4-25-2005

Generation of office buildings in large scale virtual worlds

Michael McBryde
Trinity University

Generation of Office Buildings in Large Scale Virtual Worlds

Michael John McBryde

A departmental senior thesis submitted to the
Department of Computer Science at Trinity University
in partial fulfillment of the requirements for Graduation.

April 20, 2005

Thesis Advisor

Department Chair

Associate Vice President
for
Academic Affairs

Generation of Office Buildings in Large Scale Virtual Worlds

Michael John McBryde

Abstract

Virtual worlds are used in many different areas, from military training simulations to massive multiplayer online role-playing games. In the past, the sizes of these worlds was limited by the power of the computers that ran them as well as the man-hours needed to draw them. However, as computers have become more powerful, the limiting factor has become the man-hours needed to manually draw every object in such a world. So there is now a need for large scale, traversable, dynamic, algorithmically generated virtual worlds. For these worlds to be realistic, cities need to be generated, and for these cities to be realistic, they must have commercial office buildings (skyscrapers, office parks, etc.). Previous research in this area has been solely on generating the outsides of commercial buildings, with no focus on the inside features of the buildings. This research aims to generate both the insides and the outsides of commercial office buildings, with the dual goals of realism and usability.

Generation of Office Buildings in Large Scale Virtual Worlds

Michael John McBryde

Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank my family for the love and support they have given me for many years, Dr. Maurice Eggen and Dr. Mark Lewis for their help and advice, and my friends for ensuring that I got my weekly dose of beer and breakfast tacos (though, thankfully, not both at once).

Generation of Office Buildings in Large Scale Virtual Worlds

Chapter 1

Introduction

Virtual worlds are used in many different areas, from military training simulations to massive multiplayer online role-playing games. In the past, the sizes and level of detail of these worlds were limited by the power of the computers that ran them as well as the man-hours needed to draw them. However, as computers have become more powerful, the limiting factor has become the man-hours needed to manually draw every object in such a world.

So there is now a need for large scale, traversable, dynamic, algorithmically generated virtual worlds. For these worlds to be realistic, cities need to be generated. For these cities to be realistic, they must have some way to generate traversable office buildings (skyscrapers, office parks, etc.).

This research aims to generate both the insides and the outsides of commercial office buildings, with the dual goals of realism and usability.

1.1 Previous research

There has been some research on the algorithmic generation of objects in a virtual world but less research on the algorithmic generation of buildings. What previous research there is in this area has been solely on generating the outsides of commercial buildings, with no focus on the inside features of the buildings. Most of the focus has been on city-wide modeling or on ways to speed up the rendering process of large cities.

However, some of the previous research, while not specifically on the topic of algorithmically generating traversable office buildings, has been of some use. A paper on procedural generation of cities by Greuter, et al [1] has been of great use in a part of this research which deals with generating the outside walls of large office buildings (skyscrapers, etc; see Section 3.5). Also, the research into traversable residential units by Martin [5] has been used as a starting point for some of the general ideas presented in this thesis.

1.2 Goals for this research

The primary aim for this research is to implement an algorithm which can generate commercial buildings for a virtual world. This algorithm should be measured according to three criteria: believability, usability, and extensibility.

A believable generation algorithm is paramount; if the office buildings generated by this algorithm are not believable, then the algorithm is useless. Believability has several different facets; each building generated by the algorithm must by itself be believable, and there must be some sort of variation between the buildings generated. These buildings are not intended for use by themselves, they are intended for use in large quantities, to be used for city and world-level generation. As such they must be as different from each other as commercial buildings are in real life.

After believability comes usability. This means that the algorithm used to generate the buildings has to be able to generate them fast enough to work in real time. This could mean that thousands of buildings need to be generated at approximately the same time for large urban areas.

Lastly, this algorithm should be extensible. There are many different types of office buildings; it is not possible to generate every different kind of office building. In different situations, different building types might be called for, so the algorithm created must be able to be extended to accommodate those different types of buildings.

1.3 Commercial real estate classifications

When this project was started, I began with a fair amount of knowledge about the design of modern office buildings. For one summer (2004) I had worked for a commercial real estate firm in Oklahoma City gathering market data. My job was simply to walk through every multi-tenant office building in Oklahoma City and gather information: height of ceilings, major tenants, quality of the buildings, fire sprinklers, and several other pieces of data. Through this job I got a first hand view of over 140 office buildings of all different types. The information that I gathered forms the basis for the building designs throughout this thesis.

I mentioned above that I saw buildings of all different types. In the office I worked for, buildings are classified into three different types based on size, affluence, amenities, quality, age, and many other qualities that are nearly impossible to quantify. The classifications are labeled plainly: there are A-Type, B-Type, and C-Type buildings, with A-Type buildings being the highest quality and largest and C-Type being the lowest quality and smallest.

This classification scheme merits some discussion; though it is not perfectly suited to

the needs of this research, it is a useful basis of knowledge.

C-type buildings are the low end of office buildings. They are usually less than 3 stories tall and of very simple design. They are also usually a bit run-down, they are usually older buildings, and they are usually in the poorer sections of the city. Not all of these traits have to be evident: any one of them is enough to make a building a C-type by itself. Often, however, more than one of these traits is evident.

A-Type buildings are at the opposite end of the spectrum. They are the skyscrapers in a city, as well as some of the smaller (but often still more than 10 stories high) office buildings. They are often modern and always well maintained, with a large number of amenities for the tenants and the visitors. They are usually designed with a fairly meticulous attention to detail, and often serve as the architects' playground. If an office building could be an A-Type but is more than just a little bit deficient in any area, it falls down to a B-Type building.

B-Type buildings, as might be imagined, are in the middle. These buildings are a mix of all the decent-sized, fairly well maintained, fairly modern and fairly good quality buildings. These are the buildings that make up most office parks, as well as a good number of the (not so flashy) buildings in downtown areas.

Unfortunately, this classification scheme has several drawbacks. First and foremost, this is a classification scheme for preexisting buildings. It is not a scheme for either designing or building these office buildings. Secondly, this classification scheme is more than a bit fuzzy, especially where a building might be one type or the other. There is no absolute quantification for any of these qualities, and many of these qualities vary as to importance depending on the building in question. This classification scheme boils down to simply a guide for gut instinct. But gut instinct is useful, and later on we will use this classification scheme as a basis for parts of the algorithm and adapt this terminology to our purposes.

1.4 The structure of this thesis

Chapter 2 of this thesis will deal with the problems of generating office buildings, define some terminology that will be used throughout the remainder of this thesis, as well as propose and describe a general algorithm for office building generation. Chapters 3, 4, and 5 will describe specific algorithms for generating several parts of an office building. Chapter 6 will make several conclusions about this research as well as suggest areas for future research.

1.5 Assumptions

It needs to be noted that the buildings which are generated by this algorithm are based on current (early 21^{st} Century) office buildings. While the general algorithm discussed in Chapter 2 can generate any office building, the specific algorithms discussed in the later chapters are modeled entirely after current office buildings.

1.6 A note on the programming language

All of the implementation for this project has been written in Java 1.4.2. This choice was made for several reasons. The first is the author's familiarity with the language. The second reason is that the virtual world project at Trinity University has been implemented in Java as well, and this implementation is designed (in part) to work with that preexisting virtual world. A selected portion of the code for this project can be found in Appendix A.

Chapter 2

The General Algorithm

2.1 Problems faced

There were several problems that I faced when beginning this research, chief among them how to accurately represent an office building as a 3-dimensional object in space, as well as the order in which the parts of the building should be generated.

An office building can be split up into several codependent units: the offices themselves are obviously the most important parts, but there are also hallways, restrooms, elevator shafts, stairways, lobbies and atria, as well as the outside walls of the building. There are also, depending on the quality of the building, sometimes secondary spaces on the main floor of the building: cafeterias, restaurants, shops and the like.

If the relationships between all of these parts are mapped out, a very interesting thing occurs: every piece of the building is codependent on something else. For example every piece of the building relies on the outside shape of the building, but hallways depend on lobbies/atria and elevators, elevators depend on lobbies/atria as well as hallways, restrooms depend on hallways and elevators (the restrooms on a floor are almost always next to the

elevators), etc.

The most pressing problem which comes from this is that elevators and hallways are dependent on each other; elevators are useless if they do not connect to a hallway, and hallways are useless if they aren't connected to the other floors via an elevator.

There is also the question of representation. Buildings can be represented in many ways, and the choice between those ways is very important. They can be represented very realistically, as a series of surfaces in three dimensions, with those surfaces representing the actual appearance of the building when rendered. They also can be represented very abstractly, as a series of graphs, and only very late in the process would those graphs be turned into something easily renderable. There are also many different ways which lie in the middle of these two options. All of these options have different advantages and drawbacks, and the choice that is made must take them into account.

So these were the main problems: the order in which the parts of the building should be generated and how the building should be represented.

2.2 Concepts and initial ideas

One thing was clear from the beginning of this process: the outsides of the buildings should be generated first. Every other feature of the building is spatially dependent on the layout of the outside walls of the building.

One initial idea was to first generate the outside of the building, then generate any lobbies/atria. From there the algorithm would decide how many floors were in the building then make up graphs of the hallways, offices, restrooms, etc. Then the algorithm would blow up the insides of the building until all of the hallways and offices and such reached the outsides of the building, somewhat like a balloon being blown up inside a box.

This idea, while very interesting, was abandoned fairly early on. While it would be very interesting to try, I had the feeling that it could be a very time-consuming process, and one of the two main goals of the project was to make it real-time. Also, I wanted to keep it simple. Doing this for the first time, I had very little idea what the real problems would be on the ground; a complicated overall plan with no idea of the problems that could be faced when actually coding this project seemed like a bad idea.

One important thing to note is that from the beginning the design of the general algorithm used an outside-in approach, generating the outside walls of the building before filling in the insides. This is opposed to an inside-out approach which would fill up the insides of the building then create the outsides of the building around them. The outside-in approach was judged to be simpler than the inside-out approach; however, after working on this project for several months it became apparent that such an inside-out approach could be a comparable solution to the problem (see Future research, Section 6.4).

2.3 Essential concepts of the general algorithm

The essential question that drove the beginning of this project was: What is a building? The answer, essentially, is that a building is made up of outside walls and a roof, and almost always some sort of divisions inside; a building, and especially an office building, almost always is divided up into floors.

So I started with the basic idea of a pancake design. In other words, that a building is nothing more than a series of floors stacked on top of each other. If we can generate the outside of the building, and store it by floors, we can then deal with each floor separately, add hallways or remove them, make executive offices (offices that take up entire floors, with no or very few hallways), make lobbies and atria, etc. This is a middle of the road solution

to the problem of representation. We are dealing with real points in space, but they have been split up into floors and we can deal with each floor separately. We are not dealing with absolute representations, nor are we dealing with completely abstract representations. There are some drawbacks to this approach, however, which are covered later in the chapter in Section 2.6.

The connective elements of a building are, aside from the outside walls, the most important parts. The hallways and elevators inside a building are the way that people access the various parts of that building, and the space inside the building is useless if there is no access to it. Further, hallways and elevators, in some sense, form the skeleton of the building. Everything else in the building comes off of the hallways, and the hallways and elevators form the base structure of a building. In office buildings in particular, office space is built around the hallways, rather than the other way around.

At this point, we run into the problem of generation. It is something like the chicken and the egg. What comes first, hallways or elevators? Each is dependent on the other.

To solve this, I exploited a simple observation: in almost every office building, if the outside walls are the same, the inside hallways are constant. In other words, every floor looks exactly like every other floor, with occasionally a little bit of modification; every once in a while there will be half-hallways, or none at all, to make one office suite larger than normal. But the main shape of the hallways is the same for every floor of a building, as long as the walls of the building are the same. It then becomes very easy to make an elevator shaft: there will always be some hallways that hit the same places, and those can be connected.

So the overall idea for the algorithm focuses on a few concepts: pancake representation of office buildings, outside-in generation, the importance of connectivity, and to solve the problem of generation order the observation that similar walls make similar floor plans.

2.4 Differences between A, B, and C-Type buildings in this research

Before we continue with the implementation of the general algorithm, some terminology needs to be defined. As has been noted in the introduction, in commercial real estate there is a difference between A, B, and C-Type buildings. I also have noted that these classifications are insufficient for the needs of a formal algorithm. Now, however, we have enough of a formal algorithm defined to pin down exactly what is meant by A, B, and C-Type buildings.

In addition to the three types above, one more classification has been added: the Courtyard building. Courtyard buildings are a type of building that, obviously, are buildings which surround a central courtyard. They are fairly common, and occur in all three commercial classifications. However, their structure is different enough from the rest of the buildings in the commercial classes that they deserve individual attention.

One thing worth noting is that, unlike the commercial classification scheme, these types are not meant to represent all of the buildings that can be made. Rather, they are the subset of buildings that this research focuses on. Also, this classification scheme focuses on the characteristics of the buildings that affect this research. So for instance, the A-Type building definition mentions hallways, but does not mention anything regarding building materials or the like, as this research does not focus on generating textures for rendering.

Whenever “A-Type”, “B-Type”, etc. are mentioned in the remainder of the thesis, these definitions are what is being referred to, unless otherwise noted.

2.4.1 C-Type buildings

C-Type buildings, as in the commercial classification, are very simple. They are, first of all, rectilinear and rectangular, meaning that the building is composed of straight lines, and all of the lines are either parallel or perpendicular to each other. This includes both the outside walls of the building and the inside hallways. Second, every floor is the same. Each floor has the same outside walls, and each floor has the same inside halls. Third, the outside walls of every C-Type building compose a single rectangle.

2.4.2 B-Type buildings

B-Type buildings are a bit more complex. They are, like C-Type buildings, also rectilinear and rectangular. Second, and similar to C-Type buildings, every floor is the same; each floor has the same outside walls and inside halls. However, unlike C-Type buildings, the walls of B-Type buildings are composed of two rectangles added together. In other words, B-Type buildings have floor plans shaped roughly like an L or a T.

2.4.3 A-Type buildings

A-Type buildings are different from C-Type and B-Type buildings in nearly every way. First, they are rectilinear (composed entirely of straight lines), but they are not necessarily rectangular. Second, not every floor has to be the same. Think of skyscrapers: some are composed of a single shape which continues to the top of the building, but more often they are composed of several different shapes, with the building narrowing as it gets higher. A-Type buildings are composed of one or more shapes, with different floors made out of (possibly) different shapes. These shapes can be rectangles, but are more often triangles, octagons, etc.

2.4.4 Courtyard buildings

There is a fourth class of buildings, usually lumped in with B-Type buildings for commercial classification purposes, but totally different for generation purposes: the Courtyard building. Similar to B- and C-Type buildings it also has rectangular and rectilinear walls and halls, and every floor in this type of building is the same. However, the walls and the halls of these structures are special. Their walls are composed of 4 rectangles added together to form a single shape which completely surrounds an inner courtyard, or 3 shapes added together to form a U shape which almost surrounds a central courtyard. These buildings usually have only one hallway, a ring that goes around the whole building. I had originally treated them as a subset of the B-Type buildings, but as far as general construction goes, they are different enough to merit their own class.

2.5 Implementation of the general algorithm

The base idea for the algorithm is to generate the walls of the building, then generate the connective elements inside the building, then do any other generation that the building needs (adding offices, elevators, restrooms, etc).

