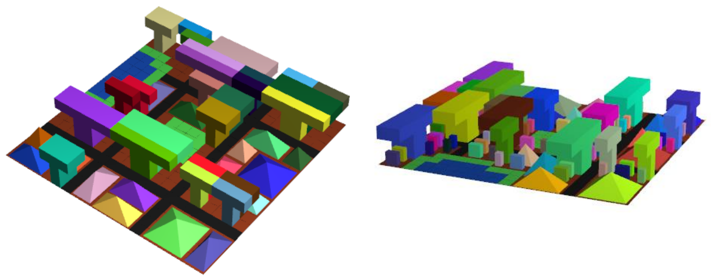
Procedural Modeling of Cities with User-Created Models

Rachel King & Michael Mortimer
Advanced Computer Graphics, Spring 2015



0. Abstract

Our goal in this project was to create procedurally generated city models using basic map constraints and input buildings. We created an application to fill map grids specified in simple, representative file inputs with a series of arbitrarily altered, user-specified buildings.

Our method handles models of an arbitrary size and shape, checks for building collisions, and prevents placement in maps spaces marked unsuitable for new development. It successfully generates densely packed and non-identical buildings that fit within given constraints, providing a user with a reasonable, customizable, low-effort city model.

1. Introduction

The modeling of cityscapes for various applications is a fairly investigated topic in computer graphics. Applications range from purely artistic use, as in games and movies, to those related to scholarship and engineering – archaeology, urban planning, and civil engineering. Such models tend to be tedious and expensive to create, however, as they require an extensive amount of repetitive work to generate by hand, both in model creation and collective placement. Subverting this unnecessarily time consuming process through automated and semi-automated methods of procedural generation is generally desirable.

A procedurally generated city model must take into account realistic layout constraints that would be present in a real-world city. Constraints include things like the roads, bodies of water, and the geometric space occupied by other existing buildings. Within the scope of the project, we aimed to procedurally create a reasonably realistic city that took as input from a user both building models and a planned map grid. The map grid is a text file that specifies the bounds and constraints of the city in question: how large and wide the final model should be, as well as any terrain constraints from a user's desired city layout. The building(s) provided by the user are accepted as .obj files and define the shape of base buildings with which to populate the grid.

2. Related Work

Work in procedural city generation spans a wide variety of implementations, each suited to its own particular set of goals and applications. When researching background information and searching for a starting point for this project, we looked into at three particular methods, illustrated as follows:

[1] *Model Synthesis: A General Procedural Modeling Algorithm* – Paul Merrell and Dinesh Manocha

Model Synthesis describes a method that procedurally generates complex 3D shapes. This algorithm, developed by Paul Merrell and Dinesh Manocha, creates a model given a singular input model and a series of articulated constraints. These constraints are applied to vertices of the model, and defined by Boolean expressions that dictate the behavior of the final, synthesized model. The constraints shape the final model in defined ways, restricting how the algorithm makes additions to the initial model in dimensions, connectivity, etc.

This implementation is presented as a sound method not only to generate models of singular objects, but also of complex buildings and cities. As such it was relevant to our initial planning of the project. The use of a user-defined input model was artistically and conceptually important to our own work, as was, initially, the idea of making modifications to individual buildings and the city as a whole through information and constraints surrounding the input model. The procedural methods presented in *Model Synthesis* have roots in CAD modeling (an appealing connection when considering applications for city generation) and are fairly comprehensive in synthesizing both convincing buildings and realistic, city-like spaces.

Further, we decided to implement the direct building editing algorithm as a stretch goal to our project. (This did not, unfortunately, come to fruition.)

[2] *Continuous Model Synthesis* – Paul Merrell and Dinesh Manocha

Continuous Model Synthesis illustrates further work by the authors of *Model Synthesis*. In this project, the two present an algorithm that takes 3D model input and creates a series of new models (altered in dimensions and texture; to a lesser degree of complexity than *Model Synthesis*) to quickly populate arbitrary space. This method uses similar ideas of algorithmically editing models based on existing constraints, but is tailored more to handle adjacency issues in placing the models, rather than try to be a catch-all procedural model editing algorithm (IE, generating all that is displayed onscreen from a singular model.)

This was one step closer to what we hoped to achieve with our project. Our goal was to fill a plane with models based on user input, with spatial constraints between that which was being placed as an important developmental aspect of our work process.

We were also attracted to the idea of making generative alterations to the building in size and texture; it would make for a more convincing and appealing landscape while still relieving the user-burden of inventing a huge number of models. It would also be somewhat trivial to implement in a basic sense. (A more complex version was, again, an unreached stretch goal.)

