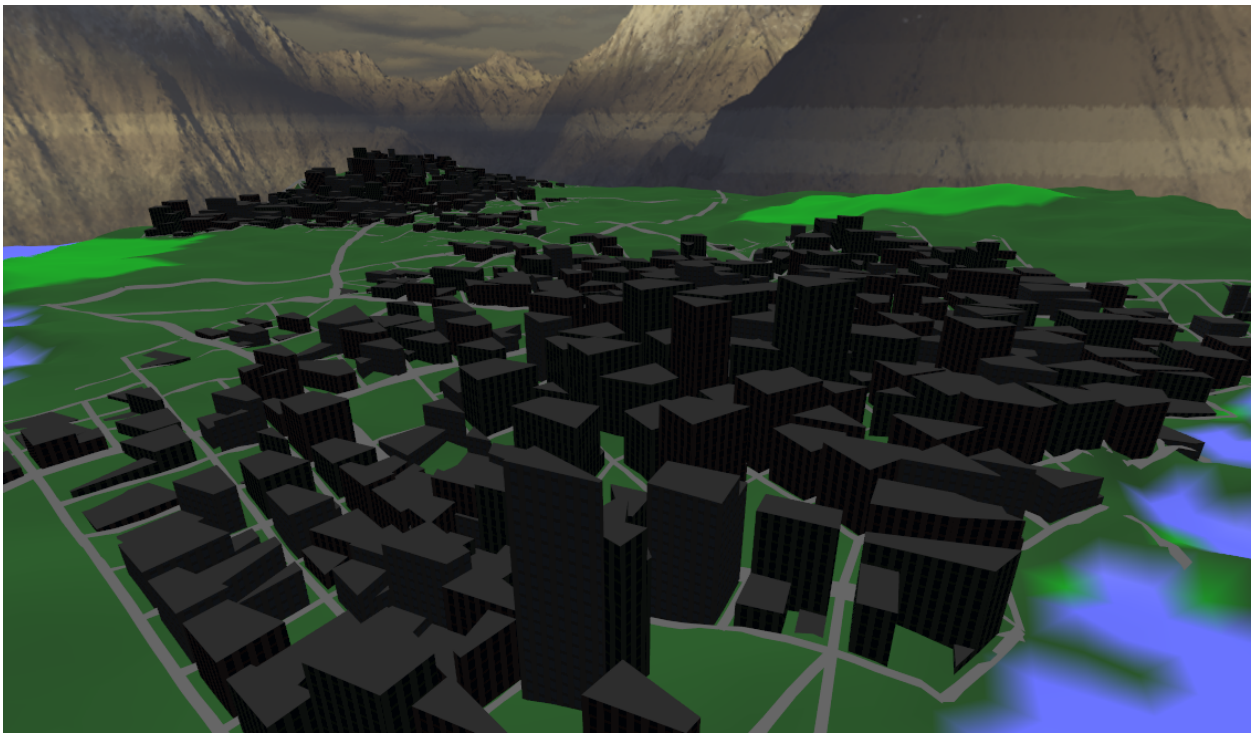
Algorithmic Architecture
CMPT464

Justin Scott Ryan Bujnowicz

27 April 2010



Contents

1 Introduction **1**

2 System Pipeline **1**
2.1 Input Maps 1

3 City Layout & Roads Generation **2**
3.1 Overview 2
3.2 Algorithm 3
3.3 Results 3

4 Procedural Building Generation **4**
4.1 CGA Shape 4
4.2 Geometric Representation 4
4.3 Rule Operations 5
4.4 Production Process 5
4.5 Example 5
4.6 Multiple Building Types 6

5 Implementation **6**

6 Enhancements **6**

A Road Generation Pseudo-code **7**

1 Introduction

As the desire for more and more complex 3D scenes increases, the scale of the work quickly becomes too much for modelers to do by hand within a reasonable amount of time. There is a need to generate large scenes quickly and with enough visual complexity and variation to satisfy the viewer. No where else is this as true as when modeling large city scenes. Cities have a certain regularity to them, the roads follow common patterns and the buildings are of similar shapes, but it is the scale and variation which takes a scene from being a collection of rendered boxes along a grid to being a city.

Procedural generation can be used to generate large amounts of differing and visually interesting models which fit the prescribed constraints of the problem. Grammar based generation systems such as L-systems have previously been employed by the computer graphics community to procedurally render plant geometry with great success. For this project we used similar techniques but applied to the topic of road and building architecture generation.

