Computer Science Department

Computer Science Honors Theses

Trinity University *Year* 2005

Algorithmic Beauty of Buildings Methods for Procedural Building Generation

Jess Martin
Trinity University, jess.martin@trinity.edu

The Algorithmic Beauty of Buildings
Methods for Procedural Building Generation

Jess Martin

Abstract

As virtual reality simulations, video games, and computer animated movies become more prevalent, the need arises to generate the content—the three-dimensional models—via an algorithm rather than crafting them by hand. Previous research in the area of procedural building generation has focused merely on the external appearance of commercial buildings. These methods are unsatisfactory for certain applications due to the lack of a walk-through feature. A new algorithm is proposed to generate residential units with realistic floor plans based partially on the architectural observations of Christopher Alexander. Results for the algorithm display real-time performance and a resemblance to real home floor plans. Also, a complex algorithmic framework for generating hyper-realistic residential units is described, along with algorithms that operate within the framework to generate more realistic residential units. The results of these two methods of residential unit generation are analyzed and the implications of this analysis is discussed. Future research areas are also suggested.

Acknowledgments

Many people have played a part in the process that led to the completion of this thesis. I would like to thank Dr. Pitts for giving me the opportunity to conduct interesting research. I would like to thank Bobby Rubio for helping me work out the simple algorithm. I would like to thank Dr. Lewis for helping to evolve the original algorithm to the complex algorithm. And finally, I would like to thank my wife Colleen for understanding on those nights when I had to sleep in the lab.

The Algorithmic Beauty of Buildings

Methods for Procedural Building Generation

Jess Martin

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

November 15, 2004

Thesis Advisor

Department Chair

Associate Vice President
for
Academic Affairs

The Algorithmic Beauty of Buildings

Methods for Procedural Building Generation

Jess Martin

Contents

List of Figures

Chapter 1

Introduction

Virtual reality has long fascinated computer scientists. The ability to recreate reality in a computer is an intoxicating idea. But like so many early dreams of the computer age, the fruits have not yet come. Recently, however, the continued upward trend of processor speeds and graphics card technology has made hardware capable of rendering complicated virtual scenes available to all. Along with this advanced hardware have come renewed hopes for virtual reality.

1.1 About Virtual Reality

When recreating reality on a computer, one can either create it manually or by developing an algorithm and letting the computer recreate it. Consider for a moment the advantages of creating via an algorithm. A recent film by Pixar, Monsters, Inc. perfectly underscores the importance of algorithmic simulation:

> "Sulley, a star of the film, is an 8 foot tall horned monster with a 700 pound body covered in blue-green hair. Having animators animate his hair by hand would have been an impossible task. Developing hair simulation software that

> can control hair movement was the answer. [Pixar] is a big fan of their new simulation software. It allowed the animators to spend more time on the performance of their characters. "That's where I want to see our animators spending their time. It was a terrific improvement." In Toy Story 2 (A previous movie by Pixar) Al's shirt and pants had to be moved by hand. "I hate animating things like that. You have to spend a lot of time on it and if it is done correctly no one will ever look at it. It doesn't help the film develop the story or provide entertainment. Having a computer do that grunt work is a great improvement." [2]

Similarly, complex simulations will come to incresingly rely on algorithmic methods to populate and lay out the environment. It would be inefficient and, in some cases, impossible for an artist to manually create and place the details of large simulated scenes. Video games, computer animated films, and simulations will benefit from the advancement of these technologies.

1.1.1 City Modeling

A key component of modern reality is the city. If virtual reality is to progress to allow complicated simulations, the city must be simulated as well. Prior research [8][5][7][4][3][6] in this field has focused primarily on the generation of cities, mainly metropolises. Little work has been done on modelling rural communities. Furthermore, the buildings modeled in these cities are often the secondary focus of the research; the primary interest is in the proper layout of the city. Secondarily, appropriate-looking buildings should occupy the proper spaces in the city. The buildings featured in current city generation research are large to small facades of commercial buildings.

1.2 Modeling Buildings

Using a computer to model the representation of a building is nothing new: architects design buildings using CAD software, computer artists model buildings for use in film, video games use buildings as setting. However, only recently has the responsibility for constructing the model been placed on computer software.

1.2.1 Previous Approaches

There have been a few previous attempts at creating software that can construct models of buildings, all of which fall into two categories, photogrammetric and procedural.

1.2.2 Photogrammetric Building Generation

This category of generating buildings draws upon still photographs and edge-finding algorithms to recreate a building. Some approaches use a single aerial photo, others use multiple aerial views from different angle. In any case, photogrammetric approaches require the availability of a fairly clear and detailed image of an area. It should be noted that these methods can produce fairly realistic results, given an excellent set of inputs. However, this method could not be used for real-time generation of non-existent virtual cities, and is more useful for modeling existing cities. Furthermore, this method does not produce buildings that have floor-plans, and thus produces non-traversable buildings.

1.2.3 Procedural Building Generation

The other major category of building generation is more interesting for its diversity of implementations. This method relies strictly on some sort of algorithm to generate the building, usually from a few or more real-world parameters, rather than relying on a pho-

tograph or pre-existing model. This method is more flexible in that in can create arbitrary buildings, new buildings that could not be created photogrammetrically. The weakness of this method is the strength of photogrammetric: this method cannot recreate pre-existing buildings. The quality of the output of this method, unlike the former method, depends almost entirely on the quality of the algorithm rather than the quality of the inputs.

Several of the papers mentioned in the City Modeling section chose this method for generating the buildings that populated their cities. We will now examine a few of these methods in detail.

The first method is discussed in "Procedural Modeling of Cities"[6] by Müller and consists of using a stochastic, parametric L-System[1] to generate the geometry of the buildings. The L-System primitives are geometric operations such as scale and move, as well as geometric templates for roofs and other geometric features of the buildings. See Figure 1.1 for an example of this method.

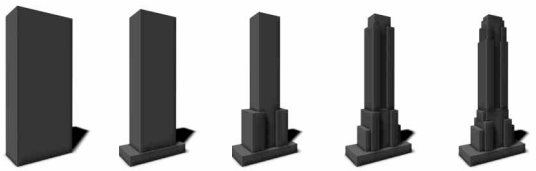

Figure 1.1: **Müller's method of constructing a building using an L-System**

A second method is discussed in "Real-Time Procedural Generation of 'Pseudo Infinite' Cities"[3] by Greuter, et al. Using this method, a building is built up, or more accurately, built down, in sections. Each section is composed of a unique extruded floor-plan. The algorithm starts with the roof, defines a floor-plan by scaling and rotating random shapes,

[1]L-Systems, also known as Lindermayer Systems after their creator are a mathematical construct that consists of a formalized grammar and production rules on that grammar to generate strings. The strings are then parsed into realistic object, most frequently vegetation. For more information, see [**?**]

and then extrudes it a certain height. The next iteration may or may not add a random shape. The algorithm continues until the desired height has been reached. See Figure 1.2 for an example of this method.

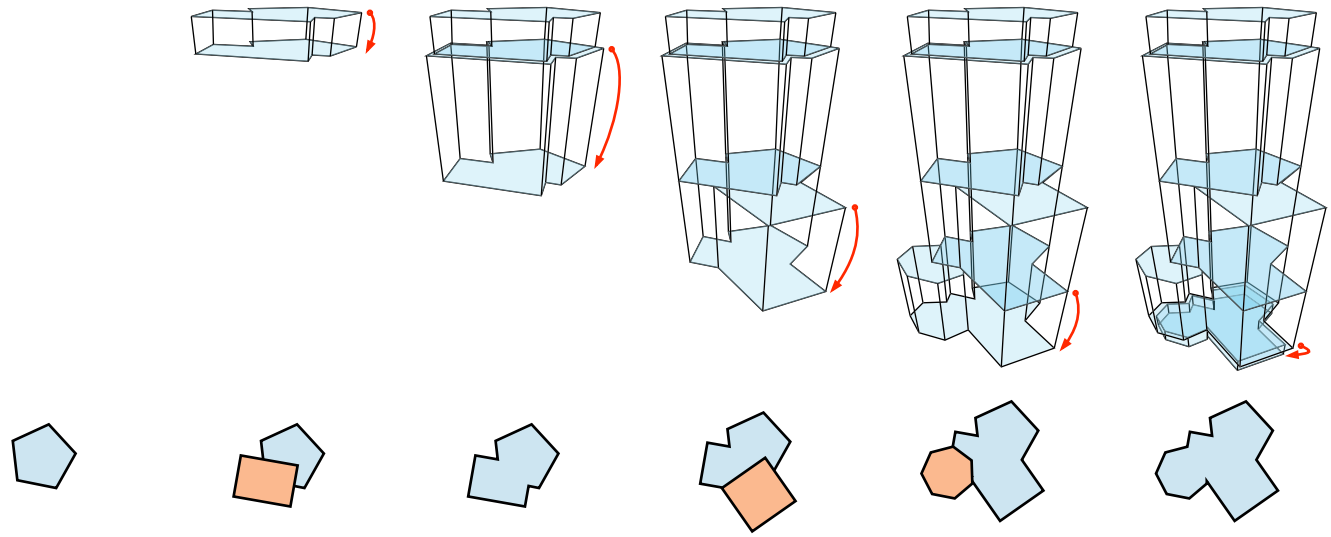

Figure 1.2: **Greuter's method of constructing a building using random extruded shapes**

Each of these methods offers similar benefits. Buildings generated by each of these algorithms have a high degree of visual complexity. Furthermore, when combined with a sufficiently complex texturing method, they each produce buildings that closely resemble office buildings one would find in many of the large cities in America. Also, each of these algorithms is fast: fast enough to generate an entire city worth of buildings in a small amount of time.[2]

That being said, each of these methods suffers similar drawbacks. First, as already

[2]For performance results, see the individual papers [3][6]. In summary, Greuter's method can render 500 buildings at 30 frames/second while Muller's method can generate 13,000 buildings in about 10 minutes.