Originally, the idea was to split up the algorithm into several different pieces, depending on the type of building to be constructed. In other words, there would be one class which generated every C-Type building, one class that would generate every B-Type building, etc. This setup was useful in the early stages of the project, when the similarities between the buildings were unclear.

Later in the project, however, it became obvious that, for instance, the hallway generation for B- and C-Type buildings would be the same, but A-Type hallway generation would be different. Similarly, every building generation algorithm would have the same overall

structure, with only the details of implementation differing. So it was decided to make a main building generation class, and to use a series of generators to create the differences in building generation.

The structure of the building generation algorithm is very simple: the constructor is passed in several variables, including what hallway generator and wall generator this particular building will be using to construct itself. It also has the maximum footprint size passed into it, so that the building knows how much land it can take up, as well as the number of floors it will be and two optional parameters, the preferred width of each office and the width of a hall. It takes these parameters, generates the walls of the building using the wall generator, then uses those walls to generate the hallways in each floor using the hallway generator. After it has this information, it can do whatever it wants with regard to adding elevators, individual offices, etc. These things might vary, or might not.

(Due to time constraints, the implementation for adding these individual offices, lobbies, etc. have not been written; however, they are fairly simple and do not vary from building type to building type.)

The advantage of arranging the algorithm in this fashion is that the different parts of the algorithm can be modified and swapped at leisure, and additional hallway generators and wall generators can be written extraordinarily easily. This gives the algorithm a large amount of extensibility and usability, and satisfies one of the goals of the research; that is, for the algorithm to be extensible past the scope of this research.

2.6 Advantages and disadvantages of this design

There are some drawbacks to the overall design, however. First, it is highly dependent on spatial relationships. This makes it a bit difficult to generate buildings from graphs.

Second, it is fundamentally designed to generate buildings from the outside in, so several (possibly very interesting) building generation ideas are impossible to implement. Also, the reliance on the pancake design makes some kinds of structures very difficult to implement; those with some kind of ramps are a good example. Finally, the separation between the wall generation and the hall generation makes it even harder to generate buildings from graphs.

But this design does have quite a lot of potential. Nearly every possible building shape can be generated in this fashion, though some are very difficult. For instance, making a building that looks like the Sydney Opera House would be nearly impossible, due to the curves and the odd shapes (most of the work, however, would be in the rendering, not the floor plan generation). But, fundamentally, it is possible. This algorithm can also create buildings that would be impossible to construct in the real world, such as upside-down cones, etc.

Chapter 3

Wall Generation

The outside appearances of office buildings have the most effect on the believability of the office buildings themselves, as well as on the believability of the virtual city in which they are placed.

Believability for each office building means that the building generated matches some idea of what is and is not an office building. Believability in the case of multiple office buildings means variability. The outsides of office buildings never look the same, even when they are in the same commercial classification. So the wall generation algorithms should rarely create the same building twice.

An observer in a virtual city will hopefully spend some time walking through the buildings and seeing what is inside all of them, but they will certainly spend a good amount of time looking at the city as a whole. If every building looks the same, then the city will look fake. So the outsides of the office buildings must have a certain amount of variability to mimic the variations found in a real city.

For this research I wrote four wall generation algorithms; the general building generation algorithm does not distinguish between them, so they can be swapped out at will. We

will discuss the overall goals for wall generation algorithms, then discuss the algorithms themselves, with an evaluation on the realism of each type of building at the end of each section. Then we will evaluate the speeds of the various wall generation algorithms.

3.1 Goals for wall generation algorithms

First and foremost, the buildings must look realistic. For C-Type buildings this isn't very hard; they are not much more than boxes with doors and windows. As the wall generation algorithm does not concern itself with either doors or windows, all that a C-type building must look like is a simple box, with varying dimensions. Courtyard buildings are similar to C-Type buildings in this regard. B-Type buildings are slightly more complex, containing two rectangles instead of one, but they are similarly simple.

A-Type buildings, on the other hand, are the showpieces of an urban area; they are the ones dominating the downtown skyline of the usual urban area. Thus, they are the focus of attention for any viewer first seeing a virtual city, and if they are not realistic, even if every single other building in the city is copied straight from a real city, the city will not look genuine.

The variation between A-Type buildings is an essential part of believability. If any A-Type building looks the same as any other building, it will be noticed and the buildings will look fake. Variation between B-Type buildings is still important, but less so. The variations between C-Type buildings and Courtyard buildings are almost unimportant, as they all look extraordinarily similar in real life.

As far as speed goes, however, the priorities are reversed: there are far more B-Type, C-Type, and Courtyard buildings than there are A-Type buildings, so these first three generation algorithms must be very fast, while A-Type wall generation can be a bit slower.

This is not to say that any of these generation algorithms can be slow; to be usable they must still be real-time.

3.2 C-Type wall generation

There are several things to note about C-Type buildings. First of all, they are extraordinarily simple. Because their commercial classification C-Type counterparts are designed to be the cheapest to build and the cheapest to rent out, they are, to put it mildly, not architectural masterpieces. They are also very popular, especially in cities like Dallas or San Antonio (I would guess because the rapid suburban growth makes it desirable to build cheap, simple buildings on the outskirts of a city). They are much more prevalent than A-Type buildings, and approximately as prevalent as B-Type buildings. Thus C-Type buildings need to be, above all, easy to generate.

3.2.1 The algorithm

So these are the requirements for the algorithm: it must be simple, it must be fast, it must be rectilinear and rectangular, and it must have at least some variation in the algorithm.

The algorithm is passed three parameters: two of them (maxX and maxY) represent the maximum space that the footprint of the building can take up. A third variable (floorCount) represents the number of floors that this building will have. The algorithm will return an array of floors, with each floor represented by an array of line segments representing walls.

For this generation algorithm the best solution is the simple solution. There is no need here for a complicated design.

If a building is going to be composed of only one rectangle, then there are three options for the shape of that rectangle: its length will be greater than its width, its length will be

less than its width, or the length and width will be equal. So the algorithm uses a random number to choose which one of these three cases is to be used. In each case the width or length (or both) of the building is maximized and the other side is given a random length from one half of its maximum length to its full length.

The variation in C-Type building generation is accomplished in three different ways. First, the algorithm depends on maxX and maxY for the final dimensions of the building. These will vary depending on the plot of land that the city generation algorithm wants to cover. Secondly, the algorithm randomly chooses from three different footprints, and uses those to generate the buildings. Third, in two of the three footprints, one dimension of the building is randomly shrunk. Between all three of these factors, there is enough randomness to ensure variation between C-Type buildings.

Figure 3.1 shows several different results from several wall generations, all of them with equal input parameters.

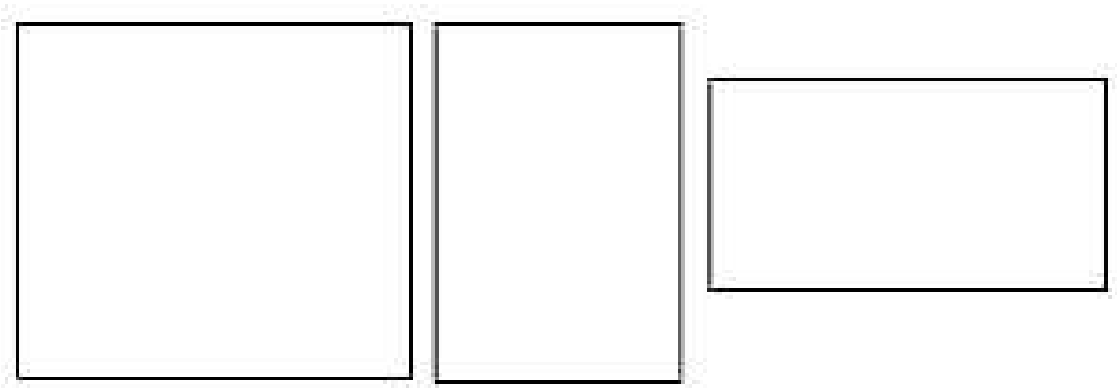

Figure 3.1: Floor plans for several C-Type wall generations

3.2.2 Evaluation of appearance

C-Type buildings (in the real world) are fairly uniform. Thus the C-Type buildings generated by this algorithm, simple as they are, are nevertheless fairly similar to the real world.

There is not a lot of variability in these buildings, but variability is not needed. There are three different options for building type, and in two of those building types there is a random factor which affects the size of the buildings. With the added variation of the input variables, there is enough variability for our purposes.

3.3 B-Type wall generation

B-Type wall generation is similar to C-Type, but with the fundamental difference that B-Type buildings can be composed of more than one rectangle. This makes the case-by-case solution for the C-Type buildings useless; there is too much variability to hard-code each possible building shape. This is a good thing, as it creates a large amount of variability in the building designs.

3.3.1 The algorithm

The algorithm is passed three variables: the maximum x-size and the maximum y-size of the area that the building can cover (maxX and maxY), as well as the number of floors desired (floorCount). The algorithm will return an array of floors, with each floor being represented by an array of line segments.

Essentially the algorithm places one rectangle into the maxX/maxY space, then adds a second rectangle onto the longer side of the first rectangle, to make an L or T shape. That same shape is then used for every floor. B-type buildings are uniform all the way up, with no variation between floors.

The first rectangle is dropped so that it is flush with the top and right sides of the space, with the top and bottom sides having lengths of half of maxX. The right and left sides of the rectangle are random, equal lengths between half of maxY and maxY. Though this doesn't ensure that the right and left sides of the rectangle are longer than the top and bottom, it does make it more likely.

At this point the second rectangle is dropped, with the left side of this rectangle flush with the right side of the first rectangle. There are three rough possibilities for the connections that these rectangles can make: the tops of the rectangles are flush, the bottoms are flush (both of these make an L shaped building), or neither the tops nor the bottoms of the rectangles are flush (this makes a T shaped building).

The height (top to bottom) of the second rectangle is always equal to half of the height of the first rectangle (though this isn't necessary, it does make the buildings look a bit better), and the width (left to right) is a random number between half of maxX and maxX.

After the two rectangles have been dropped, the building is randomly rotated 90, 180, or 270 degrees or remains the same. Then the building is scaled to ensure that the building has remained inside of the bounds maxX and maxY.

Note that there are four chances for the building to be altered inside the algorithm, as well as the three parameters sent into the algorithm. Both the height of the first rectangle and the width of the second rectangle are randomly determined, the second rectangle is randomly positioned with regard to the first rectangle, and the whole building is, at the end, rotated in one of four different ways. The size of the building is determined by maxX and maxY, and the number of floors is determined by floorCount, all three of which can be randomly determined outside of the algorithm.

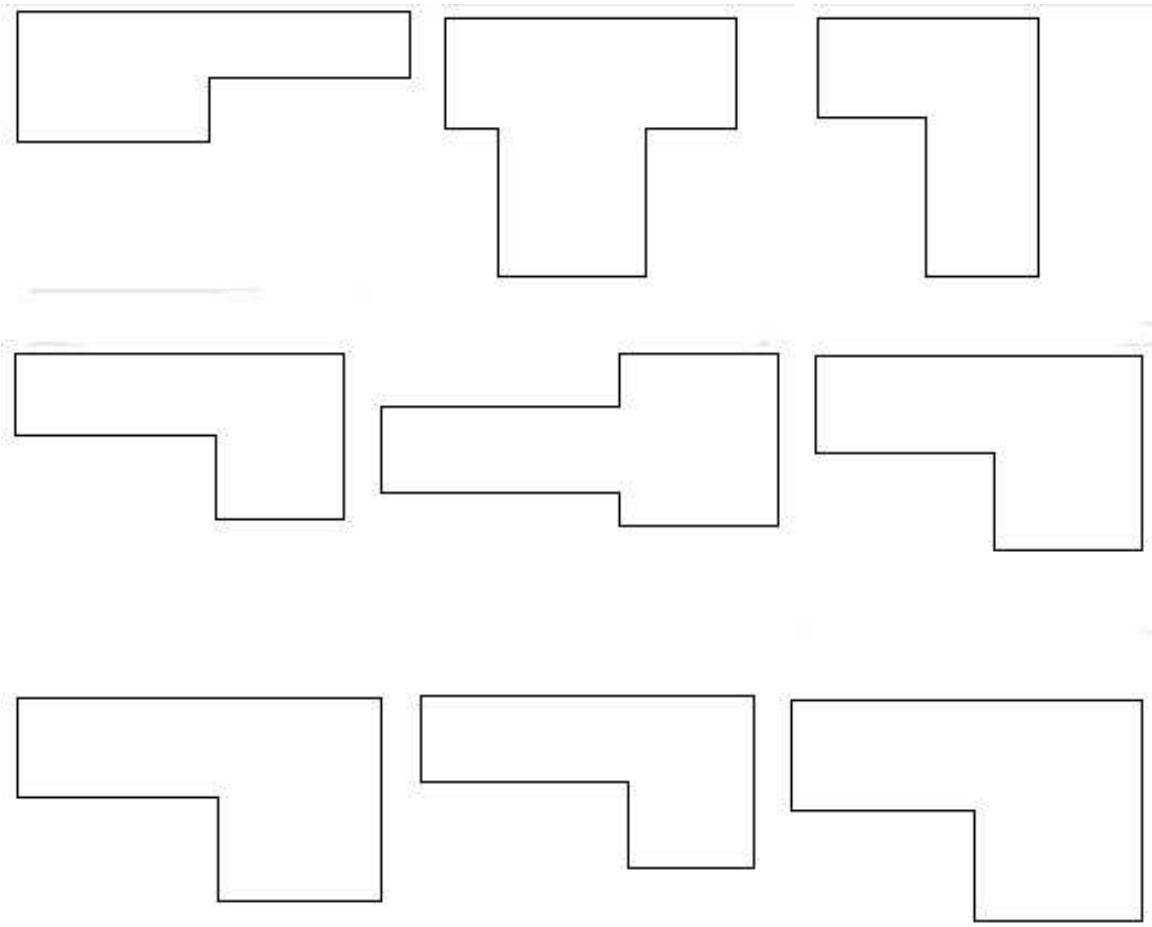

Figure 3.2: Floor plans of several B-Type wall generations

3.3.2 Evaluation of appearance

This algorithm fairly accurately recreates a large number of B-Type buildings. Many of the mid-size office buildings in a given urban area look something like the buildings generated by this algorithm. Of course, many of the architectural details would need to be fleshed out by the rendering process, but this would not be very difficult.

There is quite a bit of variation inside the algorithm. 12 different building types (three options for the rectangles to connect and four rotations) can be randomly created, and the actual dimensions of each of these buildings vary quite a bit, as seen in Figure 3.2. There is a finite number of possible buildings, but that number increases with the size of the overall floor plans.

3.4 Courtyard buildings

Courtyard buildings are interesting. They are a class of buildings that do not exist in the commercial real estate classification; they are usually lumped in with other classes of buildings. However, they are structurally quite different from the other styles of buildings that have been generated. These buildings are designed to surround a central courtyard, and in addition are very narrow: they usually have only one main hallway through the center of the building, with offices facing the courtyard and offices facing the outside.

