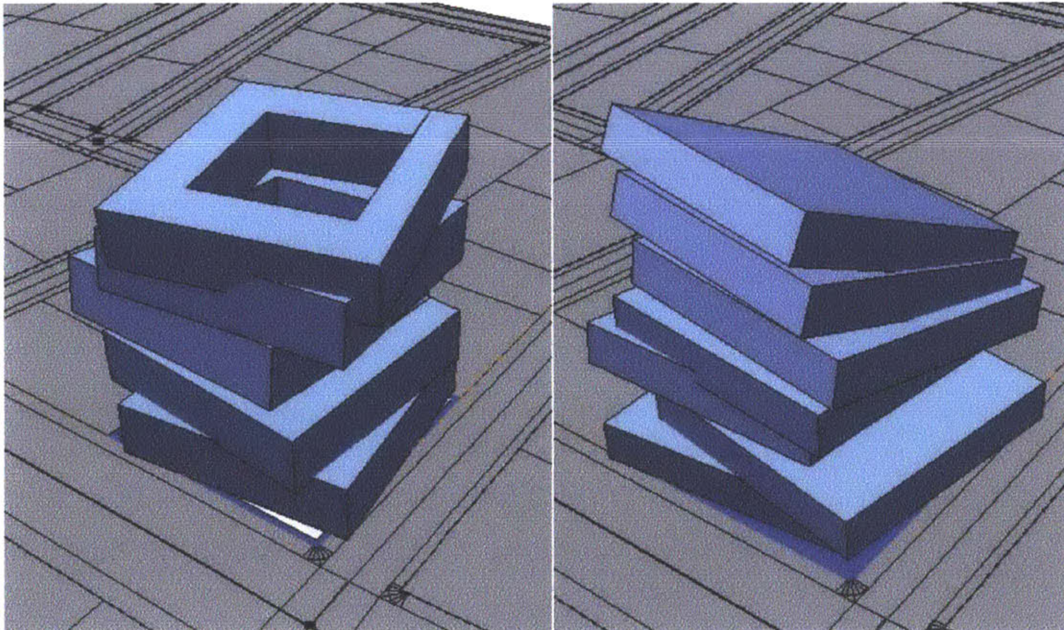


FIGURE 21 EXTRUDED O AND LOT SHAPES WITH COMBINED TRANSFORMATIONS AND SPLITTING

In a single script, parameters can be set as random number or probability generators, which allows for the random selection of shape to extrude as well as random transformations to happen at any point along the flow of the rules. This allows for a greater variety in the designs generated, as well as increasing the complexity of the geometry. The following sections will discuss the generated geometries given the rules written.

By assigning Rule 4 to an entire block of lots in the CityEngine street network, the following results were generated (figure 22-25). By running the script multiple times, various different geometries are created on those lots based on the random number generators.