[3] *Procedural Modeling of Cities* – Yoav Parish and Pascal Müller

Neither of the previous references related much to terrain or map-like restrictions in generation. We sought out *Procedural Modeling of Cities* to fill this void.

Of the systems we researched, this was the most developed. Their program, developed and published in parts over a period of time, is called *CityEngine*. It creates roadway systems for fictional urban spaces based on population density and geographic factors. It also divides buildable space into lots, to the algorithmically populate with buildings.

This was the major inspiration for our grid class and implementation. Our own placement method borrows from the basic organizational concepts in *CityEngine*, in how it divides and articulates space. Another of our stretch goals was related to this as well: complex map generation upon which to build a city, based on input data and maps.

3. Work Process

When the project started out there were two major subsystems involved: procedural building alterations and grid filling. Building alteration was meant to make it such that there were not large amounts of entirely identical buildings on the grid. Grid filling was meant to fit as many buildings as possible into the given space. However, we couldn't just start on our major goals right away; first we would need to create the framework for the representing and rendering the grid and buildings.

This consisted of a grid class, a lot class, and a building class. The grid class represents the area that the city model is within. It is an m x n rectangle in the x-z plane. The grid contains all of the VBOs and generally serves as a way to collect and organise the information from buildings and individual lots. A lot is a 1x1 square on the grid which knows its own location in grid coordinates and its colour and status. The status is a marker that indicates whether the lot is empty, full, or containing some type of special feature. The building object contains the full geometric representation of itself, its own colour, and can access information about its own spatial footprint. Together these classes make up the bulk of the city creation part of the program.

The buildings are processed into our building class from .obj files parsed from user input. As it’s constructed, it runs through two arbitrary alteration functions – one for size, which implements a uniform scale on the dimensions of the original model, and one for color, which very simply assigns randomized RGB values. The building’s footprint is created after it has been scaled. This is accomplished through a class function that first finds the maximum dimensions of the building from the .obj information. This creates a fictional 3D rectangular solid, which is used to create the initial bounds of the bounding grid. The true space occupied (in bounding grid coordinates) is then created by checking against given, defined edges. Those spaces which contain edges are marked as “occupied”, as is the space between the established bounding edges. All else is marked as empty.

The goal of the grid-filling function was to be able to put as many buildings on the grid as possible. The first idea for how to do this was to attempt to place buildings randomly within the grid. While this would have eliminated a bias toward filling one end of the grid, at a certain point this would be too unlikely to place any buildings as the available spaces were all filled. We decided to abandon the idea for something more likely to guarantee success, though it would establish a bias in the order the grid gets filled in to a certain degree. The method chosen to do this was to use a space-filling curve. Peano curves and Hilbert curves were both considered, but the final choice was a simple curve that goes all the way down one row of the grid, and then jumps to the next row and traverses it in the opposite direction. This checks every square exactly as often as the other two should, but avoids issues pertaining to discretising the two curves, which are designed to be used in continuous spaces.

Once this was decided upon fully, building placement and intersection checking was the next part of the problem to tackle. The list of building objects is sorted by building size. The first step to this was just checking against the grid lot containing the lower left corner of the building's bounding box. This would be the whole solution for a 1x1 building, but was not complete and results in a lot of intersections when any building with a dimension larger than 1 is used. This was when we wrote a new data structure, similar to the bounding box, to keep track of the building's 2D bounding rectangle in the ground plane, and also a more informative footprint for the building. The footprint is designed so that a function call can be made to determine whether the building takes up space in a particular 1x1 square within the bounding grid. If you consider a U-, T-, or L-shaped building on the ground plane, these are some of the types of cases where the area within the bounding grid is not all filled by its corresponding building. Because of the inherent discretisation of the scene there is nothing lost in using a completely discretised bounding grid for this. Intersections are checked not only at the lower left corner, but in every square that it occupies when projected onto the ground plane.

The extension of this into the third dimension is just the extension of the grid into the third dimension, and has the advantage of allowing the nesting of buildings under each other, where with a 2D bounding grid buildings cannot be placed under sections of other buildings that do not touch the ground. <img />

If the grid does not contain as many buildings as the user would like and can fit more buildings, further building generation steps can be triggered to fill more of the grid.

4. Implementation

At this point the project is able to do the following:

Make alterations to the size and colour of a building.

Populate a grid with the models in the valid empty spaces in the grid.

It does two conceptually simple things, but how these are accomplished, and how are data structures are represented, is as follows:

4.1 Building System

The building system is comprised of two components, a bounding grid and the actual building representation.