3.4.1 The algorithm

Like all of the other generation algorithms, this one takes in three parameters as inputs, two variables which determine the maximum X/Y footprint of the building (maxX and maxY), as well as one variable which determines the number of floors in the building (floorCount). The algorithm will return an array of floors, with each floor represented by an array of line

segments.

The algorithm itself is the simplest of the four wall generation algorithms discussed here. There are five footprints: a building which has four sides and completely surrounds a courtyard and four buildings which have three sides and leave one side of the courtyard open (U-shaped buildings). In this generation algorithm the first building has a 50 percent probability of being chosen, with each of the other four types having 12.5 percent probabilities. Because these types of buildings are distinguished by their narrow width, they all are only two office widths wide, plus space for a hallway. The widths and heights of the individual buildings are determined by the maxX and maxY provided, not by any random generator inside the algorithm. In this sense, the courtyard buildings are the least variable buildings that will be discussed.

3.4.2 Evaluation of appearance

As previously mentioned, the buildings generated by this algorithm (see Figure 3.3) are very regular. Indeed, the only variation inside the algorithm is which of five footprints will be decided. Most of the variation depends on the maxX, maxY and floorCount passed into the algorithm. This is acceptable, and indeed what was aimed for at the beginning. In their simplicity, these buildings are very predictable, so this generation algorithm produces fairly believable buildings.

3.5 A-Type wall generation

A-Type buildings are the most complex building designs. In the real world, this class of building encompasses all of the very large, very well-designed, and, for lack of a better way to put it, all of the very pretty office buildings. Thus, these buildings are where the

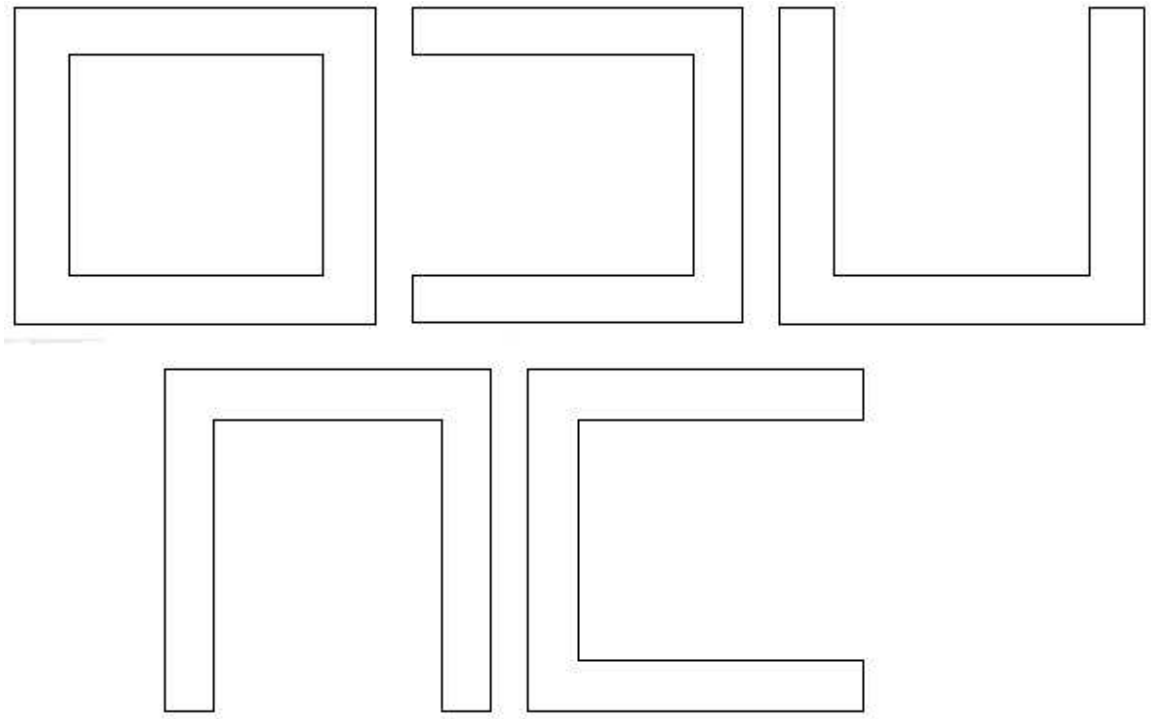

Figure 3.3: Floor plans for several Courtyard wall generations

architects go wild, and they are the largest and most variable of buildings.

For this generation algorithm I only looked at a small slice of A-Type buildings. There is a certain type of A-Type buildings that are built of conglomerations of shapes, with certain shapes being added to the building as it gets closer to the ground. This is the type of building that was generated in masse by Greuter, et al [1]. Figure 3.4, created by Greuter, is a pictorial representation of the "floor plan extrusion" algorithm he uses to create the buildings. The algorithm I wrote follows the concept of his algorithm, though certainly not to the letter, as our aims are very different.

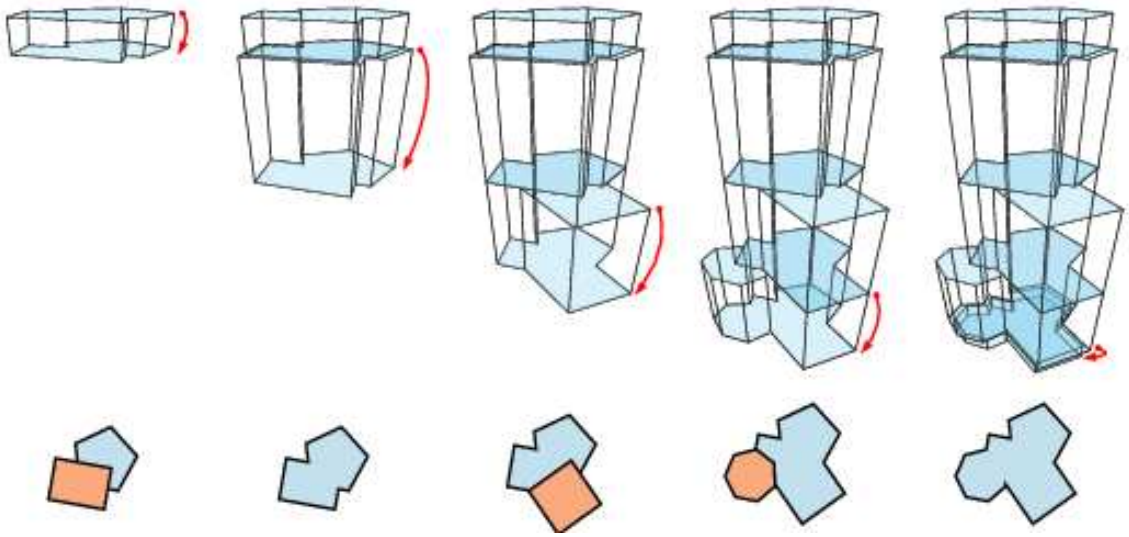

Figure 3.4: Diagram displaying the algorithm used by Greuter, et al [1] to generate buildings. The algorithm starts with a single shape, then adds one shape and extrudes down for a number of levels; it then adds another shape and repeats, generating the outside shell of an office building.

3.5.1 The algorithm

The algorithm, like all of the other algorithms, takes in three parameters, two representing the maximum size that the floor plan of the building can take up (maxX and maxY), and one represents the number of floors that the building will have (floorCount). The generation algorithm returns an array of floors, each floor being represented by an array of line segments.

The idea of this algorithm, while not as simple as the previous algorithms, is still fairly simple. The concept for this, as stated before, comes from Greuter, et al [1].

First, the algorithm decides how many shapes will be added on to each other to make the building. This number is a random number between one and the number of floors divided by eight. Eight is not a necessary number, but it is a reasonable approximation of how the complexity of these buildings should increase with their height.

Next the algorithm gets the first shape, and uses that shape for the walls of the top floor of the building. This shape can be anything, though for simplicity's sake they have been limited to simple convex polygons (rectangles, triangles, diamonds, octagons, etc). The shape is scaled by two random numbers for the X and Y coordinates and moved to a random place inside the bounds of maxX and maxY.

Then the algorithm copies that floor to the floors underneath it, ending at either the bottom of the building or approximately eight floors down (the actual number of floors is a random number of floors between seven and nine).

Then the algorithm repeats itself, adding more shapes to the building as it extrudes downwards, until it reaches the bottom of the building.

Adding more shapes to the building is the crux of the matter, as it is rather difficult to determine which lines in the new shape and which lines of the old shape should become

part of the new walls of the building. The details are not worth going into here, but the code can be found in Appendix A.

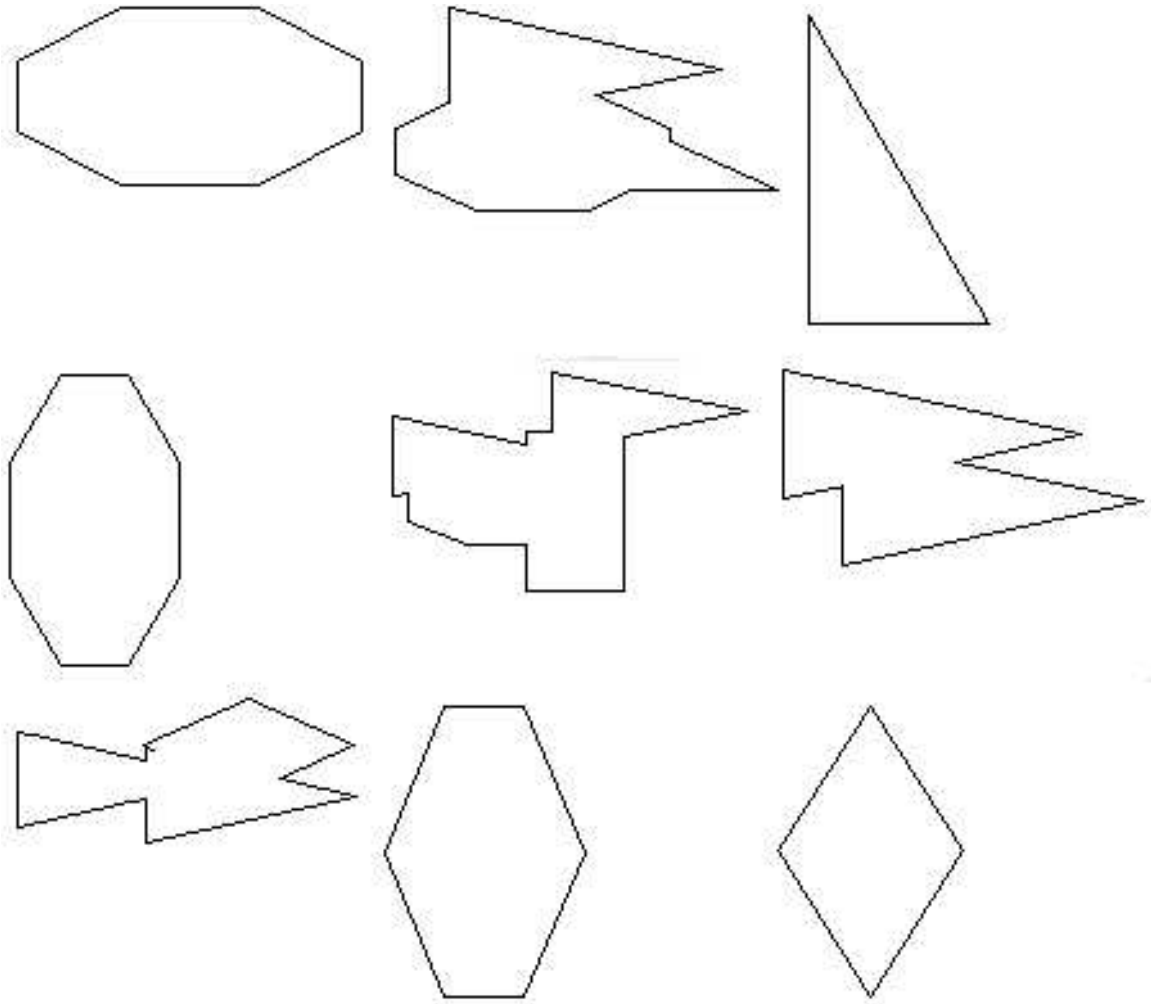

Figure 3.5: Floor plans of the ground floors of several A-Type wall generations

3.5.2 Evaluation of appearance

The buildings generated are certainly believable, as seen in Figure 3.5. No two buildings look alike, and they have floor plans similar to many buildings in real life.

This is not to say that they represent every skyscraper that can be made. There are many different types of skyscrapers (and I use the word "types" loosely, as every skyscraper is noticeably different from another), and the ones generated by this algorithm are simply one of these types. One of the beauties of the overall design of the algorithm is that it allows for more wall generators to be made and swapped out; creating more wall generators for skyscrapers is something that I would like to do in the future (see Section 6.4).

3.6 Speed evaluation of the wall generation algorithms

Speed is an important factor for all of these wall generation algorithms; they all must be able to generate the walls fast enough to be real-time. The results of the speed tests for these algorithms can be seen in Table 3.1. These tests were performed in the Eclipse Java development platform, on a machine with a Pentium 3 processor and 256 MB of RAM, running Fedora Core 3.

Wall Generator	**Avg time for 1,000 generations**
Courtyard	6.4 milliseconds
C-Type	21.5 milliseconds
B-Type	22.0 milliseconds
A-Type	248 milliseconds

Table 3.1: Results of speed tests on wall generation algorithms

Understandably, the generation algorithms take more time as they get more complex, but they are fast enough by themselves to generate in real time. The A-Type generation

algorithm is by far the slowest, yet it is still fast enough to generate just over 4,000 walls in one second. Rendering all of the buildings this fast would certainly be a task, but the wall generation algorithms would not slow the rendering down in the least.

Chapter 4

Simple Hallway Generation

We now have four algorithms for generating the outside walls of office buildings. What we need now are algorithms for generating the inside hallways. As stated before, hallways form the backbone of a building. They are the main connective units in an office building, and are one of the few constants in office buildings in general.

There are several other observations to be made about hallways in office buildings. Hallway layout in any building tries to maximize the space accessed by these hallways while minimizing the amount of space that the hallways themselves take up. On the other hand, hallway layout tries to stay as simple as possible to avoid confusion inside the building. These two factors can sometimes be in opposition to each other (i.e., if any two hallways meet at right angles, they are wasting space as the area near the intersection is accessed by two hallways at once.)

In this instance, however, we are more interested in realism, rather than optimization. While finding the optimal path for the hallways would be a very interesting problem, it lies outside the scope of this research. However, the overall algorithm that has been created would be a fairly good test bed for just such an exploration.

This chapter will describe a simple hallway generation algorithm. This algorithm is simple because it is designed to solve a very specific subset of problems in hallway generation, that of generating realistic hallways in rectangular, rectilinear spaces. That is, this algorithm is designed to generate hallways in buildings which are made up of straight lines at right angles to each other. In addition, while this hallway generator will generate the same hallways for the same outside lines, it never guarantees that a single point on every floor will be hit by a hallway; rather, the algorithm makes the assumption that the outside walls being passed into the algorithm are the same on every floor. If the walls are the same on every floor, then the hallways will be the same on every floor and this is not an issue.

All of these restrictions seem fairly onerous, but they actually cover a wide range of buildings; in fact they cover every wall generation algorithm covered in this research with the exception of the A-Type wall generator.