This project is inspired by papers from Parish[5] and Müller[3].

2 System Pipeline

2.1 Input Maps

It is assumed that the particular design of a city is influenced by the geographic and political circumstances around the city. For example you would expect there to be taller buildings in areas with higher populations densities as single family homes on large plots of land get replaced by tall, dense apartment buildings. Additionally, the design of the city will natural follow land geography, staying away from high mountain peaks and not building structures in the water. Representing these various constraints is a series of image maps, each providing data for different geographical and political variables.

Our project used data from the following type of maps:

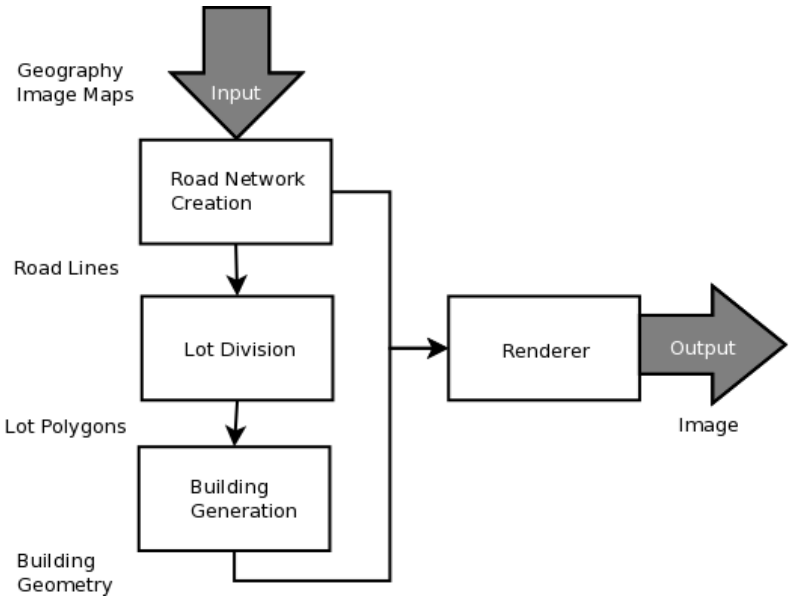


Figure 1: Overview of road and building generation pipeline.

1. Elevation
2. Land type (ground, water, park)
3. Population density

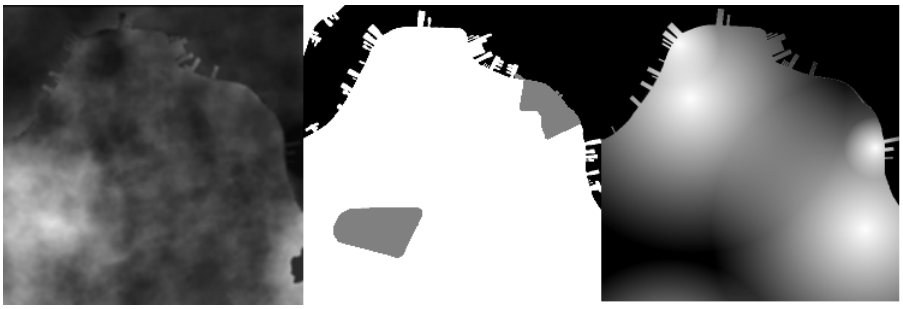


Figure 2: Examples of the elevation, land type and population density maps used.

3 City Layout & Roads Generation

3.1 Overview

Road generation is done through a system that's based on self-sensitive L-Systems. The road map is a tree structure at its core, like many plant generation systems, but differs in that it interacts with itself. This allows for a very organic structure while still maintaining the cyclic nature of a system of roads.

3.2 Algorithm

First a road is generated in the ideal position based on global goals. From the end of the last road it will probe along an arc for the highest population density as shown by Figure 3. This arc is actually quite small so that the road does not turn too sharply, and because a long stretch of highway will be made up of many smaller highway segments. This simulates the need for the road planner to connect population centers with large roads. Road patterns were in the original paper and are used to simulate patterns found in real cities, but were omitted for this project. As such, highways seem like they're built to achieve a goal yet still seem like they're overlapping without an original plan.

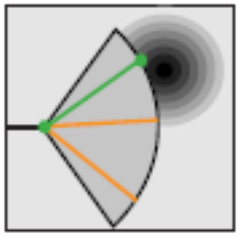


Figure 3: Probing for high density region.

