Master of Science Thesis in Information Technology
Department of Electrical Engineering, Linköping University, 2019

Detailed Procedurally Generated Buildings

Sebastian Andersson

LINKÖPING
UNIVERSITY

Master of Science Thesis in Information Technology

Detailed Procedurally Generated Buildings:

Sebastian Andersson

LiTH-ISY-EX--19/5215--SE

Supervisor: **Harald Nautsch**
ISY, Linköpings universitet

Examiner: **Ingemar Ragnemalm**
ISY, Linköpings universitet

Information Coding
Department of Electrical Engineering
Linköping University
SE-581 83 Linköping, Sweden

Abstract

With the increasing size of 3D environments manual modelling becomes a more and more difficult task to perform, while retaining variety in the assets. The use of procedural generation is a well-established procedure within the field today. There have been multiple works presented within the field before, but many of them only focus on certain parts of the process.

In this thesis a system is presented for procedurally generating complete buildings, with an interior. Evaluation has shown that the developed system is comparable to existing systems, both in terms of performance and level of detail. The resulted buildings could be utilized in real time environments, such as computer games, where enterable buildings often are a requirement for making the environment feel alive.

Contents

1 Introduction

Modeling can an extensive task when working with a three dimensional environment. At the same time do the environments that computer games and movies utilizes become bigger as time goes by and are sometimes even infinitely large. This does not only create more content for the 3D artists to model, but also poses the risk of creating infinitely large sets of models. A solution to this can either be to reuse content though out the world or to utilize procedural generated content. The former will cause repetition within the environment, while the latter offers more varied content at the expense of computational complexity.

While a common use for procedural generation is within the game or film industry, it can also be utilized within the field of archaeology and city planning. By utilizing procedural generated content one can recreate plausible cities that have existed previously or create completely new ones. This enables the user of such a system to focus on the end goal rather than creating the content that populates the environment.

Previous research has, among other things, covered the generation of urban landscapes and thus including generation of both street layouts and buildings. However, the urban landscapes that are generated typically consists of large numbers of skyscrapers with a small degree of detailing, while also often being limited to the buildings' exteriors. While this is sufficient for fast paced movement in movies or for picturing how things will look at a distance, it will not be sufficient for the close ups that tend to be common in e.g. computer games. Since movies are being shot at predetermined locations 3D artists can increase the level of detail of nearby objects manually, while utilizing generated content with less detail in the distance. But since the camera often is not fixed in computer games one cannot utilize this technique for adding detail to the scene, without having noticeable transitions between detailed and less detailed areas.

The difference between a skyscraper and typical house is great, since the latter often has far more pronounced features than the former. Houses are also far more varied in their appearance, while the average skyscraper can be approximated as a cuboid.

1.1 Motivation

While previous research have investigated the ability to generate exteriors, floor plans and furniture placement, few have incorporated all three elements within the same framework. All of which plays an important part for whatever a building will be perceived as realistic for the end user.

1.2 Aim

This thesis aims at describing a framework for procedurally generate buildings, with a high level of detail.

1.3 Research questions

This thesis will be focused on the following research questions:

- How can a system for procedurally generating buildings, that includes interior design, be constructed?
- How would the performance be from such a system, both in terms of computational time and subjective thoughts on the result?
- How do the results from the such a system compare to other implementations within the field?

1.4 Delimitations

This thesis will focus on traditional houses that are utilized for accommodation purposes and thereby not include other types of buildings such as skyscrapers and public buildings. The implemented solution will also be restricted to work as a tool within an existing 3D engine, but the general steps should remain applicable to other mediums.

1.4.1 Requirements

The developed system should follow the following requirements:

- Floor plan
  - Should have a logical and plausible layout, such that room size and connections between them appear realistic.

- Exterior
  - Should provide a plausible wall layout with associated windows, entry way and roofing.
- Furniture
  - Should be placed in such a way that the object corresponds to the room and is placed in a somewhat proper manner.
- General
  - The system should require minimal modifications in order to modify the used asset set and possible room types.