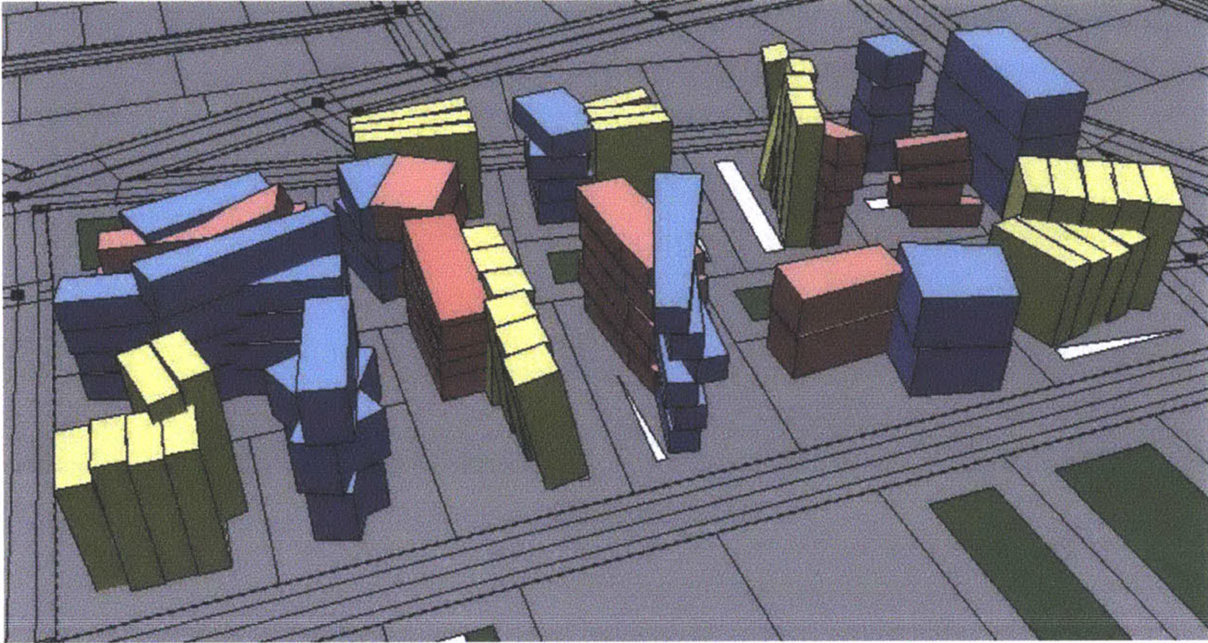


FIGURE 22 GENERATED GEOMETRY

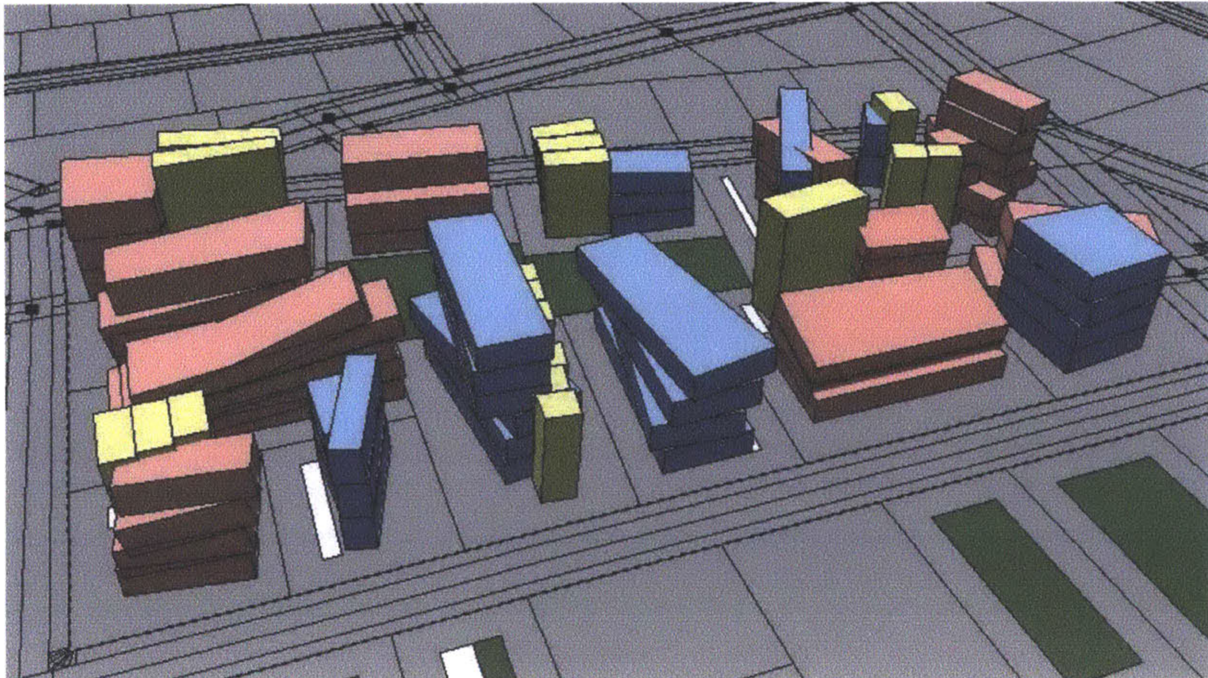


FIGURE 23 GENERATED GEOMETRY

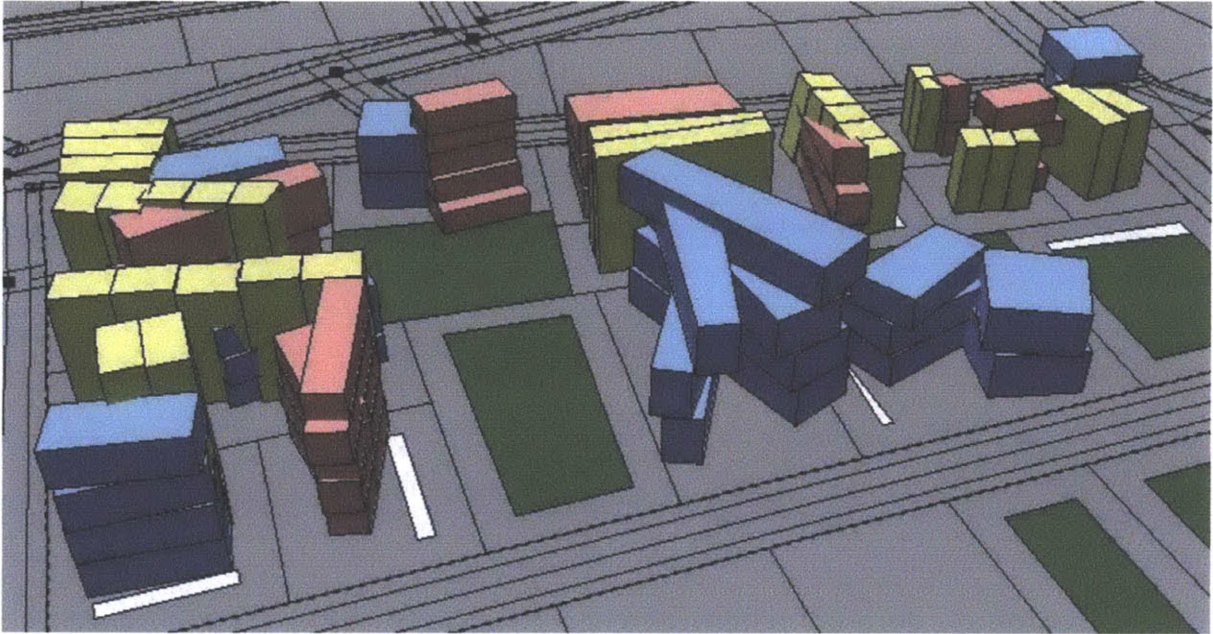


FIGURE 24 GENERATED GEOMETRY