Then the road is subjected to local constraints. The most important of these is road intersections, which are achieved differently in our project than the paper. The paper never gives a proper algorithm but rather a set of cases and how they should end up looking. Our implementation achieves the same effect by testing if end-points of roads are close enough to each other that they should snap together. Essentially, if a road gets too close to another road then it will intersect with it, but it will gravitate toward major intersections instead of wherever it happens to be. This helps make the city more grid-like while still maintaining plenty of areas that are not perfectly aligned. Intersections with certain land zones need to be considered at this point. If the land map shows that the road is entering a park, it will be stopped. If it shows that the road is entering water, it will see if the road can be extended over it to form a bridge, otherwise it will stop as well. Large uphill roads are also avoided by measuring the slope and stopping it if its too steep. Stopping in this case means removing the road and not producing further roads from it. These local constraints make sure that the roads are built around the naturally occurring land while still attempting to achieve their global goals.

Last, the road needs to split up and create branches and road segments to continue the current road. Every set number of road segments the highway will branch two highway segments to its left and right. In addition to this, smaller roads are generated whenever a road segment ends in an area with high enough population. This allows the highway to branch realistically and create the side roads that will be filled with buildings later. The side roads act a bit differently, however. They create another side road forward, but only one branch to their left. This might seem odd, but it creates a very pleasing result when these side roads are generated from every populated highway segment. The overall result is the highways are filled with road grids that provide the perfect landscape to place buildings.

3.3 Results

When you put all these techniques together, you get a very appealing set of roads that simulate a city very well (Figure 4). However, this system does not rely on L-Systems, it only simulates them. The original axiom and production rules can be found in the original paper, but we have replaced it with something resembling the recursive pseudo code found in the appendix. The original ideas of the L-System are there, but without needing to build a superfluous, generic L-System syntax. The ultimate result of this is a system with the discussed properties in a class-based representation that blends L-Systems and undirected graphs, both of which are popular methods in city generation.

Figure 4: Result of road generation algorithm.

4 Procedural Building Generation

Aside from the road geometry, the city layout also provides a graph which divides the land into polygons. These correspond naturally to city blocks. From here we can procedurally generate a building which fits within that block.

4.1 CGA Shape

The key to procedurally generating buildings is called CGA (computer generated architecture) shape, a grammar which recursively describes operations to be performed on pieces of geometry and is capable of creating impressive amounts of detail and variation with very few rules. Like any grammar, CGA shape is defined as a tuple containing a set of terminals (lower case initial letter), a set of non-terminals (upper case initial letter), a set of productions (rules) and a starting symbol (axiom). What separates CGA shape from those grammars is that each symbol also has a piece of geometry (represented by a mesh), a coordinate system (called the scope), as well as numeric properties associated with it. Numeric properties are used to pass high level information to the lower-level shape operations. For example, each symbol has the property "MaxHeight" which specifies the maximum height of the building and is originally set as a function of the population density map. Productions not only produce a symbol derivation but also provide operations which act on the geometry of those symbols.

4.2 Geometric Representation

Each shape has associated with it a specific geometric mesh and a scope. A scope is represented by a translation vector, a scale vector and three vectors representing the X, Y and Z axis of the scope. In practice the scope is used to transform a piece of geometry from its local coordinate space to world coordinate space.

The geometry is represented in a normalized local coordinate system. When the geometry is then transformed by its scope, the final dimensions are given. This allows us to modify only the scope in order to perform operations like scaling, translation and rotation on the geometry.

4.3 Rule Operations

A rule consists of a predecessor symbol, which leads to potentially multiple successor symbols as the result of applying operations on that symbol and its geometry. Multiple rules can exist for a single predecessor with probabilities dictating which of the rules is picked during the production process. Rules are expressed using notation of the form *predecessor → successor(s) : prob.*

Various operations are implemented which in some way manipulate a piece of geometry and return a successor symbol. The two basic types of operations are scope rules and split rules. Scope rules, such as Translate, Rotate and Scale modify the scope of a symbol. For example, the rule $A \rightarrow$ *Translate(4,3,1)* { B } will translate a shape with an A symbol by 4 units along the X-axis, 3 units along the Y-axis and 1 unit along the Z-axis with the newly produced symbol being a B type. Split rules split a geometry and scope along a given axis, resulting in two or more successor shapes. For example the rule $S \rightarrow$ *Subdiv(Z, 1,2,1)* { $A \mid B \mid A$ } will match an S symbol and split it along the Z-axis creating new geometry of size 1, 2 and 1 units of symbols types A, B and A. Implementation of this rule requires the geometric clipping of arbitrary meshes as well as their scopes.