2 Related Work

This chapter presents related work that have been performed within the field of procedural generation. General approaches for city generation will be presented first, followed by more in depth descriptions of methods aimed at buildings and their interiors.

2.1 Procedural city generation

Parish and Müller presented a system called CityEngine, which utilized L-systems to create credible street layouts with appropriate buildings, of urban environments [10]. L-systems consists of a set of symbols and a grammar which specifies what a set of symbols can be replaced with, hence creating a recursive rewriting system which is similar to fractals. The algorithm takes external data, such as elevation maps and population density, into account when generating the layouts. A secondary L-system was utilized for creating the buildings after the street layouts had been generated. Each building was constructed in an iterative manner which allowed for a controlled level-of-detail, where each iteration increased the amount of details.

2.2 Buildings

Müller et al. presented Computer Generated Architecture (CGA) shape in 2006, which is a shape grammar aimed at creating computer generated structures [9]. Shape grammars works in a similar fashion to L-systems, in the way that rules can be applied to shapes which results in new shapes. A simple rule could be to add a cuboid on top of another, which when applied recursively will create more complex shapes. Their proposed method works in an iterative fashion by starting with a basic building block, called an axiom, which is then being processed by applying a set of rules to it, thus adding further details. Building blocks thus have the ability to have successors which provides additional features. The grammar, thereby, provides a way of describing relations between shapes. The building's most dominant shapes can be described as mass models which often can be modeled as a combination of a set of volumetric shapes. The mass model can then be complemented with a roof, which results in a complex model of the buildings outer shell. By testing for occluded areas one can determine whenever an area on the surface will be suitable for supporting further details, such as windows. This eliminates the risk of accidentally placing details where two shapes intersect. The last improvement to result is to utilize snapping in order to align elements with each other.

Split grammar, as presented by Wonka et al., is another iterative approach for generating detailed buildings [13]. The main difference, compared to more commonly used shape grammars, is the application of splits. A split is the action of dividing a shape into smaller shapes, such as the act of dividing a cuboid into a set of three-sided prisms. This allows for additional detail to be added with each step. A shape can also be replaced by another shape in order to modify its appearance, if it still would fit within the volume of the previous shape. An example of how this method can be applied is the act of dividing a wall into a set of windows, which in turn can be divided into glass planes and decors, while the remaining space will remain as walls.

Laycock and Day developed a method for creating three dimensional models of roofs, based on building's footprints[6]. The method was based on straight skeletons, as presented by Felkel and Obdrzalek[2]. The algorithm took a building's footprint as an input and then constructed a skeleton within it. By moving vertices and intersections that were created between the skeleton and the outlined footprint one can change which style that the resulting roof will have. The algorithm also supported buildings with extensions, which would result in a merge between each part of the building's roof.

2.3 Footprint and floor plans

Rodrigues et al. presented a framework for generating realistic floor plans [11]. The system was primarily based upon utilizing an L-system in order to create a graph, which describes the relations between all rooms in a building. The links between each room are defined by a set of rules, which are classified as high- or low level links. A high-level link would be important rules that enforce a realistic plan, such as not separating two public spaces with a private one. Low level links would be common or preferred plans, such as placing the kitchen in relation to the dining room. The second step into process is to determine the size of each room, which can either be based on explicit conditions for each room or based on the total area of the building. Each room is then placed in a grid by a lazy algorithm, in order to create the final plan. The last step is to add doors, windows to the floor plan before creating the final 3D model. The resulting model is created based on the finished floor plan and utilizes the hipped roof algorithm developed by Laycock and Day [6] for the roof.