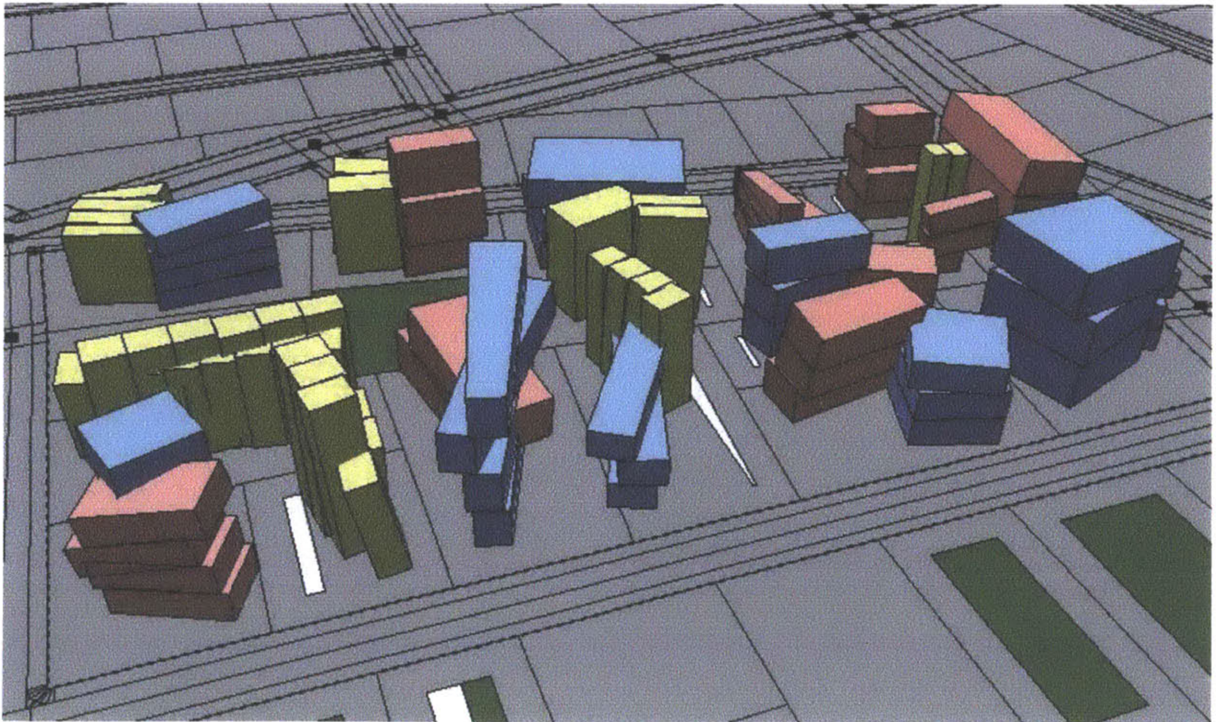


FIGURE 25 GENERATED GEOMETRY

We can begin looking at individual geometries and compare them to the precedents looked at earlier in this document.