4.1 Goals

The generation of hallways in general has two goals, shared with everything else in this research: to be believable, and to be fairly fast. In this case, however, the variability that occurs in generation of the outsides of office buildings is a drawback rather than a benefit. We want hallways generated from the same outside walls to be the same. Otherwise, it would become impossible for elevators to be placed, causing rather severe problems.

We are also limited by the constraint of real-time, so the algorithm to generate these hallways must be fairly fast, though the speed requirement is not as important as believability.

4.2 The algorithm

The development of this algorithm follows a fairly normal pattern. There was, at the beginning, a fairly good idea, which, with some amount of work, became a fairly decent algorithm. Unfortunately, at the end of this idyllic cycle, certain problems became apparent and needed to be fixed. The original idea still holds, but the algorithm grew from a fairly simple algorithm to a slightly more complicated, better one in order to fix the problems that developed.

So first I will talk about the general idea behind the algorithm, followed by the original, fairly simple algorithm. Then I will discuss the problems that arose during that fairly simple program, and how they were addressed.

4.2.1 Concept

The concept of this algorithm began with the idea that the basic requirement of hallways in any building is that they provide access to every point inside that building. So the algorithm must somehow ensure that every point in the building is accessed by a hallway; it would be possible to lay hallways out inside of a building first, without checking first for accessibility, but in this case such a test would be a waste of time. It is better to do this checking first, preferably using it as a fundamental part of the algorithm.

So to construct the hallways in a floor, we can lay a grid of points down on top of the walls (as seen in Figure 4.1), iterate through them, and test each point for hallway access. If a point does not have access to a hallway, then we construct a hallway that creates access to that point, as well as maximizes the number of other points that gain access from this new hall. So we are trying to create access to every point while at the same time minimizing the number of hallways that need to be constructed for a given building.

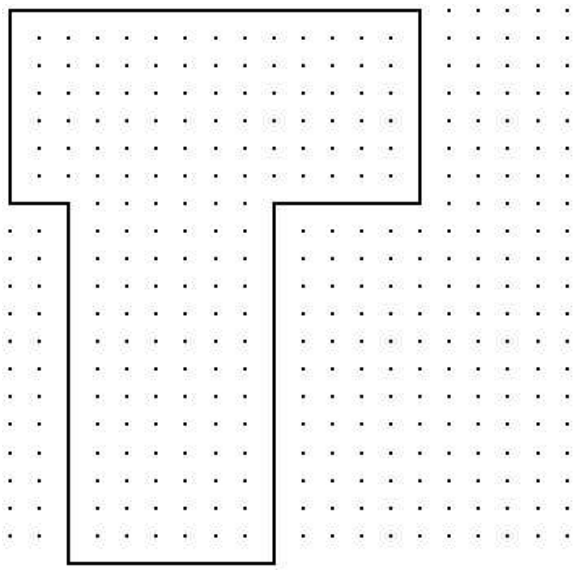

Figure 4.1: B-Type building with a point grid overlay

4.2.2 Original implementation

To implement this concept, four different things were needed: a way to lay out the grid of points, a way to test if a point is inside the building, a way to test if a point is accessed by a hallway, and a way to build new hallways.

As it turns out, the grid of points is fairly easy to construct. In fact, it is never constructed at all, it simply becomes two nested `for` loops over the area that the building covers. Unfortunately, these nested `for` loops do not discriminate between what is inside and what is outside of the building, so the need arises for a way to tell if a given point is inside or outside of a building.

If these buildings were regular polygons (rectangles, octagons, etc) it would be fairly easy to determine if a point is inside the polygon (the point has to lie in the interior of every angle made by adjoining sides). But the buildings that we will be dealing with are

not always simple polygons, so that is not an assumption that can be made, nor is that method very useful even as a starting point. As it turns out, there is a far simpler way to determine if a point lies inside of a polygon. Imagine a ray being shot from that arbitrary point in any direction, extending to infinity. If the point is inside a regular polygon, that ray will intersect exactly one side of the polygon. Similarly, if a point is inside a non-regular polygon, a ray sent in any direction will intersect an odd number of walls. If the point is outside the polygon, the ray will intersect either no walls, or an even number of walls. So we now have a very simple (indeed, simpler to compute than the original idea) concept of when a point is inside the building: send a ray from that point in some direction and count the number of intersections. This is a valid solution, as the walls of buildings are Jordan Curves; this solution is an application of the Jordan Curve Theorem.

Next we need a way to test if the points are accessed by a hallway. This is also fairly simple. When a hallway is created, the space that it accesses is well defined. So while we make the hallways, we can also make a list of the regions that those hallways cover. If we want to test if a point is accessed by a hallway, we iterate through this list of covered regions and see if any of the regions contain that point. If none of them do, then that point does not have access to hallways.

Making halls is equally simple; all that needs to happen is figuring out which direction (up/down vs. left/right) has a larger span and making a hallway that goes in that direction (except that both ends of the hall stop short one office width from the end of the building). Also, we need to add the region covered by that hallway to the list of covered regions.

At this point we (theoretically) have all we need. All that has to happen is to iterate through the grid points, and if the point is inside the building and not covered by a hallway, then we need to make a new hallway, add another covered region to the list, and move on.

4.2.3 Problems with the original implementation

This is, of course, a wonderfully clean and simple algorithm. Unfortunately, it produces rather ugly results. Granted, the results are close to what is desired, but they are messy enough to make the algorithm, in its current form, unusable. There are four main problems with the results, which lead to one major change in the algorithm and 3 minor improvements.

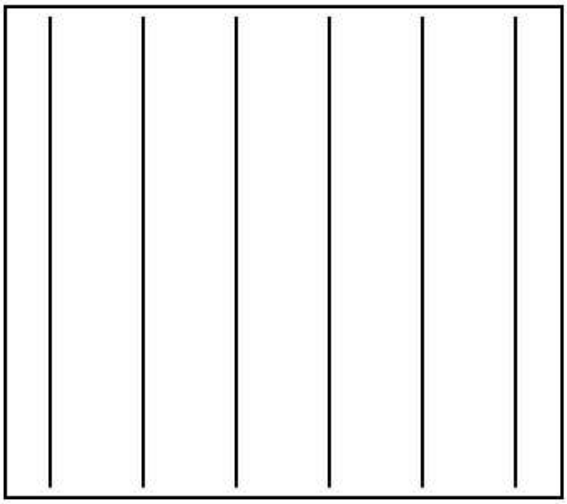

Figure 4.2: An example of unconnected hallways

Figure 4.2 is a good example of the main problem with the algorithm. In a purely rectangular shape, the halls are almost never totally connected. And hallways that are not connected to each other are useless for all practical purposes. So, first of all, we need some way to connect the halls, or barring that, some different way to generate the halls that ensures connectivity.

Figure 4.3 shows the second problem. Very often, the hallways will be unbalanced. This is not a huge problem, it is mainly aesthetic, but there is one side effect. Severely unbalanced hallways are a waste of space, and though optimal hallways are not our primary

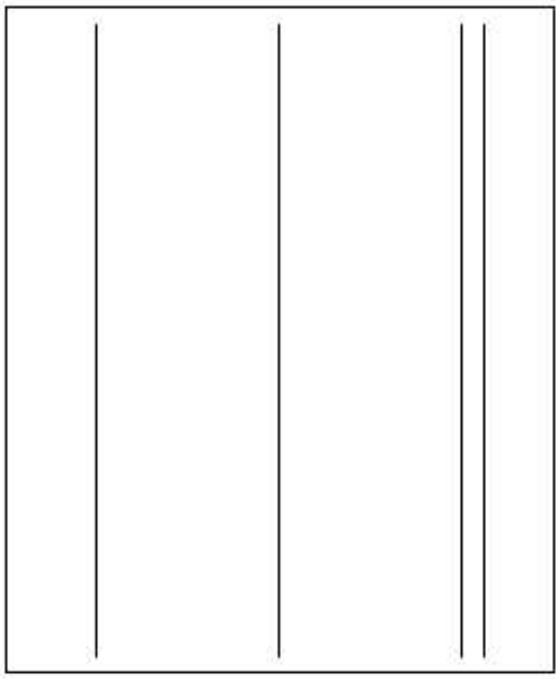

Figure 4.3: An example of unbalanced hallways

goal, hallways that use up way too much space aren't useful at all.

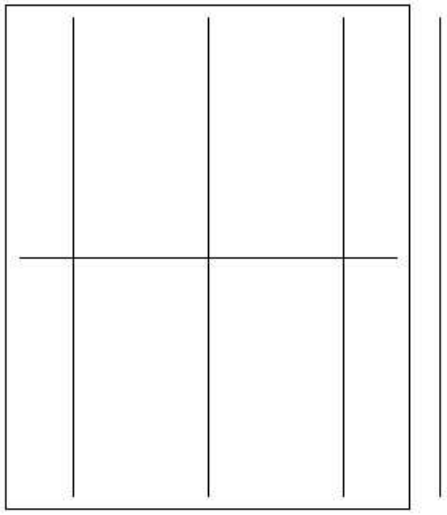

Figure 4.4: An example of hallways that lie outside or on the borders of the building

Figure 4.4 shows one of the more blatant problems: some of the halls in this building aren't even inside of the building! This happens occasionally in the original algorithm, when halls are constructed from points that are just barely inside the building. It would be possible to create every hallway inside the building, of course, but this error is actually better left in at this stage: hallways that lie just barely inside of the building get balanced with the rest of the hallways, often making all of the offices in between the hallways narrower. It is better to just let the hall be generated outside of the building and remove it later, rather than trying to remove it in the balancing process.

Figure 4.5 is the last problem. Essentially, the hallway lines place themselves in the middle of the hallway that would be generated in a real building, so the sides of each hallway are one office width plus one half of a hallway width away from one side of the building, and only one office width away from the building at the end of the hallway. When

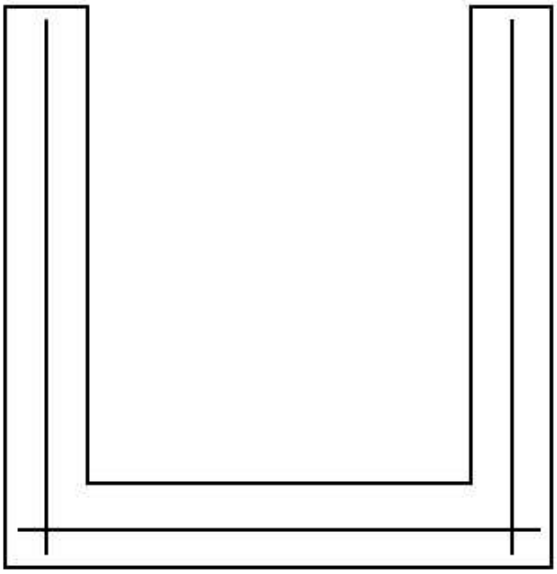

Figure 4.5: An example of untrimmed hallways

two hallways intersect, these ends need to be trimmed off. This problem, like the unbalanced hallways, is an aesthetic problem, but it makes a large difference in the appearance of the hallways.

4.2.4 Tweaks to fix the problems

The largest problem is that the hallways are not always connected. While having unbalanced hallways and hallways with untrimmed edges is certainly not pleasing to the eye, these hallways are at least still functional. But unconnected hallways are not even a valid solution to the hallway generation problem.

So the first tweak that was made to the original algorithm was a change of the overall structure from a simple iterative testing of the grid of points to a backtracking algorithm. This algorithm works exactly like the iterative testing algorithm on its first pass, but after

it has tested all of the points it checks to see if all of the hallways that have been generated are connected. If they aren't connected, the algorithm backtracks to the last hallway that was created and switches it so that it makes a hall in the perpendicular direction. It then proceeds normally until the end again, where it checks for connectedness again, and on up the decision tree until either the algorithm determines that it is not possible to make connected hallways in this building (in which case it calls on the original iterative function to do what it can), or a valid set of hallways is found.

This check for connectivity is rather interesting. First, we make two lists: a list of unconnected hallways and a list of connected hallways. Then we remove one hallway from the list of unconnected hallways and add it to the connected list. At this point we start looping. We test each connected hallway against the list of unconnected hallways, and if an unconnected hallway intersects with this test, we put it into a temporary list. At the end of each iteration that temporary list replaces the old connected list, and every hallway in the temporary list is removed from the unconnected list. In this way the method searches down the graph of connected hallways. It stops when either the unconnected or connected list is empty; if the unconnected list is empty, all the hallways are connected to each other, and there is no problem. If the connected list is empty, there are some hallways that are unconnected.

At this point the problem of unconnected hallways is taken care of. But the other three problems remain to be solved.

The problem of hallways' being outside of the building is easy to solve. A single iteration through the list of hallways is enough; if a hallway lies on the border of the building or lies entirely outside of the building, it is removed entirely. In addition, if a hallway lies partially outside of the building, the part of the hallway that lies outside of the building is trimmed off.

Balancing the hallways is a bit more complex. We are trying to shift all of the hallways around so that they are balanced. However, we don't want to shift them so much that we shift them outside of the building. To do this, first we check the distances from each of the endpoints of the hallway to the walls parallel to that hallway. The shorter of these sum distances from the endpoints is the distance that the hallway can move around in. So we take the shorter of these distances, and find out how many other hallways are in that space that we have to move around in. Then we shift the hallway to the left or the right to accommodate for the space that each of those hallways takes up. The space that each hallway takes up is determined by the number of hallways and the size of the area that has to be covered. If the new location for the hallway intersects any of the walls of the building, we have shifted the hallway too far, and instead of trying to shift the hallway back, we just leave it be.

Trimming the edges of the hallways is the easiest task to do; it just requires checking each hallway, and if that hallway is intersected at a point closer to one of the endpoints than one office width, then that end of the hallway needs to be trimmed off.

This is the outline of the full algorithm: check all of the points and make hallways when needed. Once we have reached the end of the points do the three minor modifications to the hallways (prune off all of the hallways which lie outside the building, balance the ones that lie inside the building, and trim off all of the edges of the hallways) then check for connectivity. If the halls aren't connected, back up one hallway and make a new hallway, and repeat until all the hallways are connected.