An important consideration when using these rules is the difference between absolute and relative coordinates. For example, with the Subdiv operation above, we split the original shape into three new shapes with the absolute sizes 1, 2 and 1. However, it is more common to want to split a shape by a relevant fraction of its original size. To represent these relative sizes, an 'r' symbol is appended to the size argument in the rule description.

4.4 Production Process

At the start of the building generation process we have a 3D polygon representing the city block output by the road generation. This 'lot' is the axiom, or start symbol, of the grammar. At any given time we can have many active shapes, what is called a shape configuration. For each shape in the shape configuration with a non-terminal symbol, we execute the production rule associated with that shape's symbol and add the newly created symbols to a new shape configuration. Once all shapes in a configuration have been processed, that configuration is discarded and the process continues with the new configuration. In this way symbols are being replaced by their derived successors in the shape configuration. Some productions return ϵ symbols which are not added to the configuration effectively terminating that particular branch. The process halts when there are no non-terminals remaining in the configuration, only terminals.

When a new symbol is created as part of a production rule it inherits the scope and any numerical properties of its predecessor. In this way the numerical properties initially set by higher level productions are accessible at lower levels.

4.5 Example

The main office buildings shown in our demo are developed using a few simple rules, modified from those proposed by Müller[3]. The rules used are the following:

$$
\begin{aligned}
\text{Lot} &\rightarrow \text{Scale}(1r, \text{property['MaxHeight']}, 1r) \\
&\quad\ \text{Subdiv}(\text{Z}, \text{Scope.sz} * \text{rand}(0.3, 0.8), 1r)\{\text{facade} \mid \text{Sidewings}\} \\
\text{Sidewings} &\rightarrow \text{Subdiv}(\text{X}, \text{Scope.sx} * \text{rand}(0.2, 0.6), 1r)\{\text{Sidewing} \mid \epsilon\} \\
&\quad\ \text{Subdiv}(\text{X}, \text{scope.sx} * \text{rand}(0.2, 0.6), 1r)\{\epsilon \mid \ \text{Sidewing}\} \\
\text{Sidewing} &\rightarrow \text{Scale}(1r, \text{Scope.sy} * \text{rand}(0.4, 1.0), 1r)\{\text{facade}\} : 0.8 \\
&\rightarrow \epsilon : 0.2
\end{aligned}
$$

In plain English, what these rules do is take a block outline, represented with the Lot symbol, and extrudes the outline along the Y-axis to get a rough outline of the building shape. In the same rule this

outline is then subdivided along the Z-axis into two randomly portioned pieces. One piece has a symbol of type Sidewings and the other a terminal symbol called facade. The facade represents the end of the derivation for that particular portion of the subdivision. The Sidewings however are further subdivided by another rule into two pieces along the X-axis. Notice that the sizes of the subdivisions are such that these subdivisions will sometimes have space in between them, and sometimes be connected. Finally, the results of that subdivision are processed by the Sidewing rule where with certain probabilities will either scale the wing along the Y-axis or remove it entirely.

Figure 5 demonstrates a selection of configurations resulting from these rules.

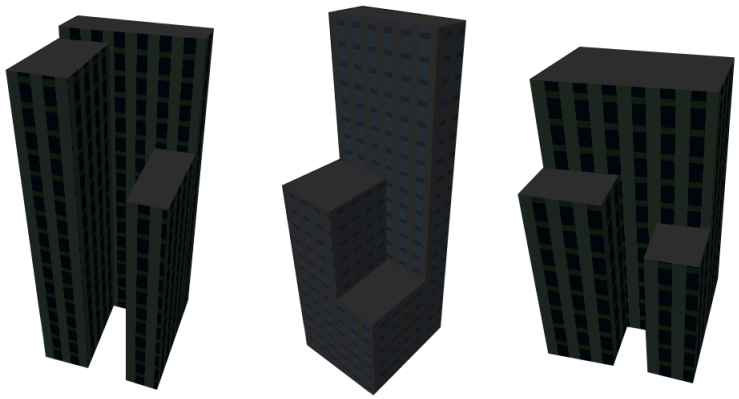


Figure 5: Example of office buildings generated with the given rules.