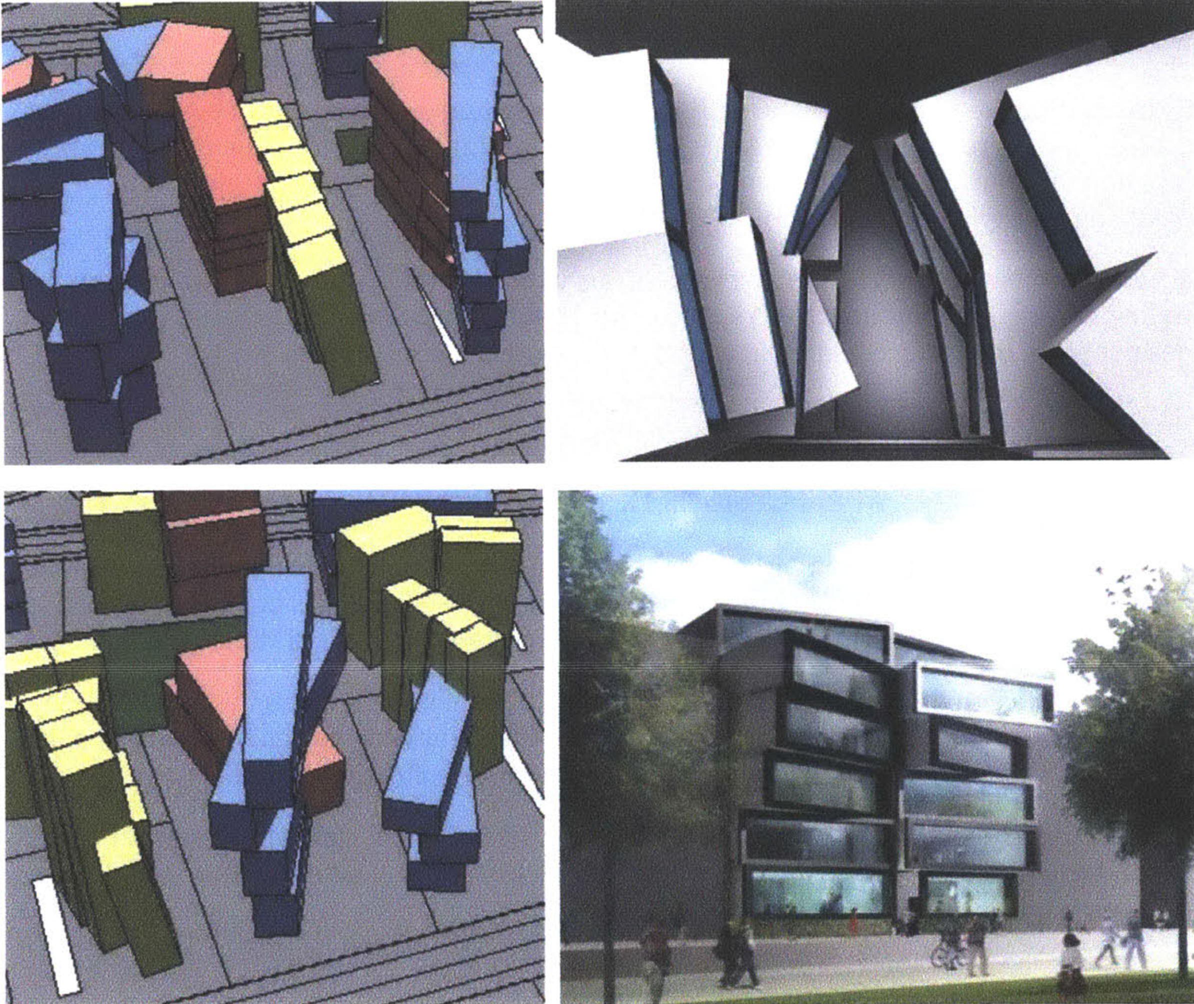


FIGURE 26 COMPARISON OF GENERATED GEOMETRY TO MATCHBOX PRECEDENTS

Some of the generated geometries, seen in figure 26, maintain the familiar tilt and shift traits that Allard Architect's Matchbox concept had as previously observed. By combining the translating, sliding, and rotating rules into the shape grammar, the results are fairly successful in mimicking the architectural and aesthetic qualities of the massing of modern complex architecture.