4.1.1 Building Class

A building is created from a file written in a restricted subset of the .obj format. The constructor function reads in all of the vertices and then all of the faces from the file. During this process it also constructs the edges from the face information and the vertices. The building class also stores the colour of the building, but that is chosen randomly and is not based on the input file. The building knows its own size and contains a reference to the bounding grid that represents the space that the building is taking up, which is created when the building is being created.

4.1.2 Bounding Grid

There are two versions of the bounding grid, a two-dimensional version and three-dimensional version. The two-dimensional version is abstracted as a 2D array of booleans. The first step in the bounding grid's creation is resizing the array of booleans to the correct size, governed by the width and length of the building. All booleans are originally set to False. Next, for each edge in the building model, if part of that edge exists within a 1x1 square in object space, the boolean at the corresponding array index is set to True. This is how the building footprint is determined. In this case the height of a building is ignored and the building is treated as a flat object on the ground plane for the purpose of determining intersections.

The three dimensional case is like the two-dimensional case, but the building footprint is a stack of 2D boolean arrays, each one corresponding to a unit range of height values. This results in a data structure that can be checked at an index to see if any part of the building exists in the unit cube extending in the positive direction on all axes from that index.

A building's size, which is a relevant factor in deciding what order to place buildings in, is decided by how many boxes of the bounding grid a building takes up.

4.2 Lot Class

A lot is a 1x1 parcel of land that stores its own x-index and z-index in the overall grid structures, its status, and its colour. The status variable can refer to several states that a lot can be in. Examples of states are empty, full, roadway, water, park, and blocked. Empty lots contain nothing and can have buildings placed in them, while full lots already have buildings in them. Any other type of lot is blocked for a separate terrain-based reason. A lot's colour is determined by its status.

4.3 Grid Class

The layout grid, referred to henceforth simply as the grid, is the object that brings everything together. The grid contains a vector of building objects and one of lot objects, as well as all of the VBOs and finalised rendering geometry. It also contains the driver functions for almost all of what the program is doing. It handles the generation and placement of buildings as well as pushing all geometry down the OpenGL pipeline for rendering.

The building placement function takes a two-dimensional space-filling curve and populates the grid with building objects by going along this curve. It also traverses through the list of buildings that have been loaded into the program and then modified. This list was originally meant to be represented with a priority queue, but this was causing irregular behaviour and was replaced by a simple vector that was sorted with the standard library sort function. The VBO-filling functions take data that has been organised in some convenient format in some other class object and push it down the graphics pipeline.

4.4 Argument Parser

The argument parser is a simple class that stores information related to single execution instances of the program, such as the specific building and grid files being loaded and the limitations on the size of the building. One instance of this object is created per execution of the program, and a reference to it is passed around to the buildings and grid where the values stored in this need to be accessed.

4.5 Canvas Object

The last part of the project is a canvas object, which handles mostly the GL and GLFW environment setup, and configures and links variables to the fragment shader. It is as such responsible for handling mouse and keyboard event callbacks. It also defines the matrix transforms between the various different views of the scene. It serves as an abstraction for the final display of the scene.

5. Results and Conclusion

When we began our project our primary goals were to be able to make superficial alterations to an input model, and to populate a simple confined space with a set of models. Further than that, we wanted to generate non-trivial layouts for confined spaces based on provided input constraints, and to fill these constrained cities with our input models. Our stretch goals were to populate more complex spaces with buildings to take into account more constraints and eventually create cities based on street layouts from real world map data.

We were able to achieve most of these goals to a certain extent. Filling confined spaces and filling constrained confined spaces are fully realised goals in our system. We were able to develop a system of input file that allowed a reasonable degree of expressiveness in terms of constraint specification, and bulidings can be placed well in grids generated with these constraints. Our building alteration system, though it can be said to be sufficient based on the wording above for the application we have developed, is not as complex as we would have liked it to be. Particularly, trouble with defining procedural textures in the fragment shader is preventing progress in this direction at this point.

Although it cannot be said that we have reached our original stretch goals, the extension of the program into the third dimension is something we have achieved. This is important in that it can allow for structures that go over and under each other, and over restricted terrain. It is a framework that allows for structures such as bridges to exist in our models, and is a step towards potential development in the direction of attaching annexes to buildings after their initial creation.

One notable bug in our implementation is that input models cannot be made with a negative coordinate on any axis. This will cause problems in resizing the model. This could be solved while the model is being read in, but this bug fix has not been implemented yet.