A simpler approach for generating floor plans was developed by Lopes et al. [7]. The algorithm is, just like the one by Rodrigues et al. [11], based on a grid where each room is placed as a one-by-one square. Each room is then expanded rectangularly, in an iterative fashion. When no room can expand further the algorithm tries to expand them in an L-shape. The last step is to check whenever there are any empty spaces within the floor plan and if any such spaces are found it fills them in. By dividing the initial grid into private and public sections, the algorithm is able to enforce similar requirements as high level ones as described by Rodrigues et al.. The placement of doors is influenced by the building's topology and can be constrained by defining rules, such as the kitchen should be linked to the living room. The algorithm also enforces that all rooms are reachable when the doors are being placed.

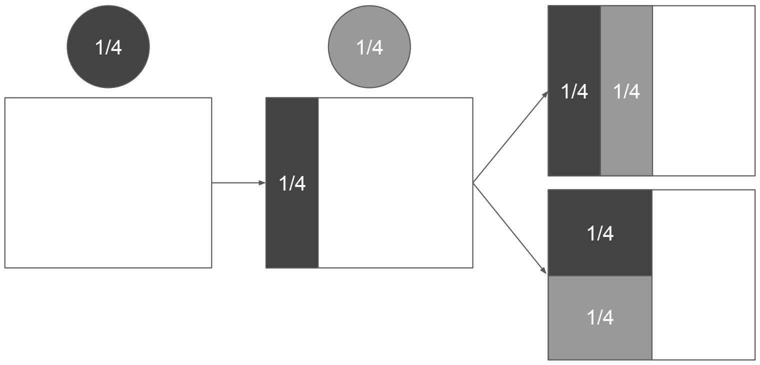
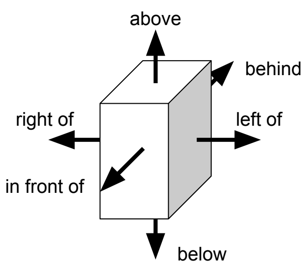


Figure 2.1: *Illustration of the concept of squarification. Based on illustration by Bruls et al.[1]*

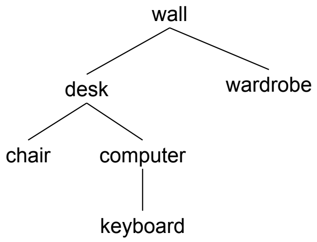
Bruls et al. developed an algorithm for generating squarified tree maps[1], which later has been utilized within the field of floor plan generation by Marson and Musse[8]. Tree maps is a way of presenting data in a visual form, by representing data as rectangles within a bounding box. Normal tree maps tend to, compared to the squarified variant, get many elongated rectangles. While the normal tree maps do accurately represent the data, it poses the risk of being less readable than the squarified version. The algorithm is based on first sorting the data in descending order, regarding to what will be represented in the finished map. This is followed by iterative creating and populating rows within the bounds of the map. Before an item is placed, it determines if the aspect ratio will improve if it becomes added to the current row or if it should create a new one. An example of such a decision is illustrated in Figure 2.1, where data that represents 0.25% of the total area is added twice to the bounding box. The process is then repeated until all data has been added. It is however worth noting that the algorithm does not guarantee an optimal result and the data will not retain its original order, since it has been sorted during the process. The the reason for why the data is sorted before being inserted is because the authors found that sorted data yielded the best results in terms of the resulting aspect ratios [1].

Marson and Musse suggested a simplified approach for creating floor plans, with the use of squarified tree maps [8]. The method is based upon generating a set of rooms where the main properly of each one is its area. The area is then used in order to populate a rectangular bounding box (the building's footprint) by utilizing the squarified tree map algorithm developed by Bruls et al. [1]. Squared tree maps are in comparison to other tree mapping methods not ordered, but provides lower aspect ratios for the elements. Since the maps are not ordered the connections between the rooms are not taken into consideration when the tree map is generated. Marson and Musse solved this issue by adding the constraint that each room should be connected to the living room and if a room was not adjacent to the living room a hallway was placed in order to enforce that constraint. A limitation with this approach is that the footprint can only be a rectangle or a square. They also presented a method for generating the interior wall, which build upon placing a small top aligned wall segment in order to create a doorway. Windows were created in a similar fashion, but with the extension of also adding a bottom aligned segment, which would then create a hole in the middle. It is worth noting that the construction of the walls was not the primary objective in their paper.