FURTHER WORK/APPLICATIONS - DISCUSSION

APPLICATIONS IN COMPUTER GRAPHICS

By expanding the library of rules further, more complex geometries could be created that expand upon more complex precedents. By implementing the imported geometry from Maya and 3D Studio Max, more complex geometries can be generated, potentially recreating the style of Tim Burton's set design.

Façade wrapping with such complex geometries would be another, further exploration in CityEngine. Figuring out how to wrap a complex geometry with a façade template that does not seam unevenly could be in and of itself another thesis project.

APPLICATIONS IN URBAN PLANNING

CityEngine can be used not only from a geometric standpoint but also to model urban environments on a larger scale. Rather than looking at the geometries of any particular building, rules can be generated that dictate where certain kinds of buildings are located.

By creating rules that dictate where the city center, commercial districts, residential areas, etc. are located and how they interact with one another, cities can be generated that could be applied to conceptual urban planning models and future city development.

This could also be expanded to be used for concept settlement design where specific rules dictate the geometry of entire developments, rather than the layout of the individual buildings.

REFERENCED WORK

[1] Assassin's Creed II. Patrice Desilets. Videogame. Ubisoft, 2009.

[2] Burton, Tim, Leah Gallo, Holly Kempf, and Derek Frey. *The Art of Tim Burton*. 1st Ed. Steeles Publishing, 2009. Print.

[3] Moltenbrey, Karen. "History in the Making." *Computer Graphics World*. December 2008: 16-25.

[4] Morley, David. "From Vision to Simulated Reality." Planning 74.9 (2008): 28-30. Business Source Complete. EBSCO. Web. 7 Oct. 2010.

[5] Müller, Pascal, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. "Procedural Modeling of Buildings." ACM SIGGRAPH 2006 Papers, SIGGRAPH '06. (2006): 614-623. Print.

[6] Y. I. H. Parish and P. Müller, "Procedural modeling of cities," in Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '01), pp. 301–308, ACM, New York, NY, USA, 2001.

[7] Procedural Inc., (2010). *Procedural: CityEngine Features*. Retrieved from http://www.procedural.com/cityengine

APPENDICES

APPENDIX A: CITYENGINE AND CGA SHAPE

While some models depend on aerial pictures as the main input for building and road construction, CityEngine creates urban environments completely from scratch, based on hierarchical rules set to reflect the user's needs. The generation of the city is simply based on generating a traffic network and buildings, based on the fact that networks, land use, and housing are the slowest changing elements in an urban environment. Streets follow some patterning on different scales, as the transportation medium for an urban population. L-systems, a branching mechanism, support roadmaps well and have the capabilities to generate large-scale road patterns. Based on data collected from London, Paris, Tokyo, and New York, the creators of CityEngine adapted an L-system branching model to enable the creation of similarly large cities. [6]

PIPELINE

The CityEngine system uses a very specific pipeline from user input to rendering the output images to generate a full city. On a more general level, the pipeline is as follows, and may be followed similarly for my project.

A system of roads is generated from user input, using an L-system that will be described later. The space between the roads is subdivided to define the allotments of buildings. By applying another L-system, the buildings are generated as operations on simple solid shapes. Then a parser interprets the results for full visualization. The visualization software can then process the geometries generated and apply texture maps (similarly to any 3D renderer).

The user input primarily consists of 2D image maps, either by drawing or scanning from statistical (population, zoning, street patterns, height) or geographical (elevation, land/water/vegetation) maps. The control of the parameters within each tool can be changed by the user, either interactively through the program's user interface of by providing parameter files. In my project, the user input could be similar: drawn maps that describe the area that the set would be located on.

ROADMAPS AND BUILDINGS

Two different L-systems are used in the program to create one complete city – one to generate the streets, another to generate the buildings. The population density (and hence the kind of buildings created) are based upon the way the network of streets fits together. Two different types of roads are used in the model – highways and streets, which differ in their purpose and behavior. Highways connect highly concentrated populations,

whereas streets traverse the areas between highways, giving each neighborhood adequate transportation to the nearest highway.

An L-system is vital for CityEngine to function effectively. L-systems are mechanisms that rewrite parallel strings based on a set of production rules. For something as complex as a street map, there are many parameters and conditions that have to be implemented into the system, whose complexity grows rapidly. Once a new parameter is introduced, new rules have to be written, and the existing rules must be modified, making the extensibility of the system very difficult. Because of this, the creators of CityEngine made the L-system create a generic template at each step rather than setting the parameters of each module within the system.