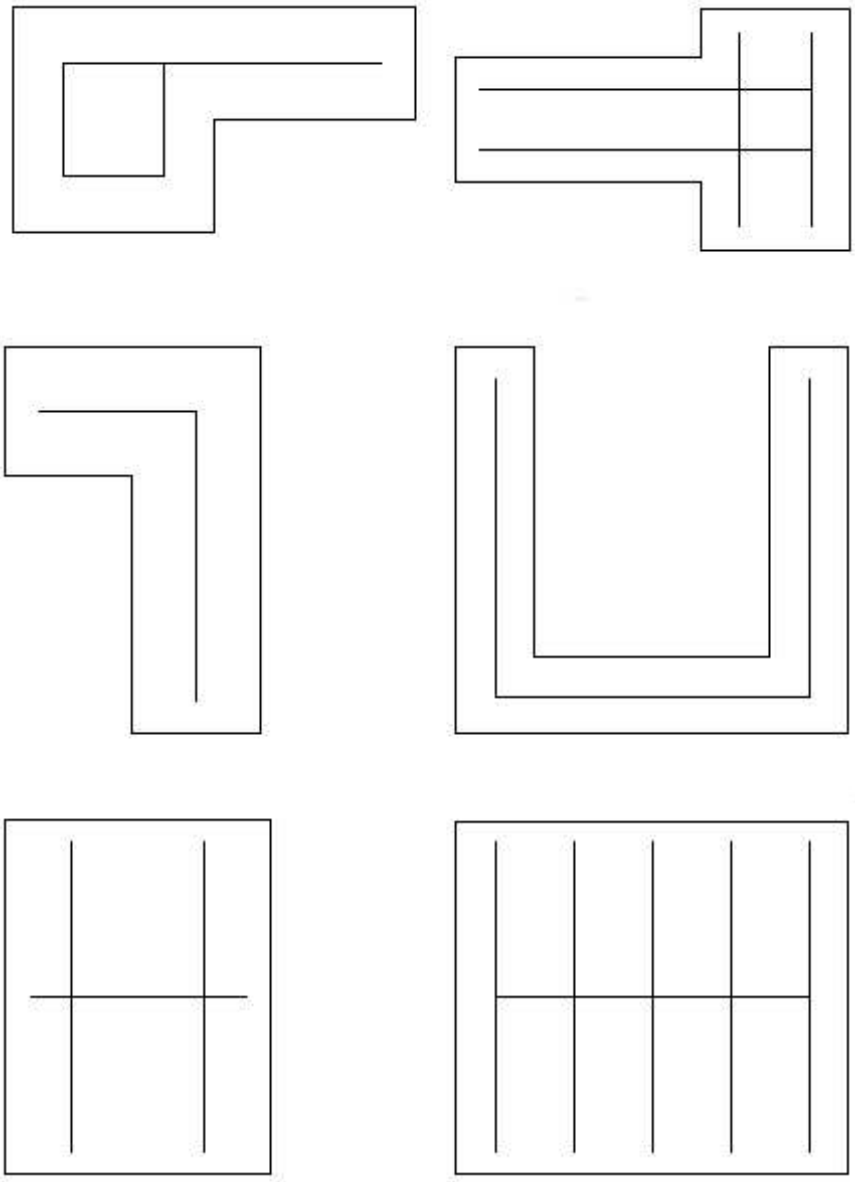

Figure 4.6: Floor plans for several B, C, and Courtyard buildings with finished hallway generation

4.3 Evaluation

As far as realism goes, this algorithm works quite well. As can be seen in Figure 4.6, the algorithm accurately creates hallways inside the subset of office buildings that we are interested in. These hallways are believable, and they are constant on the same input walls.

One proof of the soundness of this algorithm is that it was not designed to create hallways inside of courtyard buildings; at the time that this algorithm was written, the courtyard generation algorithm had not yet been created. However, courtyard buildings fit the requirements for this hall generator: they are rectangular and rectilinear, and every floor is the same. This hall generator worked perfectly without modification, generating the hallways inside of the courtyard buildings, as can be seen in Figure 4.6.

The algorithm, however, does not work quite so well where speed is concerned. (The results of the speed tests can be seen in Tables 4.1, 4.2, and 4.3.) Now, there are three things to note about the speed of this algorithm. First of all, the hallway generation algorithm is slower than the wall generation algorithms. This is to be expected as the algorithm is much more complex. Secondly, the speed of the algorithm seems to depend in part on the density of the grid of points. When that grid is denser (in other words, when the width between the grid points is smaller) there are more points that need to be tested and the algorithm runs slower.

There is one exception to this, which is the third thing to note. When the grid isn't dense enough, Courtyard buildings become much slower (see Figure 4.3). This happens because Courtyard buildings have a set width, and when the grid width becomes too wide the algorithm cannot find connected hallways inside the building and it begins backtracking. Often, it does not find a solution, and backtracks all the way back up and devolves to the original iterative generation. Note, however, that after that width becomes too wide, the

time goes down with the number of points to test; it is still maxing out the backtracking, but there is less backtracking to do.

This leads to a quandary - when there is high enough density that the hallways for Courtyard buildings do not backtrack, the hallways for B and C-Type buildings take approximately 5 to 10 times as long to generate. One solution to this would be to make another hallway generation algorithm to create hallways specifically for Courtyard buildings; another would be try to optimize the code that we already have and speed up the algorithm to work faster with a denser grid.

Nevertheless, when there is a large grid width, hallway generation for B and C-Type buildings becomes fast enough to generate over a thousand hallways in one second.

Grid area width	**# of points tested**	**Avg. time for 1000 generations**
One hallway (10)	900	4,422 milliseconds
One office (25)	144	1,422 milliseconds
Two offices (50)	36	622 milliseconds

Table 4.1: Speed tests on the hall generation algorithm using C-Type walls, with varying grid widths

Grid area width	**# of points tested**	**Avg. time for 1000 generations**
One hallway (10)	900	7,017 milliseconds
One office (25)	144	1,467 milliseconds
Two offices (50)	36	744 milliseconds

Table 4.2: Speed tests on the hall generation algorithm using B-Type walls, with varying grid widths

Grid area width	**# of points tested**	**Avg. time for 1000 generations**
One hallway (10)	900	6,713 milliseconds
One office (25)	144	62,170 milliseconds
Two offices (50)	36	43,130 milliseconds

Table 4.3: Speed tests on the hall generation algorithm using Courtyard walls, with varying grid widths

Chapter 5

Complex Hallway Generation

At this point we have several generation algorithms already implemented. We have four wall generation algorithms, covering a wide array of building styles (Chapter 3), and we have one hallway generation algorithm which covers a fairly large subset of building styles (Chapter 4). However, we do not have a hallway generation algorithm to cover the other styles of buildings that can be generated.

The simple hallway generation algorithm covers a wide range of building types, but there are restrictions on it. The buildings must be rectangular and rectilinear. The walls must also be the same on every floor; in other words every floor must be identical. Third, the walls must create a single shape; if a building, for instance, has two spires, the simple hallway generation algorithm cannot deal with it.

The complex hallway generation algorithm proposed here has only one restriction: the walls must create a single shape. There is no requirement for rectilinear or rectangular walls or a requirement for identical floors.

Because there are fewer restrictions there are a few more problems that need to be addressed with this algorithm. First of all, we need some way to guarantee that there is at

least one point in the building which can be accessed by a hallway on every floor; in other words a point for an elevator. Second, we need to find some way to ensure that our hallways get into every nook and cranny of the buildings (and with the A-Type building generation algorithm, there can be a lot of nooks and crannies).

Again, hallways are the backbones of these buildings. They are the main connective units of the buildings, and are one of the few constants in office building in general. Hallways in any building try to maximize the area accessed by the hallways while at the same time minimizing the space that the hallways themselves take up. Because we are more interested in realism than optimization, the algorithm does not focus on the optimal paths for the hallways; optimal paths would be an interesting problem, and the framework of the general algorithm would make an optimal path hallway generator rather easy to create, but they are not the focus of this research.

5.1 Goals

The generation of hallways in general has two goals, shared with everything else in this research: to be believable, and to be fairly fast. Unlike the wall generation algorithms, variability in the hall generation algorithms is a bad thing. In order to make vertically connective elements (elevators, stairs, etc), the hallways generated must be constant from floor to floor; if they aren't, and elevators and stairs cannot be made, and the building is useless.

We are also limited by the constraint of real-time, so the algorithm to generate these hallways must be fairly fast, though the speed requirement is not as important as believability.

5.2 The algorithm

5.2.1 Concept

The first thing to note about A-Type wall generation is that the floor plans generated by this algorithm (and indeed, any floor plan in general) are composed of several different shapes placed together. So the first part of this algorithm is essentially data gathering: taking the walls passed into this algorithm and splitting them up, if necessary, into several different shapes. Sometimes, when the shapes are added onto one another, the shapes lie almost on top of each other, making one large shape. Other times, however, the shapes get added onto one another in such a way that they are barely touching (see Figure 5.1). This can be a major problem, as the hallways inside that building must be connected to each other but that connection can only pass through a very small space; it would be rather hard to make a general algorithm that always manages to make a hall that passes through that small gap. Splitting up the buildings into several different shapes and keeping track of where those shapes are joined is a necessary step for an algorithm that must connect the entire floor with hallways.

After the data has been gathered, the hallways are made in a spine and rib fashion. That is, there is a main hallway that runs through the entire shape (the spine), and other, secondary hallways which branch off from the first (the ribs). The problem of nooks and crannies comes in at this point, as there are almost always parts of the building that have not been accessed by the hallways. So secondary spines connected to a main rib are made, with secondary ribs branching off of them for every area that does not have access to halls.

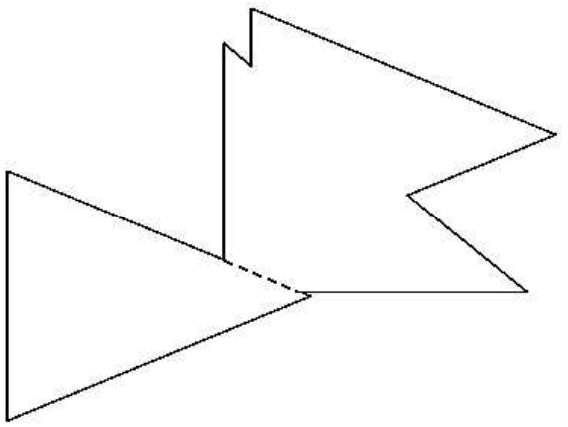

Figure 5.1: The ground floor of an A-Type building with a crux line

5.2.2 Implementation of data gathering

Gathering the data needed to construct the hallways is rather difficult. We need to get 3 pieces of information, and 2 of those are closely related. The first is finding out if there are any areas where the building is bottlenecked, where there is only a small area for a hall to pass through. Such areas can be represented as lines between the two points which define the bottleneck, called crux lines (see Figure 5.1). The second thing we need to find is how to split up the building into several different shapes, connected by the crux lines. This is necessary so that we can build the spine of the building through all of the main areas of the building. The third thing that we need to be able to find is the center of an area; this is needed for drawing the spine, though it does come in handy in other places.

Determining the crux lines of a shape, while not difficult, isn't all that easy either. First of all, we need to deal with single vertices. In the past, we have been dealing with lines, so we need to first get some sort of list or array of every vertex in the shape. Once we have that, we construct a complete graph out of the vertices then filter that graph through

several tests to determine which pairs of vertices form crux lines. First of all, we remove every pair of vertices which lie farther than two office widths plus one hall width away from each other. Next, we remove all of those pairs of vertices which share a wall with each other as well as any pairs of vertices which, when a line segment is drawn between them, intersect any of the walls of the building. Vertices which pass all of these tests still might not be valid, as the line segment might be completely outside of the shape. So we test to see if the midpoint of the line segment is inside the shape (see Subsection 4.2.2), and remove any pairs of vertices which fail. Lastly, there are some crux lines which remain, in which the crux line forms a triangle with the walls of the building, as seen in Figure 5.2; these vertices must be removed as well. After all of these tests are passed, the lines remaining are crux lines for the buildings.

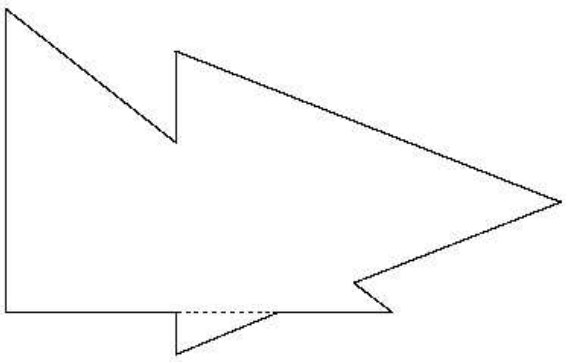

Figure 5.2: A crux line (dotted line) forming a triangle with the walls

Next we need to determine how to split up the building around the crux lines. This is fairly simple. If we map out the connections between the vertices of the building walls, with loops underneath representing the connections by the crux lines, we see that these connections are very regular (see Figure 5.3). All we need to do is go through this list following the crux lines whenever they loop, and make lines between each of the points. These lines

form the individual shapes of the building that the algorithm deals with (hereafter referred to as "building shapes").

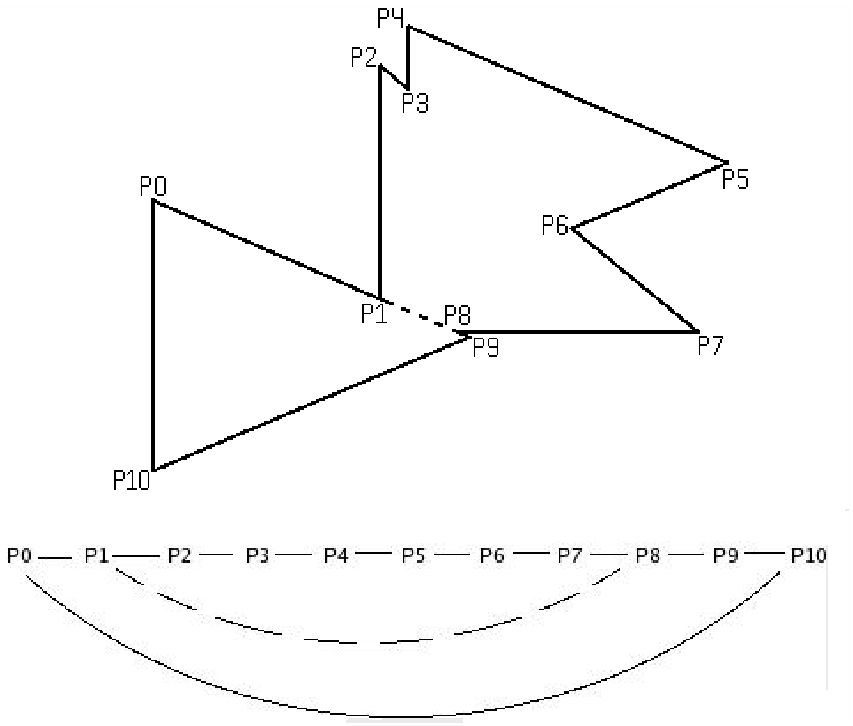

Figure 5.3: An A-type building and an abstract graph of the connections between the points

Lastly, the algorithm needs to find the middle points of each of these building shapes. There are more complicated ways of doing this, but they aren't necessary for our purposes. What the algorithm does is find the maximum and minimum X and Y values of the shape, then guesses that the point in the middle of those values is the center. It then tests if that point is indeed inside the shape, and if it is, then we are done. Otherwise, it starts moving the point around (up, down, left, and right) in increasing amounts until it finds a point that is in the shape. This does not find the actual geometric center of most shapes, but it does approximate the correct point, and that is all that is needed for our purposes.

5.2.3 Generating the spine

The spine of the building goes through every building shape. The shape of the spine in each of the building shapes depends on the number of crux lines on the border of that building shape. If there are no crux lines in a building shape, then the building has no crux lines either. Every building shape has a center, and we need to make a hallway through that center that maximizes the amount of office space covered. To do this we iterate through the walls of the building and find the farthest midpoint from the center of the building that can be reached by a hall without intersecting any of the other walls. We then make a hall from that midpoint in the direction of the center of the shape, continuing on to the other end of the building. The spine for such a building looks like Figure 5.4.