Though not every desired features has been fully implemented at this time, the program works well for what has been implemented and a there is a clear direction for future work on this project.

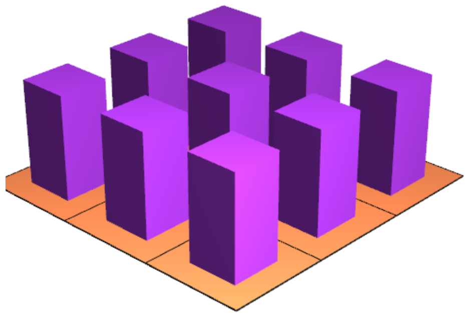
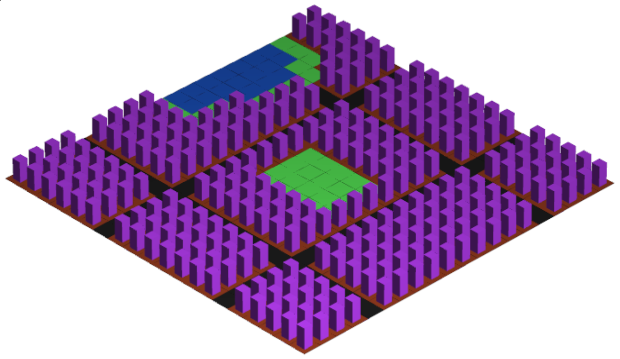
	

Figure 1: Our initial grid population, with uniform buildings from input.	*Figure 2: Grid population with terrain restrictions. Green represents grassy "park" spaces, blue represents water, and black represents roads. None are suitable for development, and are marked and treated as such.*

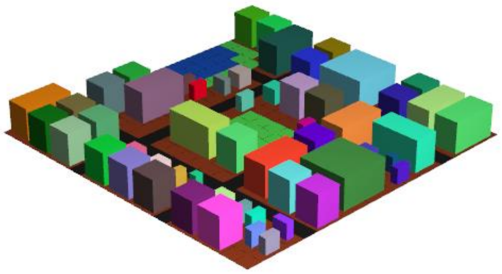
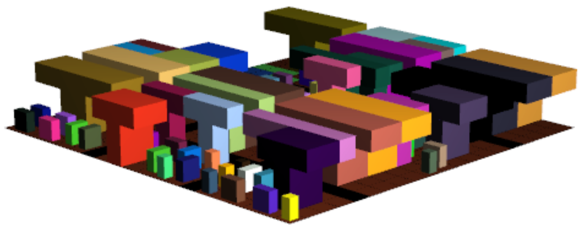

	

Figure 3: Grid population with terrain restrictions and arbitrarily resized and recolored buildings.	*Figure 4: Grid population with 2D spatial restrictions and multiple building input. Note how the smaller buildings are not nested in 3D (beneath the "T" models.)*

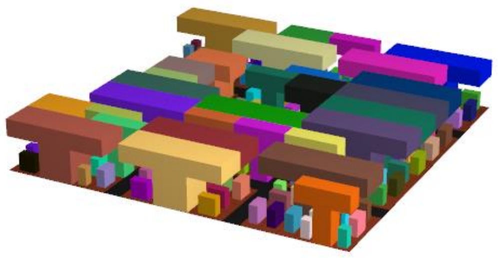
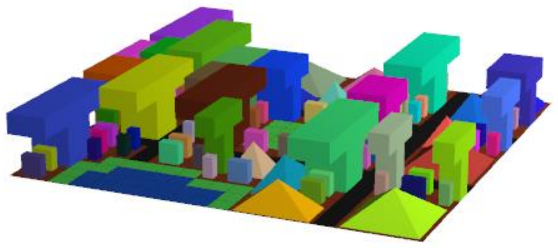
	

Figure 5: A city as (4) but which incorporates 3D spatial relationships and allows for nesting beneath structures.	*Figure 6: Our final output, a colorful city in "Weirdo Land" with an assortment of 3D-nested, altered buildings.*

6. References

[1] P. Merrell and D. Manocha. Model Synthesis: A General Procedural Modeling Algorithm. IEEE Transactions on Visualization and Computer Graphics, 2010.

[2] P. Merrell and D. Manocha. Continuous Model Synthesis. ACM Transactions on Graphics, 2008.

[3] Y. Parish and P. Müller. Procedural Modeling of Cities. ACM Transactions on Graphics, 2001

FOOTNOTE – work division and time for completion:

Michael worked on the grid class implementation, while Rachel worked on the building elements. These two aspects were combined around midway through the project, and from there we worked together. Approx. 50 hours were spent on creating, debugging, and testing the project.