Streets, in contrast, do not change the direction based on population but rather follow preset street patterns. In urban areas, streets are usually built following a superimposed pattern plan. When an area with no population is reached, the streets cease growing. Real cities contain a number of road patterns that change based on historical growth and phases of the city through time, whether they are rectangular grids (New York), radial (Paris), branching, or any combination of therein. Different zones in the city have different road patterns that should be followed, based on data gathered from existing cities. The program uses similar patterns and rules that follow population densities and their correlation to different road patterns.

The basic rule for streets follows population densities similarly to how the highway system does. The rectangular grid rule is most frequently used for urban areas, where all the highways and streets are organized in rectangles. The radial rule follows radial tracks around a city center. A special rule was added for elevation changes, where roads on different height levels are connected by smaller streets that follow the steepest elevation—usually used in areas with large, frequent differences in ground elevation, such as San Francisco. This could be useful for films in which the terrain of the sets is specifically rugged.

BUILDINGS

Once the map is generated, the areas between streets are deemed 'blocks.' Blocks are then divided into lots for the placement of buildings. The blocks are divided into lots using a recursive algorithm that divides the longest edges into lots until they reach the size threshold specified by the user. Any lots that are too small or that are closed from street access are discarded.

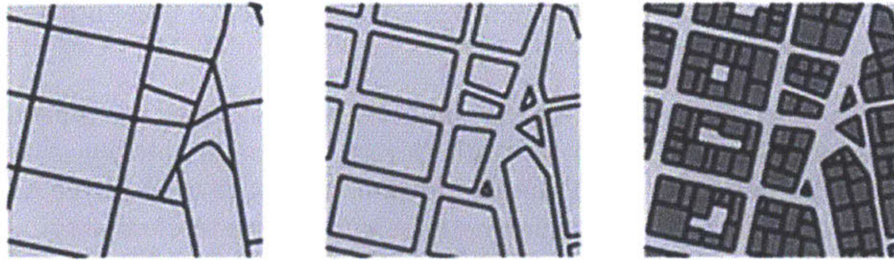


FIGURE 27 ROADS DIVIDED INTO BLOCKS AND THEN TO BUILDING PLOTS

Heights of buildings are usually defined in real cities by zoning regulations, therefore the user should control the height of a building using an image map, restricting the generation of very tall buildings to certain zones of the city.

Buildings are generated based by manipulating the ground plan that satisfies that type of building- skyscrapers, commercial buildings or residential houses. The type of building is determined by zoning rules that are defined by the user. The shape of the building is determined through extrusion and manipulation of the ground plan. Scenic detail and architectural detailing are achieved through the use of textures on the buildings.

The facades are based on one or several overlaid grid-like structures, with each grid cell accomplishing the same function (windows, doors, etc). Each grid cell influences the positions, size, and behavior of the grid cells that surround it. Thus the textures on the buildings are procedural as well

This is where the CityEngine study becomes interesting – the possibility of modeling individual buildings procedurally, rather than simply extruding them. It is possible to change the facades but not the geometry of the specific building. This could potentially be a jumping point for the project: to execute new rules for creating buildings that vary depending on the specific vision of the designer and implementing it into CityEngine to create an entire Tim-Burton-eque city, for example.

FIGURE 28 USER ABILITY TO CHANGE FACADES OF BUILDINGS IN CITYENGINE

This generation of buildings, based off a grammar, is *CGA Shape*, a new shape grammar to procedurally model architecture. [5] CGA shape produces building shells with high geometric detail, providing architectural models for computer games and movies at a very low computational cost, solving previously unexplained modeling problems, especially to

consistent mass modeling with volumetric shapes of arbitrary orientation. CGA shape is shown to efficiently generate massive urban models with unprecedented level of detail, with the virtual rebuilding of the archaeological site of Pompeii as a case in point.

The article discussing CGA Shape is particularly interesting because it shows the conjunction of previous generation techniques that simply extrude buildings and actually generate geometric details on the facades of buildings rather than simply applying a texture map onto a box.

CGA Shape is an extension of set grammars. Unlike CityEngine, which uses L-systems as a parallel grammar to create streets and blocks, CGA Shape is a sequential grammar, which allows for the characterization of structure, the spatial distribution of features and components, which is particularly useful for buildings.

The article describes the specific modeling components that go into the generation and culling of the models. This discussion should be helpful for my implementation once my assessment of different procedural modelers is finished. I believe this same approach to executing the code would be very useful to create buildings that follow a specific design style, especially if I concentrate on only generating structures rather than the organization of the city.