2.4 Interior



(a) An object's bounding box and spatial relations.



(b) Example of a formed hierarchy.

Figure 2.2: *Object representation and possible hierarchy formed when placed. Based on illustrations by Germer and Schwarz [3].*

By borrowing the concept of agents from artificial intelligence Germer and Schwarz developed an autonomous method for furniture placement within buildings [3]. Each object within a room, such as a wall or piece of furniture, is viewed as an agent. An object contains a bounding box, an object type, a list of the types of objects that are acceptable parents, which sides the parents that accepts the object. Bounding boxes are utilized a simplified representation of the object when performing collision handling. The actual collision check is performed using the separating axis theorem (SAT), which is described in more detail in section 4.4.3. Figure 2.2a illustrates the representation of an object. They also contain a description of how much available space that each face of the bounding box has that accepts further children.

The objects within a scene can then be arranged as trees, where each child is connected to a parent, which is illustrated in Figure 2.2b. All agents start in a search state, in which it searches for a suitable parent. If a parent is found, it aligns itself with it based on what face of the parent's bounding box accepts the object. If no parent is found, for any of the available objects, the room is considered to be fully arranged and the system is at rest. If a parent is moved after the placement has occurred, the children move with it and if a parent is deleted the children reenters a search state.

The system also allows for grid-based layouts where similar agents aligns themselves to each other and then gets placed at regular intervals. The authors give the example of aligning ceiling lights in a large room based on a grid-based layout. The authors present a few limitations to their method. One being the risk of having an uneven distribution of the furniture and another the fact that the object sometimes becomes placed in unlikely places. Examples of the second limitation are when a TV is placed in such a way that it cannot be used or when a bathtub is placed directly in front of a toilet bowl. The time it took to generate an average room with 20-30 objects took under 1 second, while more complex rooms such as one with close to 300 objects takes up to 4 seconds. They do, however, mention that their implementation has not been developed for optimal speed [3].

Yu et al. presented a framework for synthesizing indoor environments, which was based on a set of constraints and relationships between the objects, which were based on a set of examples that was deemed as good. The relations ships utilized was defined as spatial hierarchical and pairwise. The identified relationships were then utilized together with the constraints in order to formulate the placement of the objects as an optimization problem. The object representations included their bounding boxed, center, orientation, accessible space and viewing frustum. The accessible space defines a space that has to be accessible by a human in order for the object to be usable, while a potential viewing frustum defines spaces that has to be visible like in the case of a painting. They also utilized a constraint representing pathways between doors, in order to not block such paths.

The constructed problem was then solved with the utilization of simulated annealing. In terms of performance, the implementation took 22 seconds to create a living room with 20 objects, which was the fastest presented room type. A bedroom with 24 objects took 24 seconds and the room with the longest running time was a flower shop consisting of 64 objects took 376 second [14]. Their solution is more complex and takes more properties into account, compared to the once proposed by Germer and Schwarz [3], thus creating well-constructed results at the cost of significantly longer running times compared to the method used by Germer and Schwarz.

3 Methodology

This chapter will cover the methodology on which the results of this thesis build upon.

3.1 Prestudy

A prestudy was performed before the development of the implementation began. The goal of the prestudy was to investigate methods that have been utilized within the field previously and what performance that could be expected from such systems. The resulting data was then utilized for selecting suitable methods mainly for approaching the floor plan generation and furniture placement.

3.2 Implementation

An implementation of a proposed framework for procedurally generating buildings, based on the previously stated requirements has been developed. The purpose of the implementation is to provide a proof-of-concept and illustration of what kind of results that can be expected from the developed framework. By evaluating the implementation, one will then be able to compare it to previously developed frameworks that covers the same aspects. The implementation is described in further detail in Chapter 4.