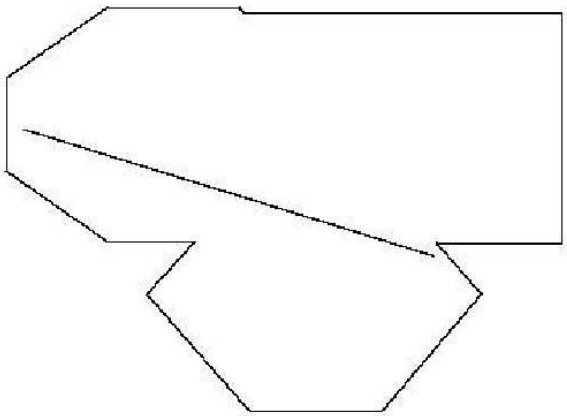

Figure 5.4: Spine of building with no crux lines

If there is one crux line, then we need to make a hallway that goes from the midpoint of that crux line through the center of the building shape to the opposite wall. This ensures that there is a hallway which connects to the other shapes. In a building with two building shapes (one crux line), the spine looks like Figure 5.5.

In a building shape with two or more crux lines, both crux lines need to have hallways

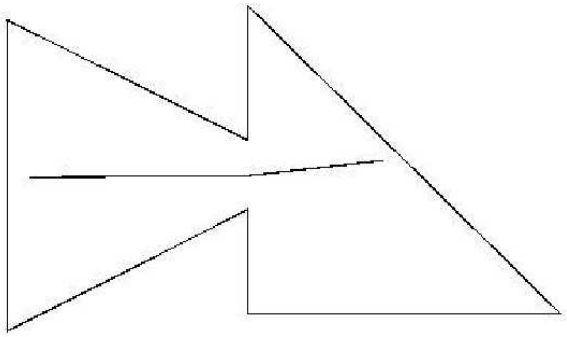

Figure 5.5: Spine of building with one crux line

going to them, to ensure that the spine goes unbroken through all of the shapes. So in this case we need to build a hallway from the midpoint of each crux line to the center of the shape. A building in which this happens looks like Figure 5.6.

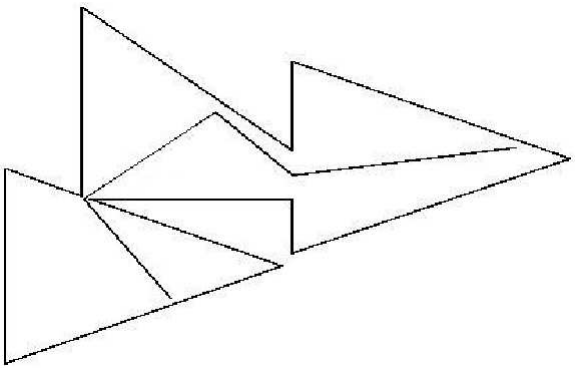

Figure 5.6: Spine of building with two crux lines

5.2.4 Generation of ribs and secondary hallways

Now that the spine of the building has been constructed, all that remains is to construct the ribs of the building and the secondary hallways. The concept behind how these hallways

should be generated is not that difficult, but due to time constraints this portion of the algorithm was not completed. However, the concept behind this part of the algorithm can be laid out.

The ribs should be constructed at right angles to the spine, at equal distances. They should extend to the edges of the building (minus one office width). This would cover a large portion of the building. These rib hallways will often meet, especially when there are two or more crux lines in a building shape (in this case the spine could be at a right angle to itself). When this happens they will truncate each other, making a series of loops going from one portion of the spine to the other.

However, there are parts of the building which cannot be reached in this fashion, and these parts of the building would have to have secondary spine and rib hallways that would connect to the nearest main rib and fill up the space that could not be reached by the main ribs.

5.3 Evaluation

This algorithm is not yet fully implemented, so it is hard to fully evaluate it. However, some things can be said. The data finding and spine generation portion of the algorithm seems to work quite well. It is fairly realistic, and provides a connection mechanism for the different shapes of the building. The rib generation portion of the algorithm should work quite well and provide hallway access to all parts of the building. As this algorithm is incomplete, speed tests are not possible.

Chapter 6

Conclusions

6.1 The goals of this research

The primary goal of this research was to generate office buildings in a virtual world, with success measured by believability, usability, and extensibility. Believability was paramount, followed by usability and extensibility.

6.2 The results of this research

For this research I did six things, and made significant progress on a seventh.

First of all, and most importantly, I developed a general algorithm for office building generation. Office buildings, under this model, are split up into floors in a pancake design. Then the individual parts of the building are generated from the outside in using external generators. These external generators are easy to implement and provide unbounded extensibility to the general algorithm.

I created two separate classes of generators: wall generators and hallway generators.

The wall generators are the most variable part of the general generation algorithm, just like the outsides of buildings are the most variable in real life. Hallway generators, while not variable, provide the interior connectivity needed in the office buildings.

I wrote four separate wall generators: A-Type, B-Type, C-Type, and Courtyard wall generators. A-Type wall generators are the most complex, and illustrate well the amount of variation that is available for the general algorithm. The other three generators, while simpler, accurately model a large number of the smaller office buildings that are the majority of office buildings in most urban areas. These three also show well how different wall generation algorithms can be used to great effect in conjunction with a single hallway generation algorithm.

I wrote one simpler wall generation algorithm which accurately generates hallways for all rectangular and rectilinear buildings; it can be used in conjunction with B-Type, C-Type and Courtyard buildings.

I also made a good amount of progress on a more complex wall generation algorithm, and laid out the plan for completing it. This wall generation algorithm accurately finds the critical points in a non-rectangular shape and creates a single hallway that spans through all of the conglomerated shapes that make up the more complicated buildings.

6.3 Evaluation of results

The development of the general algorithm was the major success of this research. It is believable, usable, and most of all extensible. Splitting up the buildings into floors then using generators to generate the individual parts of a building was a great success; it generated realistic simple buildings easily, and had very little problems with complex buildings. The general algorithm cannot be measured by speed, as most of the complexity is inside the

individual generators; however, the code for the general algorithm is little more than a series of calls to generators, and by itself is rather fast. The general algorithm is also very extensible. Through the use of different generators many more buildings types can be created as the need arises. Almost every kind of office building imaginable would be able to be generated by this algorithm using a series of generators.

The wall generator algorithms are all fast, and all produce realistic floor plans (as can be seen in Figure 6.1). The slowest wall generator algorithm (the A-Type wall generator) could still on average generate over four thousand buildings in under a second (see Table 3.1).

The simple hallway generation algorithm is very realistic, and a wonderful general algorithm. As long as the walls of the building match certain criteria, the simple hall generation algorithm works well. The Courtyard wall generation algorithm had not been written when this project was finished, yet the simple hallway generation algorithm generated the needed hallways perfectly. Unfortunately, the simple hallway generation algorithm is slow; there are a few speedups that could be done to the algorithm to make it faster, and they would have to be done before the algorithm could really be fast enough to generate an entire city of office buildings at one time.

The complex hallway generation algorithm is incomplete, but it still gathers the information needed about the walls of the building quite well and generates the central hallway of a non-rectangular office building accurately.

The goals of this research were met. A general algorithm for building generation as well as several specific generation algorithms to fill in the details were designed and implemented, and they all met the goals of believability, usability, and extensibility.

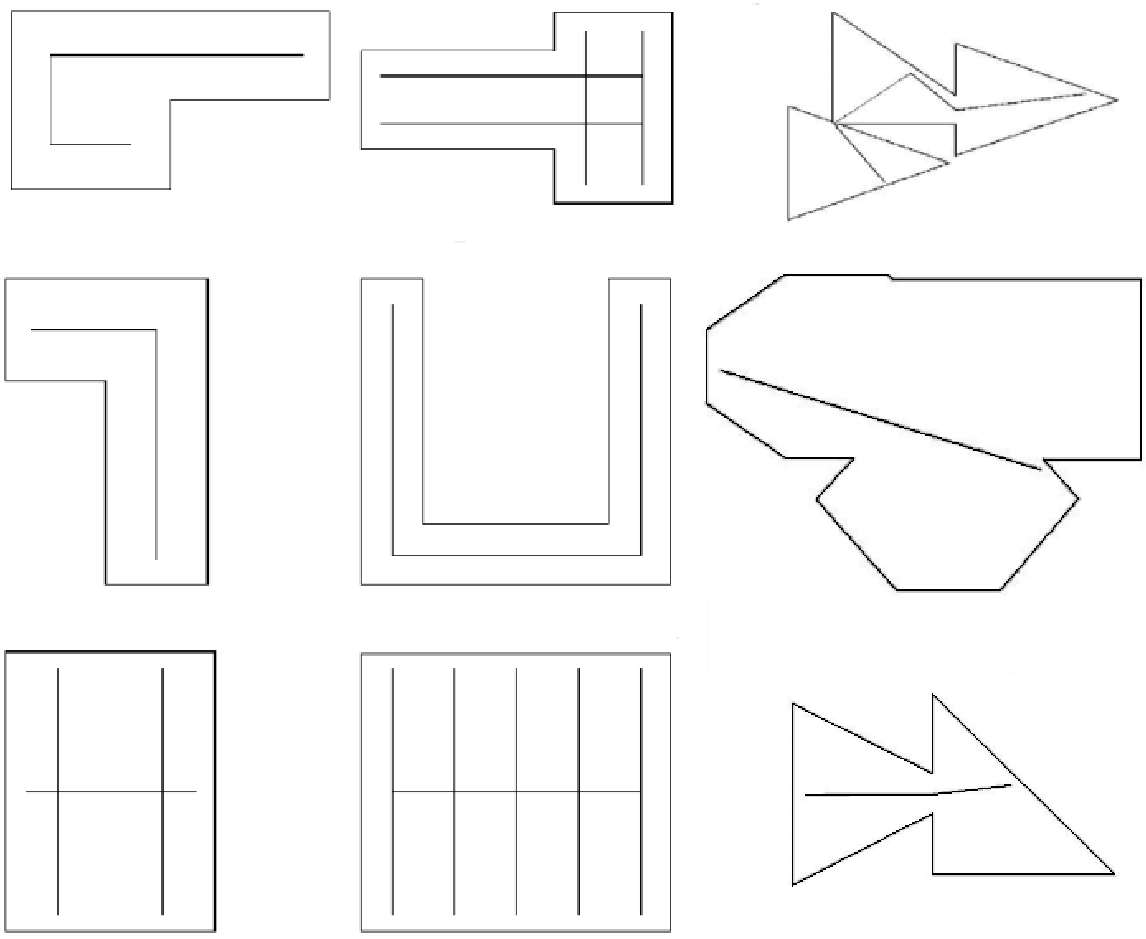

Figure 6.1: Final floor plans for several generated buildings

6.4 Possibilities for further research

There are many different possibilities for further research in and related to this area. This kind of traversable world generation is a rather new area, and not fully developed.

There are many possibilities for further research which extend directly from the topic of this thesis. First of all, there are many different styles of A-Type buildings that the A-Type wall generator cannot construct. Buildings which have multiple towers and so-called "keyhole" buildings (where there is a hole through the top parts of the building), for example, cannot be constructed by the A-Type building generator in this thesis.

Also, the hallway generators do not cover all of the possible building configurations; there is a need for hallway generation in those keyhole buildings and buildings with multiple spires.

There is also a need for algorithms which construct the more detailed parts of office buildings: lobbies and atria generators, office space generators, and window generators, as well as the elevator shafts and stairs which this research, while presuming they would exist, never actually generated.

There are also several avenues of research which, while related to this research, are fundamentally different. The general algorithm implemented in this research generates buildings from the outside to the inside. When conceptualizing this algorithm, the outsides of the buildings seemed much more important. However, during the course of this research, the thought occurred that it would be quite possible to generate buildings from the inside to the outside. A graph-generation algorithm that could generate the hallways for the building, then mold the outside of the building around those hallways would be very interesting to try, and might work quite well.

In addition, it has been mentioned several times during the course of this thesis that an

optimal path hallway generation algorithm would be very interesting. The framework of this project is exceptionally well suited to such an endeavor; with the ability to add on new wall generators at will, there is a fairly large sandbox for such an algorithm to play around in, and creating a new hallway generation algorithm to work with such a wall generation algorithm is fairly simple (it is the particular details of such a generation algorithm that would be tricky).

6.5 Concluding remarks

From the beginning this research has been very interesting. Attempting to generate entire worlds is a very interesting project which, as computing power increases in future years, could prove to be very important. At the same time though, creating a world accurately requires a large amount of creativity. Architects are not in danger of losing their jobs as a result of this research, to put it mildly. It can be extraordinarily difficult to attempt to mimic actions which humans can do easily (such as adding one shape to another in the A-Type wall generation algorithm) using a computer; that difficulty adds to the fun of the research, as well as provides a wonderful glimpse into how amazing our minds are.

Bibliography

[1] Stefan Greuter, Jeremy Parker, Nigel Stewart, and Geoff Leach. Real-time procedural generation of 'pseudo infinite' cities. In *Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia.* ACM Press, NY, NY, USA, February 2003.

[2] Markus A. Haas. *Dynamic Virtual Worlds: A Notion of Realism in Future Virtual Reality Applications.* Honors Thesis, Trinity University, 1996.

[3] Konrad Knopp. *Theory of Functions, Volume I.* Dover Publications, New York, 1946.

[4] Mark Lewis. *Large Scale Virtual Worlds: Algorithmic Construction and Dynamic Methods.* Honors Thesis, Trinity University, 1996.

[5] Jess Martin. *The Algorithmic Beauty of Buildings: Methods for Procedural Building Generation.* Honors Thesis, Trinity University, 2005.