4.6 Multiple Building Types

Multiple buildings types can be supported by applying a different set of rules depending on the building type you wish to have. By applying even slightly different rules the fundamental look of a building can be radically changed.

In our implementation we set the maximum height of a building to be dependant on the population density of its area. This results in having smaller buildings in less dense areas, and higher buildings in denser areas. This gives the impression of having smaller, commercial buildings on the outskirts of the city with large, towering skyscrapers in the core.

5 Implementation

The project was implemented in C++. OpenGL was used for rendering and Qt was used for windowing and UI.

6 Enhancements

There are a few substantial improvements that still could be made on the road generation algorithm. Most important is the fact that were still held back by using a system based on L-Systems. Its easy to program and understand, and is very intuitive, but more modern, sophisticated systems, such as the one presented by Kelly[2], rely on undirected graphs to structure their roads. This is because there is a wealth of information on graph theory that could vastly improve the computing time and accuracy of cycle detection. Another improvement from the same paper is using curves to approximate roads for better rendering. Roads rarely are formed out of straight lines, but thats what our system is best at. It could look much better if roads were just start and end points and the rest was interpolated via curves. This could also help keep roads level and

veer around obstacles. Last, the intersection detection needs improvement. Right now the roads snap to the endpoints which looks good, but it allows some roads to slip through the cracks in detection. If we treated the road as a line segment instead of just endpoints then we would have a much more accurate detection as in Figure 6. There are many various improvements that could be made, but were unable to make the cut due to time constraints and complexity.

Figure 6: Collision detection of road as a line segment.

As implemented the buildings do not have any facade detail other than a texture. Müller[3] describes a method for adding facade detail including the correct placement of windows and other details using snap lines. Additionally, support can be added to load in particular geometry instances. The most useful use of this would be for adding peaked roofs onto houses and buildings.

The actual program at this point is merely a demo of the results that we were able to generate from the road generation and building generation systems. Adding more user control as to how the roads and buildings are generated, as well as being able to save a particular configuration of a city would help to increase usability.

Appendices

A Road Generation Pseudo-code

```
processRoad(Road* road, Road* root)
{
    if(road->delayed == false)
    {
        if(road->state == Success)
        {
            road.globalRules();// === GLOBAL RULES ===
        }
        else if(road->state == Failure) // Slated for removal
        {
            road->delay = -1;
        }
```

```
    }

    if(road->delayed == true)
    {
        if(road->delay > 0) road->delay--; // Counting down
        else if(road->delay == 0) // No longer delayed
        {
            road->state = Unassigned;
            road->delayed = false;
        }
        else {} // Slated for removal
    }

    if(road->delayed == false)
    {
        if(road->delay < 0) {} // Slated for removal
        else if(road->state == Unassigned)
        {
            // === LOCAL CONSTRAINTS ===

            // Stop it if it intersects with another road
            if (road.intersection()) road.finish();
            else road->state = Success;
        }
        else if(road->state != Unassigned)
        // Slated for removal
        {
            road->delay = -1;
        }
    }

    foreach (child in road->children)
    {
        // Remove children that are marked for it
        if(child->delay < 0) delete road->children[i];
        // And process the rest
        else processRoad(child, root);
    }
}

globalRules()
{
    // Highway rules
    if(road->highway)
    {
        // Create 1 road forward...
        newHighway (forward);

        // And 2 branches or subroads
        if (road->lastJunction > JUNCTION_DISTANCE)
        {
```

```
            newHighway(left); // Highways have no delay
            newHighway(right);
        }
        else if (popDensity > SUBROAD_DENSITY)
        {
            newSubroad(left); // Subroads have a high delay
            newSubroad(right);
        }
    }

    // Subroad rules
    else if (popDensity > SUBROAD_DENSITY)
    {
        newSubroad(forward);
        newSubroad(left);
    }

    // Finalize the road
    road.finish();
}
```

References

[1] John "Lintfordpickle" Hampson. Procedural road generation. `http://britonia-game.com/?p=28`.

[2] G. Kelly and H. McCabe. Citygen: An interactive system for procedural city generation. *Fifth International Conference on Game Design and Technology*, pages 8–16, 2007.

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

[4] Radomír Měch and Przemyslaw Prusinkiewicz. Visual models of plants interacting with their environment. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 397–410, New York, NY, USA, 1996. ACM.

[5] Yoav I. H. Parish and Pascal 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.