3.3 Evaluation

The system has been evaluated both in terms of computational performance and subjective opinions on the results. Computational performance has been defined at the time it takes to generate either parts of a building or a complete one. By evaluating each major part in the implementation one can compare the results that have been found in previous studies which only has focused on a specific part of the generation.

The subjective opinions have been evaluated by a survey that covers the most typical results from the system. The results from the survey will not only act as a good measurement of how the results are perceived, but also provide insight into what potential limitations the system has. The evaluation methodology is described in further detail in Chapter 5.

3.4 Assets

Throughout this thesis there are illustrations with different furniture models. The used models are provided by Kenney2018 under the Creative Commons Zero (CC0) license. The texture utilized for the roof has been provided by the user Scouser at Open Game Art [12] under the CC0 license.

4 Implementation

The approach proposed in this thesis is based upon generating buildings with minimal human interference. Each building's initial data is based upon the overall rectangular footprint of the building and a set of available objects for the interior. Based on this data one will be able to generate a large variety of buildings that fits into the same footprint and utilizes the same overall style characteristics. Each part of the generation process is presented in this chapter, beginning with the floor plan generation followed by the exterior and interior.

The implementation of the proposed framework has been implemented in the Unity engine. The main reason for this is that it provides the essential rendering capabilities that are outside of the scope of this thesis, since the purpose of this thesis is to provide a framework for generation rather than rendering. Furthermore, does it provide in-depth profiling options which further decreases the amount of time spent on things that are considered to be out of scope. The framework itself is, however, not bound by the chosen engine and can thereby be replicated in other environments.

4.1 Interior Configuration

An interior configuration is a set of configurations that are vital for the generation of buildings. It contains room configurations for all available rooms, which specifies what objects that can be placed within them, how their minimum and maximum size, if they can be placed more than once and if they are vital for the buildings' functionality. An example of a vital room might be a kitchen or a bathroom, while a vital object might be a bed in a bedroom. It also contains information about what room that is the master room, which is defined as the room which all other rooms should be connected to and that should contain the door to the outdoor area. The connection can be either directly between the master room and another room or by utilizing a corridor. The master room should therefore preferably be a public room and has been chosen to be the living room during the development of this thesis, which is similar to what Marson and Musse [8] used in their work. Lastly, it contains what object that corresponds to the interior walls and a maximum scaling factor, which affects the size of the generated rooms. The variables present in an interior configuration are presented in table 4.1 and 4.2.

One can see an interior configuration as a style sheet for the building, that specifies both constrains and the style of the building. They hence provide an easy way of manipulating the results of the generation and by creating multiple instances of them with different furniture sets one can create different styles for different generations.

Table 4.1: *Description of an interior configuration.*

Variable	**Type**	**Description**
Master room	Room type	What type of room the master room is.
Interior wall	Placeable Object	Object to use as interior wall
Room configurations	List of room configurations	Available rooms and their corresponding configurations.
Max scaling factor	Float	Maximum ratio that room can be scaled by.

4.2 Floor plan generation

4.2.1 Room generation

Before generating the actual floor plan, the system generates a set of rooms that later will populate the floor plan. A room is defined by a room type and its area. The generation is based on creating rooms that fill up the specified area for the building's footprint in an iterative manner. Once one more room can be placed, it will return the resulting list of rooms and potential unused area will get distributed across the rooms based on their size.

Algorithm 1: Room size generation

```
$area \leftarrow$ Total area of building;
$config \leftarrow$ Interior configuration;
$config.maxScaling \leftarrow$ Maximum scaling factor;
$room \leftarrow$ Current room;
$factor = area/config.rooms.Sum(r => r.minSize)$;
$factor = Clamp(scalingFactor, 1, config.maxScaling)$;
$area = Random(factor * room.minSize, factor * room.maxSize)$;
```

The process begins with generating the master room, followed by the rooms

Table 4.2: Description of a room configuration utilized by interior configurations.