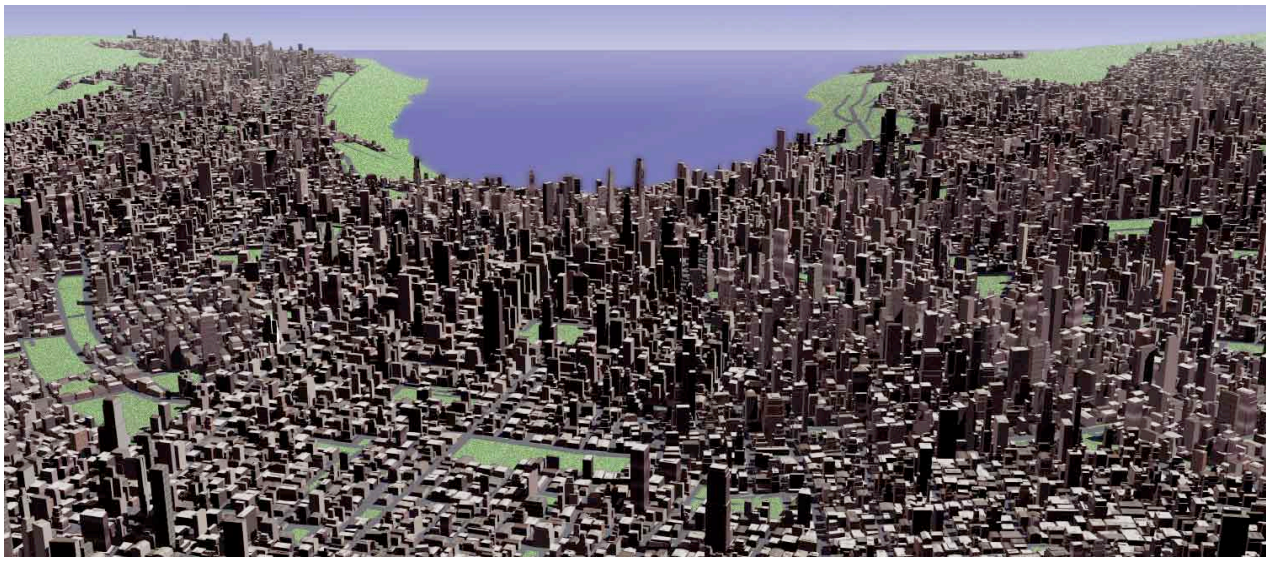

Figure 1.3: **A city generated by Müller's method. Note the skyscrapers and urban setting.**

mentioned, each of the methods focuses on creating commercial buildings, primarily urban commercial buildings. This limits the use of this algorithm to recreating large urban areas. Admittedly, each of these algorithms could be adapted to different types of buildings, but this would require hand-modification of the algorithms and would essentially require a new algorithm altogether. Secondly, and more importantly in terms of this thesis, both of these algorithms generate facades, that is, non-traversable buildings. The buildings are composed of numerous textured blocks without interior floor plans. These methods of generating buildings would then only be appropriate in simulations that did not allow one to enter buildings but merely to walk around the city. This is addressed by Müller and Pascal: "Although a large variety of building types can be generated this way [the lack of walk-through] is a limitation of the system, as the functionality of the buildings can not be represented using only these simple rules"[6, 6]. By "functionality of the building" it can be

assumed that the author is referring to the actual use of the building, say by its occupants, or determined by the internal floor plan of the building.

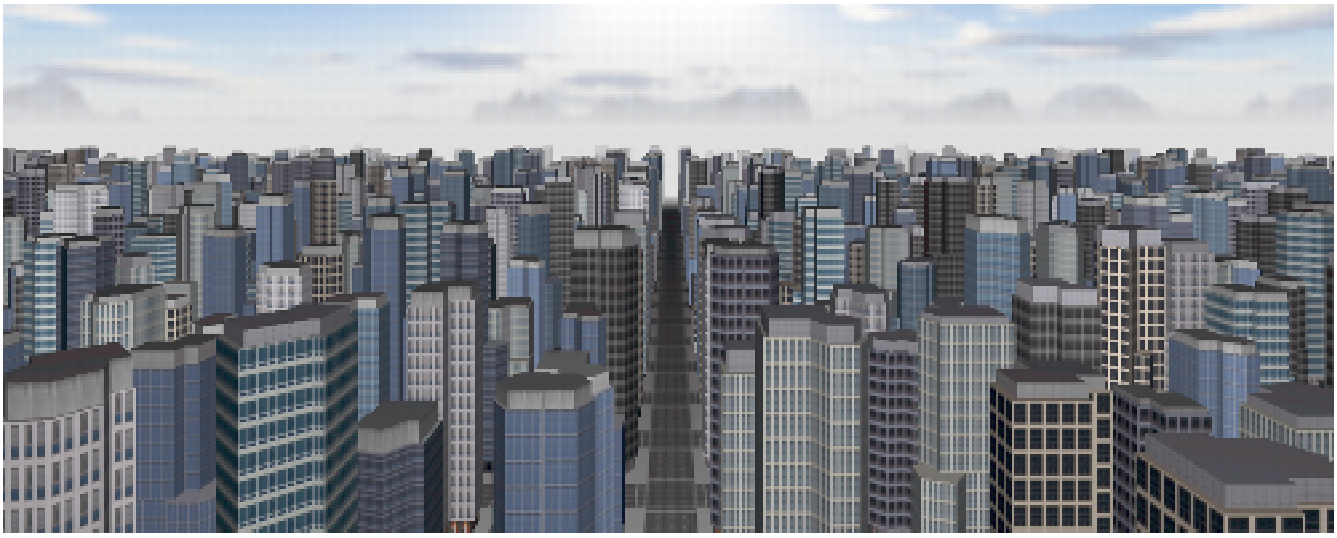

Figure 1.4: **A city generated by Greuter's method. Note once again the dominance of skyscrapers.**

Nevertheless, each of these algorithms offers insights into methods of generating buildings that will be helpful in developing a more complex system.

1.3 An Architectural Approach

What most of the previous algorithms overlook when constructing the building is the architectural component. By forsaking actual design for external visual accuracy, the algorithms limit the range of suitable applications. Admittedly, fast and simple serves the purposes of each algorithm's author, allowing them to create semi-realistic buildings and focus on the city that is generated. However, any research that focuses more specifically on the buildings constructed must take into account more architectural theory in order to be suc-

cessful. The accepted hallmark text for architectural patterns is Christopher Alexander's A Pattern Language[1]. Both authors of the previously mentioned procedural generation papers acknowledge Alexander's contribution, though each in different ways. Müller opines,

> "Alexander et al. describe a pattern language, which consists of over 250 relevant patterns for the successful construction of cities, buildings and houses. They range from very general patterns like "Ring Roads" to very specific ones like "Paving with cracks between the stones. Since these patterns are not formalized, they cannot be used in the automatic creation process of an urban environment.""[6]

Greuter rebuts,

> "Alexander describes construction patterns for the methodical creation of interior and exterior design of cities, buildings, streets and gardens in various levels of detail. Although these patterns are not organized in a format that can be directly utilized by computer software, they do provide a useful guideline to identify significant parameters that govern the visual appearance of objects and structures."[3]

This author tends to agree with Greuter's analysis, and significant thematic choices in the algorithm will be based on patterns and observations laid out by Alexander. Thus, this research marks a more formal architectural look into the structure of buildings in order to build them.

1.4 Assumptions

Before beginning to discuss how the algorithms work, a few methods and terms need to be developed to assist. These assumptions form the platform on which the research stands.

1.4.1 Procedural

Between the two approaches to building generation, procedural was preferred because of its speed and freedom from reliance on images. Procedural generation also is more flexible in that it can generate new buildings, rather than being constrained to pictures of pre-existing buildings.

1.4.2 Parameterized

Furthermore, rather than having an algorithm that simply produced random buildings, real world parameters would allow buildings to be reproducible and useful in any context that could supply values for the parameters. For this reason, it is also desirable to have a minimal set of complete parameters to control the production of the buildings.