Appendix A

Appendix of Selected Code

A.1 The General Algorithm

A.1.1 OfficeBuilding.java

OfficeBuilding.java is the implementation of the general algorithm.

```
import java.awt.geom.Line2D;

public class OfficeBuilding {
	//Constructors and Public Methods
	public OfficeBuilding(){}
	public OfficeBuilding(WallGenerator newWG, HallGenerator newHG,
			double newMaxX, double newMaxY, int newFloorCount)
	{
		wg = newWG;
		hg = newHG;
		maxX = newMaxX;
		maxY = newMaxY;
		floorCount = newFloorCount;
		officeWidth = officeWidthDefault;
		hallWidth = hallWidthDefault;

		build();
	}
	public OfficeBuilding(WallGenerator newWG, HallGenerator newHG, double newMaxX,
			double newMaxY, int newFloorCount, double newOfficeWidth, double newHallWidth)
	{
		wg = newWG;
		hg = newHG;
		maxX = newMaxX;
		maxY = newMaxY;
		floorCount = newFloorCount;
		officeWidth = newOfficeWidth;
		hallWidth = newHallWidth;

		build();
	}
	public void build(){ //This does not yet deal with any cool things like atria or elevators
		walls = wg.genWalls(maxX, maxY, floorCount); //Build the walls of the building
		halls = new Line2D.Double[walls.length][];
		for(int i=0; i<walls.length; i++){ //Iterate through the floors and build halls
			halls[i] = hg.genHalls(walls[i], hallWidth, officeWidth, walls[walls.length-1]);
```

```
                }
        }
        public void draw(){
        }
        public Line2D[][] getBuilding(){
                Line2D[][] b=new Line2D[floorCount][];
                int j;
                for(int i=0; i<floorCount;++i){
                        b[i]=new Line2D[walls[i].length+halls[i].length];
                        for(j=0;j<walls[i].length;++j){
                                b[i][j]=walls[i][j];
                        }
                        for(j=0;j<halls[i].length;++j){
                                b[i][j+walls[i].length]=halls[i][j];
                        }
                }
                return b;
        }

    //Private Members
        private HallGenerator hg; //the generator to make the halls
        private WallGenerator wg; //the generator to make the walls
        private Line2D.Double[][] walls; //double array holding the walls
        private Line2D.Double[][] halls; //double array holding the hallways
        private double maxX, maxY/*, maxZ*/; //Maximum dimensions for the footprint of thebuilding.
        private double officeWidth, hallWidth; //self explanatory
        private int floorCount;
        private double hallWidthDefault = 10, officeWidthDefault = 25; //default values
}
```

A.1.2 WallGenerator.java

WallGenerator.java is the interface for all of the wall generation algorithms.

```
import java.awt.geom.Line2D;

public interface WallGenerator {
        Line2D.Double [][] genWalls(double newMaxX, double newMaxY,
                        int newFloorCount);
}
```

A.1.3 HallGenerator.java

HallGenerator.java is the interface for all of the hallway generation algorithms.

```
import java.awt.geom.Line2D;

public interface HallGenerator {
        Line2D.Double[] genHalls (Line2D.Double[] walls, double newHallWidth,
                        double newOfficeWidth, Line2D.Double[] startShape);
}
```