RULE 1 – WALL SPLITTING

```
/**
 * File:    rule.cga
 * Created: 25 Apr 2011 06:32:12 GMT
 * Author:  Anisha
 */

version "2010.3"

###############################################
# Attributes
###############################################

attr height = 20 #rand(15,25)
attr blue2= "#84c0fc"
attr angle = rand(-15,15)
attr wallThickness = 0.8
attr randAlleyWidth = rand(3,5)
alleyprob = p(0.9)
dx = 5

###############################################
# Lot
###############################################

Lot-->
      case alleyprob == true:
            50%:  offset(rand(-2,0), inside)
                  split(x) {randAlleyWidth : Alleysrf. | ~1 : Bldg }
            else: offset(rand(-2,0), inside)
                  split(x) {~1 : Bldg | randAlleyWidth : Alleysrf. }

      else: offset(rand(-2,0), inside) Bldg

###############################################
# Generate Building Walls
###############################################

Bldg -->    extrude(height)
      //Split walls, bring each side of building up as separate wall plane
            comp(f) {   back : WallPlane|
                        front: WallPlane|
                        side: WallPlane}

###############################################
# Wall Planes
###############################################

WallPlane-->      s('3,'1,wallThickness)
                  center(xz)
                  r(scopeCenter,angle,0,angle)
                  i("builltin:cube")
                  color(blue2)
```

RULE 2 – SPLITTING LOTS

```
/**
 * File:    rule2.cga
 * Created: 3 May 2011 04:26:04 GMT
 * Author:  Anisha
 */

version "2010.3"

############################################
# Attributes
############################################

randAlleyWith = rand(1,5)
anglex = rand(0,25*split.index/split.total)
alleyProbability = p(0.6)
attr green = "#FFFF99"
attr blue2= "#84c0fc"
attr reddish = "#fda7a7"

############################################
# Lot
############################################

LotInner -->
      60%:  Lot
      else: MakeGarden

Lot -->
      case alleyProbability == true :
      # in 20% of the cases, split off an alley
            50%:  # split off left
                  offset(-5, inside)
                  split(x){randAlleyWith : AlleySurface. | ~1: LotDesigned}
                  center(xz)

            else: # …or right side
                  offset(-5, inside)
                  split(x){~1: LotDesigned | randAlleyWith : AlleySurface.}
                  center(xz)

      else: offset(-5, inside)
            LotDesigned

LotDesigned -->   # put in different building types
      70%:  MakeLShape
      20%:  FootPrint
      else: MakeGarden


############################################
# Split Lot Shape
############################################
```

```
MakeLShape -->
      split(x){ ~1: r(scopeOrigin,0,rand(0,25),0) LShapeWing | ~rand(.7,.9):
      r(scopeOrigin,0,rand(-25,0),0) FootPrint}

LShapeWing -->
      30% : split(z){ 'rand(0.8,1) : FootPrint | ~1 : NIL }
      else: split(z){ 'rand(0.5,0.8) : Wing }

Wing -->
      75%: FootPrint
      else: FootPrint

#############################################
# Extrusion, Splitting of geometry
#############################################

FootPrint -->
      Split

Split-->
      33%:  extrude(rand(12,26))
            split_Y
      33%:  extrude(rand(12,26))
            split_X
      33%:  extrude(rand(12,26))
            split_Z
      else: NIL

split_X -->

      split(y){~rand(4,8): X}*
            X -->      t(0,0,rand(-6,0))
                       r(scopeOrigin,anglex,0,0)
                       color(reddish)

split_Y -->

      split(y){~rand(4,8): Y}*
            Y -->      r(scopeCenter,0,rand(
                          -25*split.index/split.total,
                          25*split.index/split.total),0)
                       color(blue2)

split_Z -->
      split(z){~rand(4,8): Z}*
            Z -->      r(scopeCenter,0,0,rand(
                             -15*split.index/split.total,
                             15*split.index/split.total))
                       color(green)

#############################################
# Garden
#############################################

MakeGarden -->
      color("#596d4c")
```