1.4.3 Residential Units

After examining the types of buildings generated by most of the available algorithms, it should be noted that none of the previous algorithms dealt with residential buildings. For this reason, this thesis aims to construct a residential building, specifically a residence or home. Residential units were chosen for several reasons. First, the subject is largely lacking any serious research. This allows the author to fill in a needed gap in the field of virtual reality. Second because residential units tend to by multi-purpose and possess far greater variety than do commercial buildings, the complexity of an algorithm required to generate residential units is likely greater than an algorithm to generate commercial buildings. As for types of residential units, this algorithm is only designed to generate single-family residential units. Apartments and hostels, also known as multiple family unit dwellings, will not be dealt with.

1.4.4 Traversable

The single decision that sets this research apart from the existing procedural building generation research is the decision to make the buildings traversable, with realistic internal floor-plans. All of the previous research [3][6] had merely produced facades, textured externals that resembled buildings. For several reasons, it would be desirable to have buildings that were traversable. Primarily, it broadens the range of applications for the algorithm. A traversable procedural building generation algorithm could be used not only at the city level to create a fully traversable city, but also on a smaller scale to create a more localized simulation featuring, say, a single home. Furthermore, the algorithm is more adaptable for video games and computer generated animation applications, as the same algorithm can be used for both traversable and non-traversable buildings, rather than having to model the traversable buildings by hand.

1.4.5 Architectural-Period Specific

Christopher Alexander points out that "every society which is alive and whole, will have its own unique and distinct pattern language"[1, xvi]. In other words, given any two cultures, the patterns used to construct the buildings, and thus the buildings themselves, are not the same. This point must be emphasized for this thesis. The buildings which will be generated by the algorithm will necessarily be buildings bound to a particular time and culture, namely the American culture at the present time.

1.5 Goals

In setting out to create an algorithm to generate traversable residential units, two main goals were emphasized: real-time performance and believability.

1.5.1 Real-Time

To offer maximum flexibility and usability, the algorithm should be as fast as possible, ideally fast enough for the buildings to be constructed and displayed in real-time. This would, once again, broaden the range of applications of such an algorithm to allow for on-the-fly generation of houses depending on the needs of the simulation. This goal shall be deemed reached if the algorithm is able to generated buildings fast enough to populate a fair sized city without causing performance to slow considerably.

1.5.2 Believability

One substantial benefit witnessed in the other procedural algorithms for building generation was the realistic appearance of the generated buildings. In the case of those algorithms, the believability was largely attributable to the proper appearance of the building externals and the excellent texturing. Since this algorithm focuses not only on the appearance of the external of the building, but the inside as well, believability will take on a different definition. For the sake of this algorithm, believability will entail how realistic the floor plan of the building is in comparison to a real residential unit. The aim of the research is to generate floor plans that as closely resemble modern houses as possible.

Chapter 2

A Simple Method

This chapter describes the first attempt at creating a procedural building generation algorithm, favoring simplicity and speed over flexibility and realism.

2.1 Observations

Before describing this simple algorithm, several distinctions must be noted that will provide the basis of the method.

2.1.1 Public versus Private

Christopher Alexander's observations of residential units provide a useful distinction when creating an algorithm for generating homes. Alexander observed that in most homes, rooms can be broken down into public and private rooms. Common public rooms include living rooms, dining rooms, kitchens, dens, and so on. Common private rooms include bedrooms, bathrooms, studies, libraries, sitting rooms, and so forth. There are sometimes ambiguities about the function of a room, but in general this distinction is applicable. Most importantly,

this distinction gives a handle on the way an algorithm can construct the residential unit in a procedural manner. This distinction will crop up all over the place in the algorithm and is possibly the single most instrumental unit of classification in the algorithm.

2.1.2 Affluence and Population Density

To satisfy the assumption that this will be a parameterized algorithm, the method must require only the most important parameters that would allow fine-grained control over the buildings that were constructed without inhibiting the builder. Surprisingly, and in part because of the assumption of architectural-style uniformity, the parameters to supply the algorithm turn out to be fairly simple.

Obviously, one of the key parameters that determines the size of a house (number of rooms, square footage, et cetera) is the amount of money that the person buying or building the house has to spend. This can be simplified into the single parameter of affluence, which encompasses land value, wealth of the builder and so forth into one measure. This parameter will be especially important for city generation algorithms that intend to use this algorithm to populate their cities with buildings. In this way, they can control the size and grandiosity of the houses in an area by adjusting the affluence up or down accordingly.

Affluence alone does not determine the size of a residential unit. In large cities, houses differ if they occur near to downtown or in the suburbs. Houses near an urban center often have less yard space, possibly also having more floors rather than spread out on a larger lot. On the other hand, houses in the suburbs often have large yards and can sprawl out across larger lots. There are other more subtle differences. This parameter will also be useful to allow houses to differ appropriately across the different areas of the city.

These parameters also interact with one another in important ways to cause different size houses to be generated.

2.1.3 Pseudo-Random Number Generation

This algorithm, and any procedural algorithm, would produce fairly similar results, given that there are only two parameters, without the assistance of a pseudo-random number generator (PRNG). To greatly reduce the possible similarity between generated residential unit floor plans, a PRNG is used in most steps to introduce some variability. The PRNG used is found in the Java standard library.[1] To ensure that the results are reproducible, the PRNG is seeded with a seed derived from the position of the house to be built.

2.2 Residential Unit Builder: The Algorithm

Finally, it is time to take a look at the initial version of the algorithm. It consists mainly of six steps.

1. Add Plot
2. Add Front Door
3. Add Social Rooms
4. Add Private Rooms
5. Remove Overlap
6. Add Doors

Each of these steps will be detailed below. The algorithm requires the two parameters, affluence and population density, and the size of the lot to build the house on as inputs.

[1]The pseudo-random number generator "uses a 48-bit seed, which is modified using a linear congruential formula" according to the javadocs for java.util.Random. For more information, see Donald Knuth's explanation.

2.2.1 Adding the Plot

The first step to the algorithm is determining a rough bounding box for the house. The plot serves as the area where the house will be constructed and is placed on the lot. This step is dependent on both affluence and population density. In short, areas of high population density will have less yard space, and therefore the lot will be more completely covered by the plot. Areas of low population density will have more yard space. Lots with a higher affluence will yield houses with more square footage, so will have a larger plot size. Lots with lower affluence will have smaller plots.

After the square footage of the home is determined by the affluence and population density and the lot size, the plot is placed on the lot directly in the center of the lot and then is randomly adjusted towards the front or the back by some amount.

2.2.2 Adding the Front Door

The next step begins the placement of parts of the house. This step is crucial because the rest of the house has to be placed based on it. In assessing what step should come first, what is needed is a feature common among all possible floor plans generated by this algorithm. After eliminating every element of a house that could vary, the only true invariant is an entrance. Every house, from the poorest shack to the largest mansions, shares the common feature of a door. In most cases, the door also occurs on the front of the house. Thus, the front door was chosen as the first step to begin the construction of the house.

The front door is first constructed from a set of constants governing the size of a front door. In early versions of the algorithm, the front door was placed at the very front of the plot. This was soon realized to be inappropriate after studying several dozen floor plans. Thus, the door is displaced away from the front of the plot by a random amount. Because

the door of a house is rarely centered on the plot, the door is then adjusted left or right a random amount as well.

2.2.3 Adding the Social Rooms

This step in the algorithm requires plotting the range of houses that could possibly be constructed using this algorithm. In the most extreme case of poverty, where the aforementioned affluence parameter is zero, the algorithm must still produce a rational result: something resembling the poorest houses in the world. In India and China, and other places where there is extreme poverty, those who do have houses may have only a single room at their disposal for an entire family. This room serves as bedroom, living room, and kitchen, serving both social and private functions. So, according to the social versus private classification, the poorest homes in the world are composed of a single social room, which happens to have multiple functions.

For this reason, adding the first social room is a distinguished step unto itself that takes place automatically. If the affluence is below a certain threshold such that the home will be composed only of a single social room, then the single social room takes up all of the area for the house. If the affluence is sufficient such that there will be multiple rooms in the house, the size of the first social room is calculated and adjusted randomly as a normal room.

The shape of social and private rooms are calculated according to the following equations:

$W = \sqrt{A} \pm (rand \times \sqrt{A} - \frac{\sqrt{A}}{2})$, $H = \frac{A}{W}$, $H = height$, $W = width$, $A = area$

This equation ensures that the room's width is never greater than twice its height, and vice versa. This prevents long narrow rooms from being created, which would not resemble realistic house plans.

Next, the decision of where to place the social room must be addressed. According to the definition of social room, multiple people will be using the room at once and it will be a room that functions in a social context. Further, the only piece of the house we have at the moment is the front door. Since the first room to be added is a social room, it makes sense to place this social room adjoining the front door so that the front door opens into it.

The first social room is placed off of the front door and then is adjusted randomly to the left or the right, while checking the bounds of the room to ensure that it remains inside of the plot.

Next, the remaining social rooms must be added. Rules to consider about social rooms are that they should all be connected so that it is never required for a person to walk through a private room to get to a social room. Moreover, we want to prevent the social rooms from being added in a block at the front of the plot and have them instead span from the front to the black of the plot. For these two reasons, we place each successive social room so that it adjoins the previously placed social room.