A.2 Wall Generation Algorithms

A.2.1 AWallGenerator.java

AWallGenerator.java is the implementation of the A-Type wall generation algorithm.

```
import java.awt.geom.Line2D;
import java.awt.geom.Point2D;
import java.util.Iterator;
import java.util.Random;

import buildingShapes.BuildingShapeGenerator;
import buildingShapes.BuildingShape;

public class AWallGenerator implements WallGenerator {
        //Constructor and Public Methods
        public AWallGenerator(){}//<-------Null Constructor
        public AWallGenerator(double officeWidth, double hallWidth){}//<-------Null Constructor
```

```
public Line2D.Double[][] genWalls(double newMaxX, double newMaxY, int newFloorCount) {
        Random rgen = new Random();
        //Figure out how many different shapes we are going to use
        //I figure one possible shape for 8 floors is a decent estimation
        if(newFloorCount/8==0)
                numShapesInBuilding=1;
        else{
                numShapesInBuilding = rgen.nextInt(newFloorCount/8)+1;
        }
        fc=newFloorCount;
        maxX=newMaxX;
        maxY=newMaxY;
        walls = new Line2D.Double[fc][];

        int currentFloor=fc-1; //Start at the top floor
        for(int i=0;i<numShapesInBuilding-1;++i){ //Do this for number of shapes minus one times.
                addShape(currentFloor);//Add a shape to the building
                int numF=rgen.nextInt(3)+6;
                extrudeDown(currentFloor,numF);//Start at a "floor", extrude down x number of floors;
                currentFloor=currentFloor-numF-1;//Move currentFloor down to the next open floor
        }
        addShape(currentFloor);
        extrudeDown(currentFloor,currentFloor); //Extrude down to the bottom of the building.

        return walls;
}

//Private Members
private Line2D.Double[][] walls;
private int fc, numShapesInBuilding;//floorCount, numberOfShapes
private double maxX, maxY;
//Private Methods
private void addShape(int currentFloor){
        //Add a shape to the list of floors, and make this floor.
        /* Get a new Shape from generator;
         * If floor is the top floor, the walls are the shape
         * If not:
         *      The walls for the floor above the current floor will be comprised of Line2Ds
         *      Make sure the walls for the current shape intersect the walls of the above floor
         *      If they dont, shift the new bounding box.
         *      Go through the vertices of the new shape.
         *              Remove all vertices inside the old building.
         *      Go through all the vertices of the old shape
         *              Remove all vertices inside of the new shape.
         *      Combine the two temporary shapes to be the new walls
         * Set walls[currentFloor] to be the combination
         */
        //Get a new shape from the generator and scale it
        Random rgen=new Random();
        double ran=rgen.nextDouble();
        BuildingShapeGenerator bsgen = new BuildingShapeGenerator();
        BuildingShape bs = bsgen.getBuildingShape(rgen);
        bs.scale(ran*maxX/4+maxX/4,(1-ran)*maxY/4+maxY/4);
        bs.translate(rgen.nextDouble()*maxX/2,rgen.nextDouble()*maxY/2);

        if(currentFloor+1 >= walls.length){ //If this is the first shape
                Line2D.Double[] ls=bs.getWalls(),
                        temp=new Line2D.Double[ls.length];
                for(int i=0;i<ls.length;++i){
                        temp[i]=new Line2D.Double(Math.rint(ls[i].getX1()),
                                        Math.rint(ls[i].getY1()),
                                        Math.rint(ls[i].getX2()),
                                        Math.rint(ls[i].getY2()));
                }
                walls[currentFloor]=temp;
        }
        else{//If this is not the first Shape
                //Get the shape of the walls for the floor above.
                Line2D.Double[] aboveWalls=walls[currentFloor+1];
                //Get the shape of the walls for the current floor.
                Line2D.Double[] newWalls=bs.getWalls();
                //Make sure that the floors intersect.

                int intersectCount=0;
                while(intersectCount==0){
                        intersectCount=0;
```

```
                for(int i=0;i<newWalls.length;++i)
                        for(int j=0;j<aboveWalls.length;++j)
                                if (newWalls[i].intersectsLine(aboveWalls[j])) intersectCount++;
                if(intersectCount==0)
                        bs.translate((rgen.nextDouble()-.5)*maxX/2,(rgen.nextDouble()-.5)*maxY/2);
        }

        ObjectList wl=new ObjectList();
        alterOldLines(aboveWalls,newWalls,wl);//Modify the older lines to work around the new shape
        alterNewLines(aboveWalls,newWalls,wl);//Modify the new lines to work around the old shape
        walls[currentFloor]=new Line2D.Double[wl.length()];
        int i=0;
        Line2D.Double l;
        for(Iterator iter=wl.iterator();iter.hasNext();){
                l=(Line2D.Double)iter.next();
                walls[currentFloor][i]=new Line2D.Double(Math.rint(l.getX1()),
                                Math.rint(l.getY1()),
                                Math.rint(l.getX2()),
                                Math.rint(l.getY2()));
                ++i;
        }
    }//End else{If this is not the first shape in the building
}
private void alterOldLines(Line2D.Double[] aboveWalls, Line2D.Double[] newWalls, ObjectList wl){
    //Alter the oldlines.
    for(int i=0; i<aboveWalls.length;++i){
        //Count the number of vertices of the oldwall line inside the new walls
        int vertsInside=0;
        if(inShape(new Point2D.Double(aboveWalls[i].getX1(),aboveWalls[i].getY1()),newWalls))
                vertsInside++;
        if(inShape(new Point2D.Double(aboveWalls[i].getX2(),aboveWalls[i].getY2()),newWalls))
                vertsInside++;

        if(vertsInside==0){ //There are 2 cases - no intersects, or an even number of intersects
                //Find the number of intersecting lines
                ObjectList intersectingLines=new ObjectList();
                for(int j=0;j<newWalls.length;++j){
                        if(aboveWalls[i].intersectsLine(newWalls[j])) intersectingLines.add(newWalls[j]);
                }
                if (intersectingLines.length()==0) //No intersecting lines
                        wl.add(aboveWalls[i]);
                else{ //There are intersecting lines
                        //Find out the distances from each endpoint to each intersecting line.
                        //Put the intersecting lines into an array.
                        double[][] dists= new double[2][intersectingLines.length()];
                        Line2D.Double[] intLines=new Line2D.Double[intersectingLines.length()];
                        int j=0;
                        for (Iterator iter=intersectingLines.iterator();iter.hasNext();){
                                intLines[j]=(Line2D.Double)iter.next();
                                dists[0][j]=intLines[j].ptLineDist(aboveWalls[i].getP1());
                                dists[1][j]=intLines[j].ptLineDist(aboveWalls[i].getP2());
                                j++;
                        }
                        //Find the Points of Intersection
                        Point2D.Double[] pointsOfIntersection
                                =new Point2D.Double[intersectingLines.length()];
                        double xdiff=aboveWalls[i].getX2()-aboveWalls[i].getX1(),
                                ydiff=aboveWalls[i].getY2()-aboveWalls[i].getY1(),
                                ratio;
                        for(j=0;j<pointsOfIntersection.length;j++){
                                ratio=dists[0][j]/(dists[0][j]+dists[1][j]);
                                pointsOfIntersection[j]=new Point2D.Double(
                                        aboveWalls[i].getX1()+xdiff*ratio,aboveWalls[i].getY1()+ydiff*ratio);
                        }
                        //Find out what point of intersection is closest to the first endpoint
                        double minLen=Double.POSITIVE_INFINITY;
                        int minPointIndex=-1;
                        for(j=0;j<pointsOfIntersection.length;j++){
                                if(aboveWalls[i].getP1().distance(pointsOfIntersection[j])<minLen){
                                        minLen=aboveWalls[i].getP1().distance(pointsOfIntersection[j]);
                                        minPointIndex=j;
                                }
                        }

                        if(minPointIndex==-1){
                                return;
```

```
                                }

                                //Make a line from the first endpoint to that intersection, add it to wl
                                wl.add(new Line2D.Double(aboveWalls[i].getP1(),pointsOfIntersection[minPointIndex]));
                                //Find out what intersecting line is closest to the second endpoint
                                minLen=Double.POSITIVE_INFINITY;
                                minPointIndex=-1;
                                for(j=0;j<pointsOfIntersection.length;j++){
                                        if(aboveWalls[i].getP2().distance(pointsOfIntersection[j])<minLen){
                                                minLen=aboveWalls[i].getP2().distance(pointsOfIntersection[j]);
                                                minPointIndex=j;
                                        }
                                }
                                //Make a line from the second endpoint to that intersection, add it to w;
                                wl.add(new Line2D.Double(aboveWalls[i].getP2(),pointsOfIntersection[minPointIndex]));
                        }
                }
                if(vertsInside==1){ //Only one case this time around.  We only care about the outer intersect
                        //Find the newlines that intersect the oldline
                        ObjectList intersectingLines=new ObjectList();
                        for(int j=0;j<newWalls.length;++j){
                                if(aboveWalls[i].intersectsLine(newWalls[j])) intersectingLines.add(newWalls[j]);
                        }
                        //Find the distances from the endpoints to the intersecting lines
                        double[][] dists= new double[2][intersectingLines.length()];
                        Line2D.Double[] intLines=new Line2D.Double[intersectingLines.length()];
                        int j=0;
                        for (Iterator iter=intersectingLines.iterator();iter.hasNext();){
                                intLines[j]=(Line2D.Double)iter.next();
                                dists[0][j]=intLines[j].ptLineDist(aboveWalls[i].getP1());
                                dists[1][j]=intLines[j].ptLineDist(aboveWalls[i].getP2());
                                j++;
                        }
                        //Find the endpoint outside of the shape
                        int outsidePointIndex=-1;
                        if(inShape(aboveWalls[i].getP1(),newWalls))
                                outsidePointIndex=1;
                        else
                                outsidePointIndex=0;
                        //Find the Points of Intersection
                        Point2D.Double[] pointsOfIntersection
                                =new Point2D.Double[intersectingLines.length()];
                        double xdiff=aboveWalls[i].getX2()-aboveWalls[i].getX1(),
                                ydiff=aboveWalls[i].getY2()-aboveWalls[i].getY1(),
                                ratio;
                        for(j=0;j<pointsOfIntersection.length;j++){
                                ratio=dists[0][j]/(dists[0][j]+dists[1][j]);
                                pointsOfIntersection[j]=new Point2D.Double(
                                        aboveWalls[i].getX1()+xdiff*ratio,aboveWalls[i].getY1()+ydiff*ratio);
                        }
                        //Find the closest intersecting line to the outside point.
                        double minLen=Double.POSITIVE_INFINITY;
                        int minPointIndex=-1;
                        double pDist=-1;
                        for(j=0;j<pointsOfIntersection.length;j++){
                                if(outsidePointIndex==0) pDist=aboveWalls[i].getP1().distance(pointsOfIntersection[j]);
                                else pDist=aboveWalls[i].getP2().distance(pointsOfIntersection[j]);
                                if(pDist<minLen){
                                        minLen=pDist;
                                        minPointIndex=j;
                                }
                        }
                        //Make a line from the outside endpoint to that intersection, add it to w;
                        Point2D.Double pointOfIntersection;
                        if(outsidePointIndex==0){
                                wl.add(new Line2D.Double(aboveWalls[i].getP1(),pointsOfIntersection[minPointIndex]));
                        }else{
                                wl.add(new Line2D.Double(aboveWalls[i].getP2(),pointsOfIntersection[minPointIndex]));
                        }
                }else{} //If both vertices are inside, dont do anything (erase the line)
        }//End for(over all oldlines/abovewalls)
}
private void alterNewLines(Line2D.Double[] aboveWalls, Line2D.Double[] newWalls, ObjectList wl){
        //Now loop over the new lines and do the same thing.  basically ;)
        for(int i=0; i<newWalls.length;++i){
                //Count the number of vertices of the oldwall line inside the new walls
```

```
    int vertsInside=0;
    if(inShape(new Point2D.Double(newWalls[i].getX1(),newWalls[i].getY1()),aboveWalls))
            vertsInside++;
    if(inShape(new Point2D.Double(newWalls[i].getX2(),newWalls[i].getY2()),aboveWalls))
            vertsInside++;

    if(vertsInside==0){ //There are 2 cases - no intersects, or an even number of intersects
            //Find the intersecting lines
            ObjectList intersectingLines=new ObjectList();
            for(int j=0;j<aboveWalls.length;++j){
                    if(newWalls[i].intersectsLine(aboveWalls[j])) intersectingLines.add(aboveWalls[j]);
            }
            if (intersectingLines.length()==0)
                    wl.add(newWalls[i]);
            else{
                    //Find out the distances from each endpoint to the intersecting lines.
                    double[][] dists= new double[2][intersectingLines.length()];
                    Line2D.Double[] intLines=new Line2D.Double[intersectingLines.length()];
                    int j=0;
                    for (Iterator iter=intersectingLines.iterator();iter.hasNext();){
                            intLines[j]=(Line2D.Double)iter.next();
                            dists[0][j]=intLines[j].ptLineDist(newWalls[i].getP1());
                            dists[1][j]=intLines[j].ptLineDist(newWalls[i].getP2());
                            j++;
                    }
                    //Find the Points of Intersection
                    Point2D.Double[] pointsOfIntersection
                            =new Point2D.Double[intersectingLines.length()];
                    double xdiff=newWalls[i].getX2()-newWalls[i].getX1(),
                            ydiff=newWalls[i].getY2()-newWalls[i].getY1(),
                            ratio;
                    for(j=0;j<pointsOfIntersection.length;j++){
                            ratio=dists[0][j]/(dists[0][j]+dists[1][j]);
                            pointsOfIntersection[j]=new Point2D.Double(
                                    newWalls[i].getX1()+xdiff*ratio,newWalls[i].getY1()+ydiff*ratio);
                    }
                    //Find out what point of intersection is closest to the first endpoint
                    double minLen=Double.POSITIVE_INFINITY;
                    int minPointIndex=-1;
                    for(j=0;j<pointsOfIntersection.length;j++){
                            if(newWalls[i].getP1().distance(pointsOfIntersection[j])<minLen){
                                    minLen=newWalls[i].getP1().distance(pointsOfIntersection[j]);
                                    minPointIndex=j;
                            }
                    }
                    //Make a line from the first endpoint to that intersection, add it to wl
                    wl.add(new Line2D.Double(newWalls[i].getP1(),pointsOfIntersection[minPointIndex]));
                    //Make a note of the index of that line for removal
                    int lineToBeRemovedIndex=minPointIndex;
                    //Find out what intersecting line is closest to the second endpoint
                    minLen=Double.POSITIVE_INFINITY;
                    minPointIndex=-1;
                    for(j=0;j<pointsOfIntersection.length;j++){
                            if(newWalls[i].getP2().distance(pointsOfIntersection[j])<minLen){
                                    minLen=newWalls[i].getP2().distance(pointsOfIntersection[j]);
                                    minPointIndex=j;
                            }
                    }

                    if(minPointIndex==-1){
                            return;
                    }

                    //Make a line from the second endpoint to that intersection, add it to w;
                    wl.add(new Line2D.Double(newWalls[i].getP2(),pointsOfIntersection[minPointIndex]));

                    //Remove both lines from intersectingLines
                    intersectingLines.remove(intLines[minPointIndex]);
                    intersectingLines.remove(intLines[lineToBeRemovedIndex]);
                    //If intersectingLines is empty, we are done.
//else there are some line segments we need to take out
                    if(intersectingLines.length()!=0){
                            //Find the points of intersection for the other lines
                            pointsOfIntersection
                                    = new Point2D.Double[intersectingLines.length()];
                            j=0;
```

```
                        for(Iterator iter=intersectingLines.iterator();iter.hasNext();){
                                Line2D.Double l=(Line2D.Double)iter.next();
                                ratio=l.ptLineDist(newWalls[i].getP1())/
                                        (l.ptLineDist(newWalls[i].getP1())+l.ptLineDist(newWalls[i].getP2()));
                                pointsOfIntersection[j]=new Point2D.Double(
                                                newWalls[i].getX1()+(newWalls[i].getX2()-newWalls[i].getX1())*ratio,
                                                newWalls[i].getY1()+(newWalls[i].getY2()-newWalls[i].getY1())*ratio);
                                j++;
                        }
                        //Remove any lines that contain those points from wl.
                        ObjectList temp=new ObjectList();
                        for(Iterator iter=wl.iterator();iter.hasNext();){
                                Line2D.Double l=(Line2D.Double)iter.next();
                                for(j=0;j<pointsOfIntersection.length;++j){
                                        if(l.getX1()==pointsOfIntersection[j].getX() &&
                                                        l.getY1()==pointsOfIntersection[j].getY())
                                                temp.add(l);
                                        if(l.getX2()==pointsOfIntersection[j].getX() &&
                                                        l.getY2()==pointsOfIntersection[j].getY())
                                                temp.add(l);
                                }
                        }
                        for(Iterator iter=temp.iterator();iter.hasNext();){
                                Line2D.Double l=(Line2D.Double)iter.next();
                                wl.remove(l);
                        }
                }//End if there are more than two intersecting lines
        }//End if there are any intersecting lines
}//End if vertsInside==0
else if(vertsInside==1){ //Only one case. We only care about the outer intersect
        //Find the newlines that intersect the oldline
        ObjectList intersectingLines=new ObjectList();
        for(int j=0;j<aboveWalls.length;++j){
                if(newWalls[i].intersectsLine(aboveWalls[j])) intersectingLines.add(aboveWalls[j]);
        }
        //Find the distances from the endpoints to the intersecting lines
        double[][] dists= new double[2][intersectingLines.length()];
        Line2D.Double[] intLines=new Line2D.Double[intersectingLines.length()];
        int j=0;
        for (Iterator iter=intersectingLines.iterator();iter.hasNext();){
                intLines[j]=(Line2D.Double)iter.next();
                dists[0][j]=intLines[j].ptLineDist(newWalls[i].getP1());
                dists[1][j]=intLines[j].ptLineDist(newWalls[i].getP2());
                j++;
        }
        //Find the endpoint outside of the shape
        int outsidePointIndex=-1;
        if(inShape(newWalls[i].getP1(),aboveWalls))
                outsidePointIndex=1;
        else
                outsidePointIndex=0;
        //Find the Points of Intersection
        Point2D.Double[] pointsOfIntersection
                =new Point2D.Double[intersectingLines.length()];
        double xdiff=newWalls[i].getX2()-newWalls[i].getX1(),
                ydiff=newWalls[i].getY2()-newWalls[i].getY1(),
                ratio;
        for(j=0;j<pointsOfIntersection.length;j++){
                ratio=dists[0][j]/(dists[0][j]+dists[1][j]);
                pointsOfIntersection[j]=new Point2D.Double(
                        newWalls[i].getX1()+xdiff*ratio,newWalls[i].getY1()+ydiff*ratio);
        }
        //Find the closest intersecting line to the outside point.
        double minLen=Double.POSITIVE_INFINITY;
        int minPointIndex=-1;
        double pDist=-1;
        for(j=0;j<pointsOfIntersection.length;j++){
                if(outsidePointIndex==0) pDist=newWalls[i].getP1().distance(pointsOfIntersection[j]);
                else pDist=newWalls[i].getP2().distance(pointsOfIntersection[j]);
                if(pDist<minLen){
                        minLen=pDist;
                        minPointIndex=j;
                }
        }

        if(minPointIndex==-1){
```

```
                        return;
                }

                //Make a line from the outside endpoint to that intersection, add it to w;
                //Point2D.Double pointOfIntersection;
                if(outsidePointIndex==0){
                        wl.add(new Line2D.Double(newWalls[i].getP1(),pointsOfIntersection[minPointIndex]));
                }else{
                        wl.add(new Line2D.Double(newWalls[i].getP2(),pointsOfIntersection[minPointIndex]));
                }


                //Remove that intersecting line from intersectingLines
                intersectingLines.remove(intLines[minPointIndex]);
                //If intersectingLines is empty, we are done.
//else there are some line segments we need to take out
                if(intersectingLines.length()!=0){
                        //Find the points of intersection for the other lines
                        pointsOfIntersection
                                = new Point2D.Double[intersectingLines.length()];
                        ObjectList temp=new ObjectList();
                        for(Iterator iter=intersectingLines.iterator();iter.hasNext();){
                                Line2D.Double l=(Line2D.Double)iter.next();
                                ratio=l.ptLineDist(newWalls[i].getP1())/
                                        (l.ptLineDist(newWalls[i].getP1())+l.ptLineDist(newWalls[i].getP2()));
                                temp.add(new Point2D.Double(
                                                newWalls[i].getX1()+(newWalls[i].getX2()-newWalls[i].getX1())*ratio,
                                                newWalls[i].getY1()+(newWalls[i].getY2()-newWalls[i].getY1())*ratio));
                        }
                        //Arrange them in order of distance from the outside point
                        double minDist;
                        Point2D.Double minPoint=null;
                        Point2D outPoint;
                        if(outsidePointIndex==0) outPoint=newWalls[i].getP1();
                        else outPoint=newWalls[i].getP2();
                        for(j=0; j<intersectingLines.length();++j){
        //Arrange the points in order of distance from outPoint
                                minDist=Double.POSITIVE_INFINITY;
                                for(Iterator iter=temp.iterator();iter.hasNext();){
                                        Point2D.Double p=(Point2D.Double)iter.next();
                                        if(outPoint.distance(p)<minDist){
                                                minDist=outPoint.distance(p);
                                                minPoint=p;
                                        }
                                }
                                pointsOfIntersection[j]=minPoint;
                                temp.remove(minPoint);
                        }
                        //Make Lines between the points
                        for(j=0;j<intersectingLines.length()/2;++j){
                                wl.add(new Line2D.Double(pointsOfIntersection[2*j], pointsOfIntersection[2*j+1]));
                        }

                }//End if there are more than two intersecting lines
        }//END IF one vertex is inside
        else if (vertsInside==2){
//If both vertices are inside and there is any points of intersection on the line
                //Find the newlines that intersect the oldline
                ObjectList intersectingLines=new ObjectList();
                for(int j=0;j<aboveWalls.length;++j){
                        if(newWalls[i].intersectsLine(aboveWalls[j])) intersectingLines.add(aboveWalls[j]);
                }
                if(intersectingLines.length()!=0){
                        //Get the points of intersection
                        Point2D.Double[] pointsOfIntersection = new Point2D.Double[intersectingLines.length()];
                        ObjectList temp=new ObjectList();
                        double ratio;
                        for(Iterator iter=intersectingLines.iterator();iter.hasNext();){
                                Line2D.Double l=(Line2D.Double)iter.next();
                                ratio=l.ptLineDist(newWalls[i].getP1())/
                                        (l.ptLineDist(newWalls[i].getP1())+l.ptLineDist(newWalls[i].getP2()));
                                temp.add(new Point2D.Double(
                                                newWalls[i].getX1()+(newWalls[i].getX2()-newWalls[i].getX1())*ratio,
                                                newWalls[i].getY1()+(newWalls[i].getY2()-newWalls[i].getY1())*ratio));
                        }
                        //Arrange them in order of distance from one of the points
```

```
                                double minDist;
                                Point2D.Double minPoint=null;
                                Point2D outPoint;
                                for(int j=0; j<intersectingLines.length();++j){
                        //Arrange the points in order of distance from outPoint
                                        minDist=Double.POSITIVE_INFINITY;
                                        for(Iterator iter=temp.iterator();iter.hasNext();){
                                                Point2D.Double p=(Point2D.Double)iter.next();
                                                if(newWalls[i].getP1().distance(p)<minDist){
                                                        minDist=newWalls[i].getP1().distance(p);
                                                        minPoint=p;
                                                }
                                        }
                                        pointsOfIntersection[j]=minPoint;
                                        temp.remove(minPoint);
                                }
                                //Make Lines between the points
                                for(int j=0;j<intersectingLines.length()/2;++j){
                                        wl.add(new Line2D.Double(pointsOfIntersection[2*j], pointsOfIntersection[2*j+1]));
                                }
                        }
                }
        }//End for(over all newWalls)
}
private void extrudeDown(int currentFloor, int numFloors){
        //Copy the currentFloor to numFloors underneath
        Line2D.Double[] temp=walls[currentFloor];
        for(int i=currentFloor-1;i>=currentFloor-numFloors;i--){
                walls[i]=temp;
        }
}
//Private Helper Methods
private boolean inShape(Point2D p, Line2D[] w){
        /*
         * inBuilding tests whether a point is in the interior of a building by 'shooting a ray'
         * in one direction and counting the number of walls that it intersects.  If the number
         * of walls is odd, then the point is in the building
         */
        int count=0;
        for(int i=0;i<w.length;++i){
                if(w[i].intersectsLine(new Line2D.Double(p.getX(),p.getY(),Double.POSITIVE_INFINITY,p.getY()))){
                        ++count;
                }
        }
        if (count%2==0)
                return false;
        return true;
}
private int compareD(double d1, double d2){return java.lang.Double.compare(d1,d2);}
private void draw(Line2D.Double[][] l){
        for (int i=0; i<l.length; ++i){
                System.out.println("L["+i+"]:");
                for (int j=0; j<l[i].length; ++j){
                        System.out.println("  "+lineToString(l[i][j]));
                }
        }
}
private void draw(Line2D.Double[] l){
        System.out.println("Drawing a Line2D.Double[]");
        for(int i=0;i<l.length;++i){
                System.out.println("  "+lineToString(l[i]));
        }
}
private String lineToString(Line2D l){
        String s ="("+l.getX1()+","+l.getY1()+") -> ("+l.getX2()+"," +l.getY2()+")";
        return s;
}
private static class ObjectList{
        public ObjectList(){
                end=new Node(null,null,null);
                end.next=end;
                end.prev=end;
        }

        public void add(int where, Object o) {
                Node rover=end;
```

```
                for(int i=0;i<where && rover.next!=end;++i,rover=rover.next);
                Node tmp=new Node(rover,rover.next,o);
                rover.next.prev=tmp;
                rover.next=tmp;
                len++;
        }
        public void add(Object o){
                add(0, o);
        }
        public Object remove(int where) { //1 is the head
                Object ret=null;
                Node rover=end.next;
                for(int i=0; i<where && rover.next!=end; ++i){
                        rover=rover.next;
                }
                if(rover==end) return null;
                ret=rover.data;
                rover.next.prev=rover.prev;
                rover.prev.next=rover.next;
                len--;
                return ret;
        }
        public Object remove(Object o){
                return remove(indexOf(o));
        }
        public void clearList(){
                end.next=end;
                end.prev=end;
        }
        public int indexOf(Object o) {
                int ret=0;
                Node rover;
                for (rover=end.next; rover!=end && !rover.data.equals(o); rover=rover.next, ++ret);
                if(rover==end) return -1;
                return ret;
        }
        public int length(){
                return len;
        }
        public boolean isEmpty() {
                return end.next==end && end.prev==end;
        }
        public Iterator iterator(){
                return new MyIterator();
        }

        private Node end=null;
        private int len;
        public class MyIterator implements Iterator{
                private Node rover=end.next;
                public boolean hasNext(){
                        return rover!=end;
                }
                public Object next(){
                        Object ret=rover.data;
                        rover=rover.next;
                        return ret;
                }
                public void remove(){
                        rover.prev=rover.prev.prev;
                        rover.prev.prev.next=rover.prev.next;
                        rover=rover.next;
                }
        }
        public class Node{
                public Node(Node p, Node n, Object d){
                        prev=p;
                        next=n;
                        data=d;
                }
                public Object data;
                public Node prev, next;
        }
    }
}
```

A.3 Building Shapes

This is the generator and interface needed by the A-Type Wall Generation Algorithm

A.3.1 BuildingShape.java

```
package buildingShapes;

import java.awt.geom.Line2D;

public interface BuildingShape {
	Line2D.Double[] getWalls();
	void scale(double x, double y);
	void translate(double x, double y);
	void sysPrint();
}
```

A.3.2 BuildingShapeGenerator.java

```
package buildingShapes;

import java.util.Random;

public class BuildingShapeGenerator {
	//If you want to add a shape - add a private member, add 1 to numShapes, and add a case
	//Constructor and Public Methods
	public BuildingShapeGenerator(){}
	public BuildingShape getBuildingShape(Random rgen){
		//Random rgen=new Random();
		int shapeNum=rgen.nextInt(numShapes);
		switch(shapeNum){
			case 0: return rectangle;
			case 1: return diamond;
			case 2: return triangle1;
			case 3: return triangle2;
			case 4: return hexagon;
			case 5: return octagon;
		}
		return null;
	}

	//Private Members
	private int numShapes=6;
	private Rectangle rectangle = new Rectangle();
	private Diamond diamond = new Diamond();
	private Triangle1 triangle1 = new Triangle1();
	private Triangle2 triangle2 = new Triangle2();
	private Hexagon hexagon = new Hexagon();
	private Octagon octagon = new Octagon();
}
```