RULE 3 - SPLITTING, TRANSFORMING GEOMETRY

```
/**
 * File:    rule3.cga
 * Created: 9 May 2011 04:49:55 GMT
 * Author:  Anisha
 */

version "2010.3"

############################################
# Attributes
############################################

const constBuildingHeight = rand(8,26)
randAlleyWith = rand(1,5)
alleyProbability = p(0.6)
wallThickness = 1
attr green = "#FFFF99"
attr blue2= "#84c0fc"
attr reddish = "#fda7a7"

############################################
# Lots
############################################

LotInner -->
      60%:  Lot
      else: MakeGarden

Lot -->
      case alleyProbability == true :
      # in 20% of the cases, split off an alley
            50%:  # split off left
                  split(x){randAlleyWith : AlleySurface. | ~1: LotDesigned}
                  center(xz)

            else: # or right side
                  split(x){~1: LotDesigned | randAlleyWith : AlleySurface.}
                  center(xz)

      else: LotDesigned

LotDesigned -->
      # different building types
      80%:  MakeLShape
      10%:  FootPrint
      else: MakeGarden

############################################
# L-shaped Lot
############################################

MakeLShape -->
      split(x){ ~1: r(scopeOrigin,0,rand(0,25),0) LShapeWing | ~rand(.7,.9):
      r(scopeOrigin,0,rand(-25,0),0) FootPrint}
```

```
LShapeWing -->
      40% : split(z){ 'rand(0.8,1) : FootPrint | ~1 : NIL }
      else : split(z){ 'rand(0.5,0.8) : Wing }

Wing -->
      75% : FootPrint
      else: FootPrint

#############################################
# Extrusion
#############################################

FootPrint -->
      extrude(constBuildingHeight)
      comp(f) {   back : WallPlane |
                  front: WallPlane |
                  side: WallPlane }

WallPlane-->
      s('1,'1,wallThickness)
      i("builltin:cube")
      split_Y

Split-->
      33%:  split_Y
      33%:  split_X
      33%:  split_Z
      else: NIL

split_X -->
      split(x){~rand(4,8): X}*
      X --> r(scopeCenter,rand(0,25*split.index/split.total),0,0)
            color(reddish)

split_Y -->
      split(y){5: Y}*
      Y --> r(scopeOrigin,-45,0,0)
            center(z)
            i("builltin:cube")
            color(blue2)

split_Z -->
      split(z){~rand(4,8): Z}*
      Z -->       r(scopeCenter,0,0,rand(
                                  -15*split.index/split.total,
                                  15*split.index/split.total))
                  color(green)


#############################################
# Garden
#############################################

MakeGarden -->
      color("#596d4c")
```

RULE 4 – FINAL CODE

```
/**
 * File:    rule4.cga
 * Created: 9 May 2011 20:48:19 GMT
 * Author:  Anisha
 */

version "2010.3"

############################################
# attributes
############################################

attr front = 8
attr left = 8
attr right = 6
attr top = 8
attr green = "#FFFF99"
attr blue2= "#84c0fc"
attr reddish = "#fda7a7"
attr garden = "#596d4c"

############################################
# Lots
############################################

LotInner-->
      25%: MakeGarden
      else: Lot

Lot-->
      offset(-1, inside)
      Shape

############################################
# Choose U,L, O or Lot shape
############################################

Shape-->
      22%:
            shapeL(front,left) { shape : extrude(rand(10,20))  Split|
remainder: NIL }

      22%:
            shapeU(front,right,left) { shape : extrude(rand(10,20)) Split |
remainder: NIL }
      22%:
            shapeO(front,right,top,left) { shape : extrude(rand(10,20)) Split
| remainder: NIL }

      22%:
            LotLSplit
      else: NIL
```

```
###############################################
# Split lot shape
###############################################

LotLSplit-->
      split(x){ ~1: r(scopeOrigin,0,rand(0,25),0) LShapeWing | ~rand(.7,.9):
      r(scopeOrigin,0,rand(-25,0),0) Shape}

LShapeWing -->
      30% : split(z){ 'rand(0.8,1) : Shape | ~1 : NIL }
      else: split(z){ 'rand(0.5,0.8) : Wing }

Wing -->
      75%: Shape
      else: Shape

###############################################
# Determine which direction to split
###############################################

Split-->
      33%: split_Y
      33%: split_X
      33%: split_Z
      else:NIL

split_X --> split(x){~rand(4,8): X}*
            X --> t(0,0,rand(-6,0))
                  r(scopeCenter,rand(-25*split.index/split.total,0),0,0)
                  color(reddish)

split_Y --> split(y){~rand(2,8): Y}*
            Y --> r(scopeCenter,0,rand(-15*split.index/split.total,
                                      25*split.index/split.total),0)
            color(blue2)

split_Z --> split(y){~rand(6,8): Z}*
            Z --> r(scopeCenter,0,0,rand(-20*split.index/split.total,
                                        20*split.index/split.total))
            color(green)

###############################################
# Make lots green to represent a garden
###############################################
MakeGarden -->
      color(garden)
```