The social room being placed may actually not fit in the desired space. To prevent this, the algorithm iterates over each social room, starting with the last one placed, and tries to add on all four sides of the previous social room, starting with the rear first. If there is space for the room, then it is placed there. If there is not space for the room, it generates a few different social rooms and checks each for fit. If after five social rooms there is no fit on a certain spot, it moves on to the next spot. In this way, the rooms are placed as far back in the plot as possible before moving closer.

Once the social room has been placed, a slide is performed on the recently added social room to cause its edges to match the edges of the building appropriately. The slide method does not, however, check for overlap of rectangles. In order to make the outside walls of the building more uniform and the algorithm faster, the room can slide over the top of other

rooms. This is resolved in the Remove Overlap step.

```
procedure slide
        if newRoom is left/right of oldRoom then {
                if newRoom is closer to left wall then
                        slide right until flush with oldRoom
                else
                        slide left until flush with oldRoom
        else
                if newRoom is closer to top/bottom wall then
                        slide down until flush with oldRoom
                else
                        slide up until flush with oldRoom
```

After the room has been placed and slid to flush, a connection between the new social room and the room that it added off of is noted for use later in the Adding the Doors step. Then, the next room is added and the sequence starts all over again.

```
procedure addSocialRooms
        while social rooms to add do {
                room := new social room
                if first social room then
                        place room off of front door
                else {
                        placed := false
                        while not placed do {
                                place room off of last social room
                                if room not overlapping other room then
                                        placed := false
                                else
                                        room := new social room
                        }
                        slide(newRoom, oldRoom)
                        setConnection(newRoom, oldRoom)
                }
        }
```

2.2.4 Adding the Private Rooms

This step of the algorithm may not execute if the affluence for the home is too low. In that case, there would only be a single social room and no adjoining private rooms. In the case that there are private rooms to add, the following steps ensue.

For each private room to be added, the same formula that calculated the size of a social room is used to calculate the size of the next private room. An attempt to add the private room to the floor plan is then made, once again starting with the last social room added and starting towards the rear of that room. This fits the observation that most private rooms are adjoining a social room rather than a private room. The steps for successful adding of the room are the same here as in the social room, except that if the private room is not able to be added on any social room, the private rooms are then cycled through, and an attempt to add the room off of a private room is made. Also, if no room can be added to either social or private room, the algorithm reduces the size of the room to be added by 10% and attempts to add it again.

```
procedure addPrivateRooms
        while private rooms to add do {
                room := new private room
                placed := false
                while not placed do {
                        place room off of last social room
                        if room not overlapping other room then
                                placed := false
                        else
                                room := new private room
                }
                slide(newRoom, oldRoom)
                setConnection(newRoom, oldRoom)
        }
```

As in the Add Social Room step, when the room is finished adding, a slide is performed and a connection is noted between the room that was added and the room that was added off of for use in the Add Doors step. After that, the process repeats until all of the private rooms are placed.

2.2.5 Remove Overlap

In the slide step, overlap could be created, as the algorithm does not do bounds checking on that step. This actually has several advantages. As mentioned before, the original intention was to force the outside walls of the building to line up appropriately. It also, as mentioned, speeds things up to not check boundaries when sliding. However, it has the added benefit of increasing the complexity of the internal geometry of the floor plan. If each room were placed so that it did not overlap any other room, the internal floor plan would be extremely blocky because the shapes of the rooms would remain strictly rectangular. Because overlap is allowed, the internal room shapes are more interesting, and the intersections of the rooms more complex, and surprisingly more realistic.

The overlap is removed by starting with the first social room and iterating over all the other social rooms, overwriting all other rooms geometry with the selected room. After all social rooms are cycled through, the private rooms are cycled through, following the same pattern of overwriting the other rooms.

2.2.6 Adding the Doors

The final step of the algorithm is connecting the various rooms that have been added with doors. During the addSocialRooms and addPrivateRooms steps a record was kept of all the rooms and their connecting rooms when each social and private room was added. This step will need to iterate through all of those connections, retrieving each of the pair of rooms

involved in the connection. Next, the relative position of each pair of rooms is determined to discover which wall they have in common. Next, a door is placed along the common wall and checks are made to ensure that the door joins only those two rooms. This is repeated until all connections have been examined and doors have been added to all adjoining rooms listed. Note that this method for adding doors will not usually add doors between private rooms unless they were added off of another private room, which is appropriate. Finally, this step places a back door on the house by finding the farthest back social room and placing the door along the rear-facing wall. If no social room has a rear-facing wall, the back door is added on the rear-facing wall of the farthest back private room.

```
procedure addDoors
    while connections remain do {
        door := new private door
        placed := false
        room1 := connection[1]
        room2 := connection[2]
        while not placed do {
            randomly place door on adjoining wall of room1 and room2
            if door intersects 2 rooms then
                placed := true
            else
                randomly place door on adjoining wall of room1 and room2
        }
    }
```

2.3 Results

The algorithm described can be analyzed by two criteria: performance and believablity. These criteria stem directly from the goals of the experiment to build an algorithm that would generate realistic floor plans in real-time.

2.3.1 Performance

The primary goal of this research was to have the algorithm run quickly enough to be usable in a variety of real-time contexts, with the main emphasis on large-scale simulations. In large scale simulation, hundreds and possibly thousands of buildings would have to be generated at a time to populate a city or virtual world. It is important that the generation of these buildings commences as quickly as possible.

2.3.2 Reduction Factor

One of the steps of the algorithm that is expected to consume a large portion of the time to generate a home is termed the reduction factor. The reduction occurs during the `addPrivateRooms` method. When the plot is full and an appropriately sized private room cannot be generated to fill the available space, the size of the next set of generated rooms is reduced by a percentage, which I call the reduction factor. By default, the reduction factor will be set to 10% and the following tests have a 10% reduction factor. With the reduction factor set at 10%, for every building that is generated the reduction runs approximately five times.

2.3.3 Middle-Class Home Test

To try to test the ability of the algorithm to generate houses quickly, a test was set up that would approximate the average running time of the algorithm for a large number of buildings. The test suite involved generate several hundred to several thousand middle class homes. Middle class homes were chosen because they are the average home that will be generated in any normal context. To generate a middle class home, the affluence parameter that is passed into the algorithm needs to be set to the average value. These tests were

conducted on a Pentium 4 2.4 Ghz with 512 MB RAM running RedHat Linux 8.0. The tests were timed using the JUnit Testing Framework. To target the applications of the algorithm that will not require thousands of buildings, the tests are run generating 100, 500, and 1000-9000 buildings in increments of 1000.

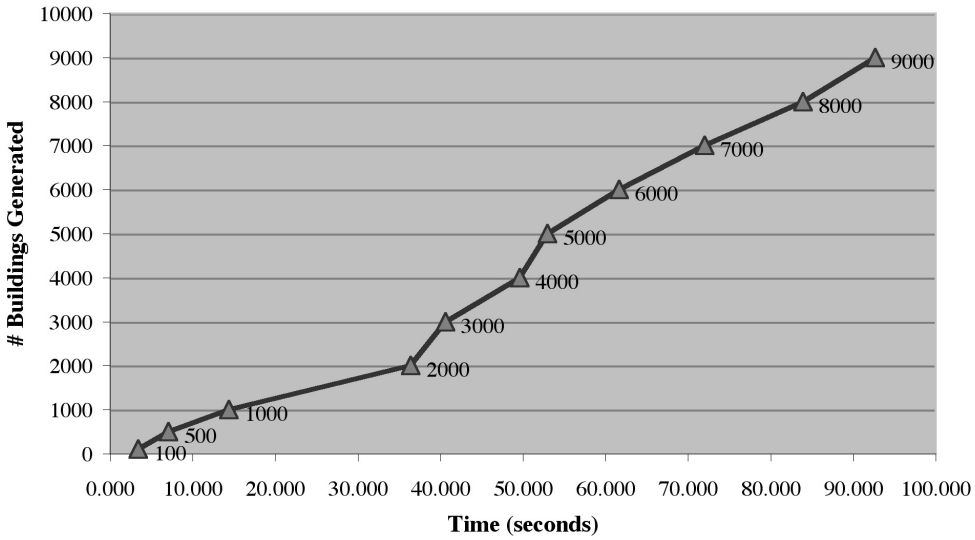

Figure 2.1: **Figure 2.1: Middle-class house test**

Tests indicate that the algorithm scales in a fairly linear fashion. On the test machine, a building costs on average 0.014 seconds to generate with a reduction factor of 10%.

2.3.4 Comparison

Comparing these numbers to the limited figures available in the two previously mentioned city-generation papers is useful to estimate if the algorithm could be used in real-time.

Müller's method created 13,000 buildings in "approximately 10 minutes. The buildings were extruded from the shape of the allotment and automatically textured"[6]. The paper

does not provide a break down of how long each step took, so we are left to guess. However, in comparison, this algorithm can generate 13,000 houses in 46.488 seconds, taking less than 1/10th of the 10 minutes to generate the building. If the extrusion and texturing takes ten times as long as generating the plot, which is a safe assumption, then the algorithms are comparable. Furthermore, since the algorithm described in "Procedural Modeling of Cities" is considered to be a real-time algorithm, this algorithm could also be used for generating buildings in real-time.

In Greuter's "Real-time Procedural Generation of 'Pseudo-Infinite' Cities" the amount of time required to render buildings is never specified. However, the application never generates more than 1000 buildings at a time. The algorithm described here generates 1000 houses in approximately 2.5 seconds. This figure would easily be fast enough to qualify for real-time status.

It is clear that by hypothesizing about the speed of this algorithm versus the incomplete data published in other papers that the algorithm described here is at least in close proximity to real-time rendering. If close is not close enough, there are optimizations that could be performed at the cost of a minute fraction of visual accuracy, mentioned below.

2.3.5 Optimization

In the case of more demanding simulations where speed is of the utmost necessity, the reduction factor could be increased, thus reducing somewhat the number of times it would be required to run, at little cost to the user. The effects of the reduction factor can only be calculated by calculating the actual square footage of the floor plan and comparing it to the expected square footage of the floor plan. The effects of increasing the reduction factor has little effect that can be noticed by observation. It is left up as an excercise for the user to increase the reduction factor arbitrarily until a visual difference can be detected. As

the comparison above illustrates, with a reduction factor of 10%, the algorithm performs adequately for most normal uses.

2.3.6 Believability

Beauty, as they say, is in the eye of the beholder. One of the goals of the algorithm was to create realistic floor plans. "Realistic" can be interpreted loosely and subjective analysis of the created floor plans would not be useful. Therefore, in an effort to make concrete the concept of a "realistic floor plan", floor plans generated by the algorithm will be compared side-by-side against actual floor plans.[2] For the purpose of determining how believable the floor plans generated by the algorithm are, five floor plans of a middle class house (three private rooms, three social rooms) were randomly generated and analyzed against actual floor plans to determine realism. Keep in mind that this is not the Turing test, and the algorithm is not intended to fool a user in to believing that the building was designed by an architect but to bear enough resemblance to a house that it could be passed off as such.

Before examining the images, keep in mind the following general "rules" that the algorithm attempts to enforce. Most were mentioned throughout the previous section but are summarized here for the sake of convenience.

Rule 1 A path should exist through all social rooms.
Rule 2 Every private room should adjoin at least one social room.
Rule 3 The walls of the house should be "roughly" square.
Rule 4 The front door should adjoin a social room.
Rule 5 There should be a door between any adjoining social and private room.
Rule 6 There should not necessarily be a door between any two private rooms.

[2]Floor plans courtesy of Dream Home Source. http://www.dreamhomesource.com

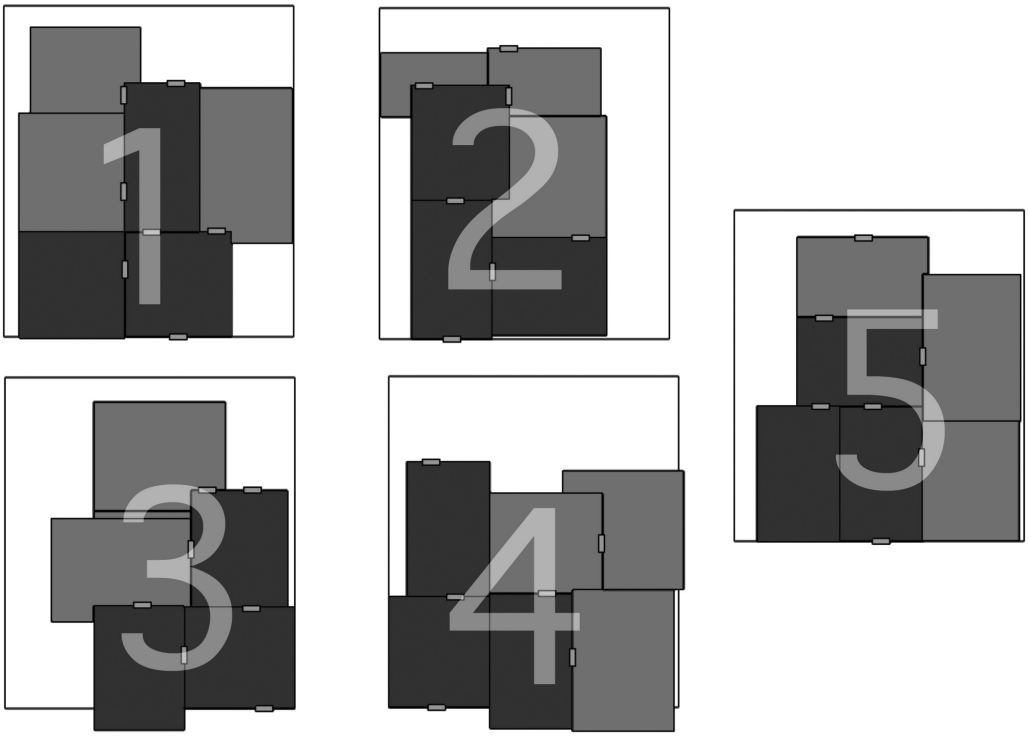

Figure 2.2: **Figure 2.2: Five randomly generated middle-class floor plans**

Figure 2.2 illustrates that most of the time the algorithm successfully generates fairly realistic floor plans. For the majority of the images, most of the rules are preserved. Only image three has serious floor plan problems in that one private room gets truncated.

Moreover, finding floor plans that matched these generated houses was a simple affair. Four out of the five plans bear a strong resemblance to actual floor plans found on Dream Home Source.

2.4 Conclusion

In this chapter, a simple algorithm for generating houses was presented. This method involved a rigid series of steps that led from a set of parameters to a constructed house in one pass. While fleshing out the details of this algorithm, many of the deficiencies of this approach were exposed. For example, the generation of a particular room's dimensions, it's connectedness to other rooms, and it's placement in the house were all determined in the same step. This complicated blend of steps sacrificed flexibility for the sake of expedience. This led to certain rules and behaviors that could not be captured by the algorithm.

The experience gained in working on the simple algorithm yielded insights which led to the creation of a better algorithm. The key distinction noted while developing the simple algorithm was that the connectedness of the rooms in a residential unit is merely a graph that can be constructed independently of the placement and sizing of those rooms. After a graph of the connected rooms is created, a separate algorithm can then be used to determine the appropriate size and specific location of those rooms.

This separation of phases of the algorithm has several key benefits. First, different algorithms can be used to generate the graph or the placement, making it possible to mix and match different algorithms and test the results. Second, each of the steps can be more

focused on the content of that step without getting confused by the rules and constraints of other steps. Therefore, more attention can be paid to enforcing rules and behaviors specific to that particular step.

The next chapters will describe a more sophisticated algorithm that will generate more accurate residential units through the use of the separation of steps just mentioned. Chapter three will discuss an algorithm to create a graph that depicts the connectedness of rooms in a house. The subsequent chapter will discuss an algorithm that will place the rooms to determine their location, and an algorithm to determine the size of the walls.

Chapter 3

Graph Generation Algorithms

The first step in the complex algorithm is to generate a graph that represents the interconnectivity of the various rooms in a house. Generating this graph independently from other parts of the algorithm makes it possible to focus on capturing the proper relationship among connecting rooms. Later, other parts of the algorithm can create the rooms without having to worry about deciding which rooms connect since that information will already be provided in the form of this graph.

3.1 Algorithm Components

The graph is composed of various elements, mainly rooms and connections between rooms. Each room in the graph is represented by a node. A connection (or door) between two rooms is indicated by an undirected edge between two nodes.

3.1.1 Rooms

Nearly all rooms in the graph are classified, as in the previous algorithm, as either public or private. This basic distinction can, even in a simple algorithm, capture much of the behavior of the layout of the modern home. In addition to the public/private classification system, this algorithm will go beyond the previous algorithm in that rooms will not only have a class (public or private) but a type as well. The type will represent the commonly assigned name of the room. In other words, this algorithm will be able to track the difference between a living room and a kitchen. This information will turn out to be particularly useful when decorating the interior of the generated houses. Furthermore, knowing the type of the room will help to determine the appropriate size relative to other rooms during the placement portion of the algorithm. Additionally, each room will have a magnitude associated with it. Rather than have this magnitude be specific in terms of square footage, this magnitude is relative to the sizes of other rooms in the house. This allows the placement algorithm to build larger houses with the same number of rooms while maintaining the appropriate sizing relationship among rooms within that house. See appendix A for a listing of each Room Type and their associated magnitudes and room class.

Non-Terminals

Some rooms will exist in the graph as non-terminal rooms, though there will be very few of these in the final version of the graphs. These non-terminals are best thought of as placeholders, specifying the public or private classification or even the magnitude of the room without specifying which type of room it is. In the final graph, some of the non-terminals will remain in the graph to allow for hallways or stairs. This overcomes one of the primary deficiencies noted in the last algorithm: the lack of logical hallways. Hallways

can now be captured as a non-terminal room that has multiple terminal rooms branching off of it. This will also allow flexibility on the part of the placement algorithm: it can either "terminalize" the non-terminals or implement them as hallways or stairs.

Terminals

Most rooms in the final graph will be terminal rooms, that is rooms that have not only their classification specified as either public or private, but also the type of room and the magnitude of the room. They are termed terminals because they do not need to be specified further as the non-terminals do.

3.1.2 Statistics

Beyond maintaining information about specific rooms and their public/private classification, this algorithm will take into account which rooms are or are not typically located adjacently. A Statistic object will be maintained on each type of room that will contain such valuable information as which rooms the room normally is located adjacent to, which room in particular it must be connected to, and the maximum and minimum number of rooms of that type that are normally located in a house. This information will allow the algorithm to more intelligently place rooms as they might actually occur in a house. See appendix B for a listing of each Room Type and its associated statistics. Figure 3.1 shows basic UML for the Statistics object maintained for each Room.

3.1.3 Rules

In order to allow the graph to grow dynamically and realistically, there are rules that define what room or set of rooms can replace another room on the graph. For instance, one of

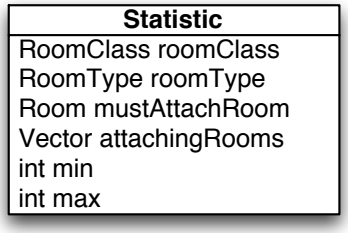

Figure 3.1: **Figure 3.2: UML for a Statistic object.**

the basic rules is the front door can be replaced by the front door attached to a public non-terminal room. This is represented by a rule of the form pictured in Figure 3.2.

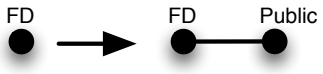

Figure 3.2: **Figure 3.2: Example of a rule.**

Rules, in this form, are specified before the algorithm begins and are collected in a ruleset. A rule can also have multiple right hand side options and probabilities associated with these options.

3.1.4 Magic Number

A magic number is used in place of a square footage or land value figure as used in the simple algorithm. This magic number allows some flexibility in the algorithm in that it can be computed from any set of inputs that may be imagined. This enables the complex algorithm to be effective in a wider range of applications.

3.2 Algorithm Description

The graph generation algorithm operates in four phases. In the first phase, the public rooms are all added without specifying what type of room they are. In phase two, the private subgraphs that branch off of the public rooms are added. In the third phase, the public rooms are converted into specific public rooms. Finally, in stage four, the "stick-on" rooms are added.

Before the four phases are run, the algorithm randomly determines the amount of space in the house that will be used for social rooms and the amount that will be used for private rooms.

3.2.1 Phase One: Social Rooms Added

As mentioned in the previous chapter, a logical first step for any algorithm that generates a house iteratively is to add the front door. After the front door is placed, all of the non-terminal social rooms are added. This is logical because all social rooms must be connected to all other social rooms. If it were not possible to get from one social room to another without passing through private rooms, then the intervening private rooms must actually be public. The non-terminal social rooms are added by following a set of rules, with the front door node as the first node looked at. Replacement then proceeds according to the rules listed in Figure 3.3.

Replacements occur until the social space in the house is as completely filled as possible. Since at this point each room is merely a non-terminal without a specified type and there is no way of knowing the actual size of the rooms, each non-terminal social room is assumed to be an average sized social room. The result of this step is a graph with a front door connecting to a mishmash of interconnected non-terminal social rooms.

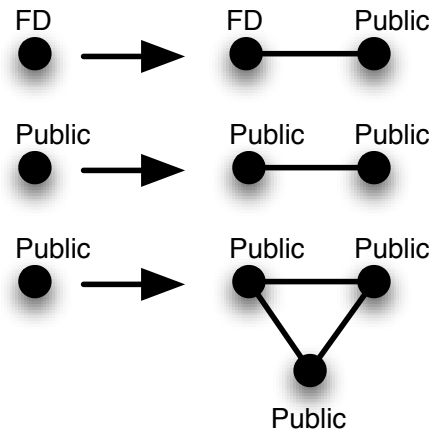

Figure 3.3: **Figure 3.3: Rules that govern non-terminal social rooms.**

3.2.2 Phase Two: Private Subgraphs Added

The next phase involves adding all of the private rooms to the graph. With all of the social rooms already placed in the graph and interconnected, the private rooms can now be connected off of the social rooms. It does not matter that the social room's room types are not specified, as private rooms, or at least hallways to private rooms, tend to have entrances from pretty much any social room in the house.

This process occurs by starting at the room immediately off of the front door, usually a foyer or great room, and walking the graph a random distance and attaching a private non-terminal room to the social non-terminal. The private non-terminal rooms are typically connected to the social room near the front door room with a tendency to cluster around a single node. This behavior is implemented to accurately capture the configuration, seen in many houses, of the private rooms tending to connect primarily to either the foyer or great room, though the rooms are by no means limited to connecting only off of those two rooms. The ruleset is then checked to see what private rooms can be produced from a

private non-terminal room. Rules that are used to determine what a non-terminal private room can be replaced by are list in Figure 3.4.

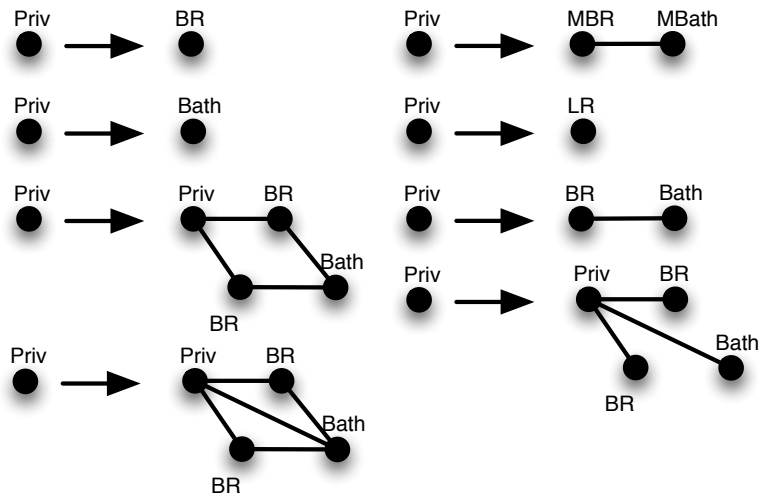

Figure 3.4: **Figure 3.4: Rules that govern the adding of private terminal rooms.**

The proposed replacement is then checked against the statistics to verify that the room can be added to the graph and that there is not another room that is more necessary to the process. These steps are run until the private space in the house is filled.

The results of this phase are a graph with all of the social rooms added and connected but merely existing as non-terminals with clusters of private rooms sprouting off of the non-terminal social rooms.

3.2.3 Phase Three: Converting Social Rooms to Specific Social Rooms

This phase takes place in order to transform all of the non-terminal social rooms that are already in the algorithm into terminal social rooms with both an appropriate room type and a magnitude appropriate to that type of room. This step will allow us to enforce that some rules adjoin each other while others are not adjacent.

The process begins at the room that the front door enters into. This room is usually either a foyer area or a living room. Depending on the size of the magic number, the algorithm chooses to replace this room with either a foyer or a great room, which is usually the largest living room in the house. After replacing this room with either a foyer or a great room, the algorithm continues to replace rooms using the statistics for the room type.

Once a social non-terminal room is located, the replacement occurs in one of the following manners. First, the statistic for that type of room is checked and if there is a room that must connect to that room, then the replacement occurs with that room type. If there is no required room for that room type, the algorithm receives from the statistic the list of social rooms that can attach to the room. From those rooms, the algorithm checks each one to see if there are any rooms in the list that have a minimum amount of rooms greater than zero and that have not reached that minimum in the graph. If such a room is found, the non-terminal room is replaced by that room. Finally, the statistic is checked for any rooms that can be connected to but that are not required by the statistic and status of the graph. In the event that all other rooms are added to the maximum amount, note that their is no maximum number of living rooms that can be added to the graph. Living rooms act as the additional room that can always be added to the algorithm to fill in social rooms after all of the other social rooms have been added. This also mirrors the way that houses work in real life. After all of the necessary social rooms have been added, the rest of the social rooms are all variations on the living room. The flow-chart in Figure 3.4 illustrates the steps by which the appropriate terminal social room is chosen.

The traversal occurs, specifically, by traversing down one random path through the social rooms, starting at the Foyer or Great Room, and proceeding until there are no non-terminal social rooms to be found that link to the room on which the replacement just took place. Then, the graph is search for any room that has a non-terminal social room linking

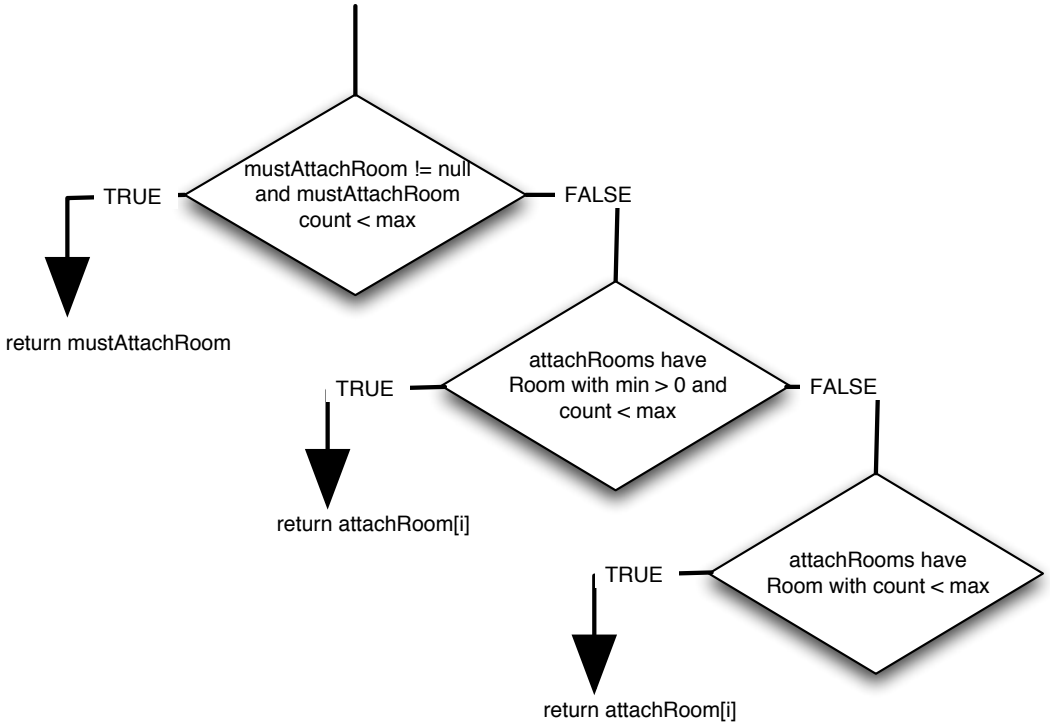

Figure 3.5: **Figure 3.4: Flowchart demonstrating how the appropriate terminal room is chosen from the statistic object.**

to it, and the algorithm traverses that path to its end. When the graph has no more social non-terminals, this phase of the algorithm is complete.

3.2.4 Phase Four: Stick-On Rooms Added

The final phase of the algorithm iterates over each room in the graph and checks to the Statistic object to see if that roomType has any stick-on rooms that are associated with it. Some stick-on rooms are added every time, some are added based strictly on probability, and others are added based on a relationship with the magic number. Some examples of stick-on rooms would be pantries, which are normally stuck on to kitchens, and linen closets, which are normally stuck on to hallways.

Chapter 4

Placement and Wall Algorithms

After the graph of a residential unit has been created, an algorithm or algorithms is needed to take that graph and transform it in to a floorplan, complete with locations and sizes of rooms. Due to the flexibility of the new method of house generation, this goal can be reached in many different ways. One type of algorithm would be to take the graph and use a single algorithm to create the rooms and place them in respect to the other rooms. These algorithms are referred to as placement and wall algorithms, because they both choose a specific room size and place the rooms. Another option would be to have two separate algorithms, one for placing the rooms in an initial configuration and another for establishing the dimensions of the rooms. These algorithms are classed placement and wall algorithms, respectively.

Due to the fact that this portion of the house generation can be executed in several ways, the author will focus on describing a single algorithm of each type in detail and then suggest several alternative methods that have not been implemented but may provide better results upon further testing.

4.1 Placement Algorithms

The placement algorithm acts as an initial seed for those wall algorithms that need the rooms to be laid out intelligently before they can determine the shape and size of each of the rooms. There are many possible ways to place the rooms, each with some obvious advantages and disadvantages up front. However, to truly determine which placement algorithm is most effective, implementation is the best test. A possible method for placement is described below, it's advantages and disadvantages are discussed, and suggestions are made for other placement methods that may be more advantageous.

4.1.1 Push Placement - A Placement Algorithm

The push placement algorithm is a simple way to set out the rooms apart from each other than will act as a seed for a wall algorithm. The idea is to simply "push" each room out from the first room in the house by the magnitude of that room and spacing each room out proportionately. The goal of the algorithm is to space the rooms out quickly according to their magnitude. See Figure 4.1 for an example of an appropriately distributed graph.

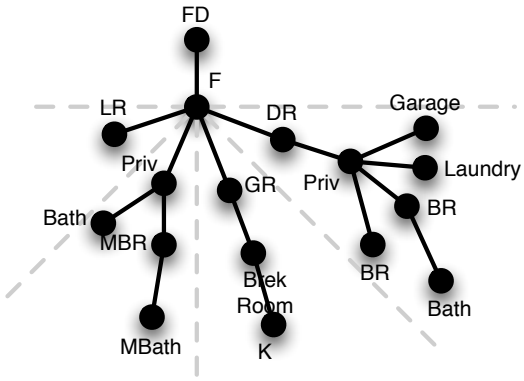

Figure 4.1: **Figure 4.1: Example of a graph that has been placed using the push placement algorithm.**

The algorithm operates by doing a breadth-first search through the graph. For each node that is visited, the algorithm splits the available angle evenly between each of the rooms and displaces each of them by their magnitude. Then each node is visited and the process is repeated. In the case of graphs with no cycles, the result will be a tree-like structure, as seen in above in Figure 4.1. In the case of graphs with cycles, the results will more closely resemble Figure 4.2, a tree with a few backward links.

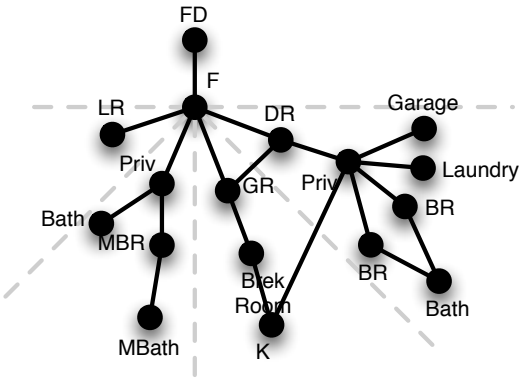

Figure 4.2: **Figure 4.2: Example of a graph with a loop that has been placed using the push placement algorithm.**

The results of this method are then passed on to a wall algorithm which determines the sizes of the individual rooms.

4.1.2 Analysis of Push Placement

The push placement excels in its simplicity by being fairly fast. The breadth-first traversal takes O(V + E) to traverse the graph and push the rooms out. It can be executed fairly quickly and provides a simple initial placement that wall algorithms can then use to determine the sizes of the rooms.

Unfortunately, there are several disadvantages to the method's simplicity. Primarily, the

push placement method places all rooms on a single story. Granted, if the wall algorithm has the ability to pop rooms up to a second floor, this will not be a problem. However, some of the wall algorithms will lack this bit of complexity, restricting the push placement method in conjunction with those limited algorithms to only one story houses. This may be impractical for some particularly large house sizes.

Additionally, the push placement algorithm simply traverses the connecting rooms in order. This may cause problems in the case where two rooms that have rooms that connect to each other or have other rooms that may connect to each other are not in order in the list of connecting rooms. For illustration of the problem, see Figure 4.3. With the current graph generation algorithm, this problem is non-existent since connected rooms are added in order, however, it is important to keep in mind that other algorithms may be used at some point to generate graphs. With that mind, it is important to disclose all possible problems or unexpected side effects that may occur should a different algorithm be used to either generate the graph or create the walls.

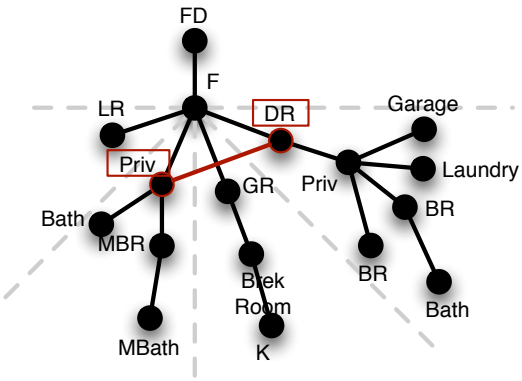

Figure 4.3: **Figure 4.3: Example of a push placed graph with a loop that causes a problem.**

4.1.3 Other Possible Placement Algorithms

The push placement algorithm is, of course, not the only algorithm that could be used to place the rooms prior to their wall creation. There are surely many other ways that the rooms could be creatively placed. A major disadvantage of the push placement algorithm is that it places all the rooms on the same story, effectively limiting the method to one story houses unless the wall algorithm does something tricky. A placement algorithm that avoided this problem would be beneficial, as it seems appropriate that the placement algorithm, rather than the wall algorithm should be concerned with how many stories a house has and which rooms are to be placed on which stories.

For example, a placement algorithm that would work for multiple stories might work as follows. Each room becomes a sphere with a size that matches the magnitude of the room. Each connection between rooms becomes a spring that pulls the two rooms together. The plot of the house, the external walls, then becomes a container in which is dropped the room-spring mesh. The first layer will only have room for so many rooms, at which point to the next layer will begin to fill up. To enforce some of the rules of two story houses, public rooms will made to have more weight than private rooms, thus making it so that private rooms and subgraphs are more likely to end up on the second floor. The final positions of the rooms are recorded once they come to rest and they are fed into any of the available wall algorithms.

4.2 Wall Algorithms

After the graph of a residential unit has been generated and the rooms have been placed, an algorithm needs to run to determine the locations of the walls in the house. Like the other algorithms in the house generation method, there are several ways that could be considered

to create the walls of the rooms.

One possible way to create the walls would be to use an algorithm that treats each of the rooms as an expanding square bubble, with internal pressure equal to the magnitude of the room. In this algorithm, each room would expand until equilibrium is reached. This algorithm could be made more complex by adding springs in between the "bubbles" that would keep the rooms attached to each other so that doors could be added later. It is possible without the springs that the rooms would get moved around so that connections between them were no longer possible. Also, special types of rooms such as stairs and hallways would have to special deformation rules that allow them to bend and stretch within certain parameters.

The true test of each of these algorithms would be implementation and analyzing how they work in conjunction with the different graph and placement algorithms.

4.3 Placement and Wall Algorithms

It is possible that certain algorithms for transforming the graph into a floor plan may work better by incorporating the placement of the rooms and the determination of the walls in the same step. There are several situations that may necessitate an algorithm that combines the two steps. This flexibility is a key strength of the second method for house generation.

Chapter 5

Further Research

Both the simple and the complex algorithms have their place in certain applications. Both algorithms could use some improvement to make the results even more broadly usable.

5.1 Applications

In the Introduction, multiple uses of building generation algorithms were noted. These include simulations which occur in real time and those that are rendered once.

5.1.1 Simulations

Previously, simulations that used building generation algorithms were limited to remaining "in the streets" so to speak. None of the buildings added to the algorithm were able to be traversed. Furthermore, any simulation of this type previously had to feature an urban downtown setting, restricted by the types of buildings that were created. Now these simulations can extend to the suburbs as well. Traversable residential units will hopefully expand the range of possible simulations.

5.1.2 Model Creation

Outside of real time simulation applications, there is also the need for algorithmic building generation to generate static models in areas such as computer games or movies. These algorithms can be used to generate dozens of traversable homes to fill in a city model. Also, the internal of the building can, for the first time, be generated and populated in order to render indoor scenes.

5.2 Improvements

Some of the weaknesses of both the simple and the complex algorithm have been explored throughout the paper. The simple algorithm can generate simple house plans that will pass as real as long as no serious scrutiny is applied. The algorithm itself is fragile and not open to significant changes. The complex algorithm, on the other hand, can already handle fairly complex residential units that far surpass the capabilities of the simple algorithm. Also, because of the flexibility built into the complex algorithm, the algorithm can be greatly changed and improved to handle a wider range of more detailed floor plans.

5.2.1 Multiple Story Houses

Adding the requirement to generate multiple story houses would affect the simple and the complex algorithms in different ways. The simple algorithm would simply not be able to handle them. This is because the simple algorithm acts as a space-creating algorithm, defining the exteriors of the house as it places rooms. Extended up to a second floor, the simple algorithm has no method for filling a pre-defined space.

For the complex algorithm, multiple stories would simply require either more sophisticated placement or wall creation algorithms. One way multiple story houses could be

handled is by using the drop-in-the-box placement method. Furthermore, this would probably allow for arbitrary geometry, that is, different exterior geometry for each floor. Or, multiple story houses could be handled using the "square bubble" growing algorithm by enabling it to "pop up" certain rooms when pressure gets too great on a lower floor. Both of these methods would require adding weight to the rooms so that most public rooms stayed on the bottom floor while private rooms were more likely to "pop up."

5.2.2 Roofs

Roofs are another complexity that would require more intelligent placement or wall creation algorithms for the complex method to handle. With some types of roofs, they can be merely added as an afterthought after the house has been generated. Though the roofs themselves may be non-trivial to determine, at most they represent an extra step that must take place after the house is generated. However, some roof styles and features of roofs actually influence and interact with the internal geometry of a home. Examples include A-frame houses, dormers, and other roofing features.

For the simple algorithm, dealing with roofs is out of the question. For the complex method, with the right combination of algorithms, it is most likely possible to generate houses that intelligently handle the different types of roofs that would influence the floor plan.

5.2.3 Porches and Patios

Currently the rooms that are generated by this algorithm are internal rooms only. The structure of the area around the outside of the house, though important, is not currently considered. Out of all of the features to add, this may be one of the easiest because it does not necessarily require modification of the algorithm, but only an additional step after

the floor plan is generated. It is important to note that while porches and patios may not influence the internal floor plan, they will definitely have an effect on the types of roofs that the home will have, which may in turn influence the internal layout.

5.2.4 Odd Shaped Rooms

Both the simple and complex algorithms currently produce only rectangular shaped rooms. Using the drop-in-a-box method and even a non-rectilinear version of the "square bubbles" wall creation algorithm may be able to recreate arbitrary room geometry in the complex algorithm.

5.2.5 Windows

The focus of both the simple and complex algorithms has been on laying out the rooms intelligently. A part of creating a floor plan, however, is specifying the location and type of windows. This step, while not considered in this thesis, would likely greatly increase the believability of three-dimensional representations of the home. For implementation in the complex method, it seems that windows should be added as a final step that occurs after the walls are created.

Chapter 6

Conclusion

Building and city research have experienced a revival of interest in the past few years. These research projects generate buildings in two different ways: photogrammetrically and procedurally. Photogrammetric methods have serious drawbacks the prevent entirely new buildings from being created. Previous research in the area of procedural building generation has been secondary to the city generating research, often detailed in a small section of a paper focused on city generation. Those researchers used algorithms that generated only the externals of commercial buildings, preventing walk-throughs and locking cities in to one class of building.

This thesis introduces a simple algorithm for procedurally generating houses with realistic floor plans fast enough to be considered real time. It can be used in addition to or as a supplement of existing building generation algorithms currently in use in city generation as it was designed from stage one to be minimally parameterized and fast. The buildings are generated quickly, but because of some of the sacrifices made during the algorithm, do not stand up to deep scrutiny and should only be used to generate houses whose appearance is not required to be perfectly authentic.

Also introduced is a more complex method of generating residential units, which is probably better considered as a framework for the generation of houses. This framework consists of several algorithms that generate the intermediate steps involved in the house creation, from the graph generation, to the graph placement and finally to the wall creation. Each of these steps executed one after another using a particular algorithm at each step would be the framework. This framework is remarkable for its flexibility, extensibility, and ease of use. Also illustrated were a particular graph generation algorithm and a graph placement algorithm. Possible algorithms for wall creation were described.

6.1 Concluding Remarks

It is this author's sincere hope that this research will continue and be extended. In previous algorithms, a vast difference exists between how the computer builds buildings and how the architect builds buildings. By incorporating just a bit of architectural know-how (the difference between social and private rooms) into the algorithm, the results are visually interesting and impressive enough to pass for houses. This is encouraging and it is likely that incorporating more architectural knowledge into an algorithm will generate even more realistic and interesting buildings. Interested parties would be best served by looking first at the work of architect Christopher Alexander.

While many have noted the importance and impressive scope of Alexander's work on patterns in architecture, most have discounted the work's ability to translate to a computer algorithm because of a lack of formalization among the patterns. While some of the patterns do appear based on intuition rather than strict rule and reason, many of the patterns contain practical, implementable advice that could be incorporated into an algorithm, be it building generation or city generation. The author's suggestion would be to

read A Pattern Language and pick several of the patterns that Alexander marks as "a true invariant: in short, that the solution...stated summarizes a property common to all possible ways of solving the stated problem"[1].

Bibliography

[1] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language*. Oxford University Press, 1977.

[2] Karl Cohen. Monsters, inc.: The secret behind why pixar is so good. *Animation World Magazine*, 6(7):6–12, October 2001.

[3] Stefan Greuter, Jeremy Parker, Nigel Stewart, and Geoff Leach. Real-time procedural generation of 'pseudo infinite.

[4] Stefan Greuter, Jeremy Parker, Nigel Stewart, and Geoff Leach. Undiscovered worlds towards a framework for real-time procedural world generation. DAC, 2003.

[5] Norbert Haala, Claus Brenner, and Karl-Heinrich Anders. 3d urban gis from laser altimeter and 2d map data. *Journal of GIS*, 1999.

[6] Yoah I H Parish and Pascal Muller. Procedural modeling of cities. Association of Computing Machinery, 2001.

[7] Y. Takase, N. Sho, A. Sonem, and K. Shimiya. Automatic generation of 3d city models and related applications. *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, XXXIV-5W10, 2001.

[8] C. K. Yap, H. Biermann, A. Hertzman, C. Li, J. Meyer, H. K. Pao, and T. Paxia. A different manhattan project: Automatic statistical model generation. *Journal of Software Engineering*, 2000.