Variable	**Type**	**Description**
Name	string	Optional name for room.
Type	Room type	Type of room.
Singular	Boolean	Whenever multiple instances of the room are allowed.
Vital	Boolean	Whenever the room is vital to the building's functionality.
Minimum size	int	Minimum area for room.
Maximum size	int	Maximum area for room.
Objects	List of <Placeable object, Int>	Objects that are allowed to be placed within the room and the maximum amount of each one of them.
Vitals	List of furniture types	Vital types of objects that are vital for the room's functionality.

that are marked as vital. Once all vital rooms have been generated it will fill up the rest of the available space by selecting the largest possible rooms that fits the remaining area. The size of each room is determined by the minimum and maximum area that is specified for the room in the interior configuration. The minimum and maximum area is then adjusted based on the ration between the total area of the building and the sum of the minimum areas of all rooms in the interior configuration. The adjustment is made in order to allow the rooms to grow beyond the specified limits when the building is larger than the sum of the minimum areas, hence allowing large buildings to have more open space. Algorithm 1 describes the method used for calculating the area of a room.

4.2.2 Layout

The floor plan is generated with the use of squarified tree maps in a similar manner to what has been proven by Marson and Musse[8]. The process starts by creating a set of rooms that are suitable for the given bounding box's area, where the bounding box corresponds with the building's footprint. Each room is assigned a total area, which is utilized for populating the building's bounding box with the rooms. Once all rooms have been placed each room's corners are quantized in a uniform manner, in order to remove unrealistically small displacements in otherwise continuous wall segments. The used step size for the quantization has been 1 m. It is worth nothing that all bounding boxes, for both the rooms and the building's footprint, will have a rectangular or squared shape as an effect of the tree map-based method.

Once every room has been placed, the algorithm adds corridors that connect rooms that lacks possible neighbours to connect to. This eliminates the risk of having rooms that are unreachable. The process can be described as the following:

1. Find the non-connected room with the longest distance to a master room, which is a room which allows connections to all other rooms (usually the living room, but can be chosen arbitrarily during setup).

2. Create a corridor between the room and the master room or an existing hallway.

3. Repeat until all rooms have valid connections.

By prioritizing connections to other corridors over valid rooms one will minimize the number of different corridors that leads to the same area. Since the approach starts with the room that is the furthest away it becomes likely that a later room will have a corridor nearby.

A potential problem that with this approach is that the living room will have a large amount of doorways. A large amount of doorway will leave little room for furniture to be placed in a way without blocking one or more doorways. The proposed solution to this problem is to place the master room first, hence ignoring its size. This will place the master room in one of the corners which eliminates the possibility to add doorways along the outer walls, hence decreasing the risk of creating large amounts of doorways.

4.2.3 Ordered vs. Random

The squarified tree map algorithm is an approximate way of getting square like representations and the original specification is based upon sorting the data in decreasing order, according to the size of the elements. This creates an ordered behavior within the map, which means that larger rooms are followed by smaller ones. While this can be a positive behavior for visualizing file systems, since the overview will group large files together, it will not necessarily be a good behavior when creating floor plans. The reason for this is that a building is not forced to have large rooms at one end and small ones in the other, thus creating an unrealistic layout. Variation and more natural behavior can thereby be achieved by utilizing unsorted data as the input for the algorithm, at the cost of having less square like shapes. The problem with visible ordered rooms is less pronounced if the number of rooms are small or have similar sizes. The system has therefore the ability to either sort or randomize the data before creating the tree map, depending on the preferred aesthetic. Possible corridors will not be affected by this, since they are generated after the rooms has been placed.

4.2.4 Corridor generation

Unconnected rooms are found by finding the overlapping sides of the master room and the other rooms. If no overlap was found or if the overlap is big enough to fit a door the room is marked as unconnected.

All unconnected rooms are first sorted in descending order, in regard to their distance to the master room. All rooms, then performs an omnidirectional search along the inner and outer walls, with the goal of finding either the master room or an existing hallway. The open set is initially set to be all four corners of the room where the search originates from. The actual hallway is then created by offsetting the walls that are part of the found path towards the room's center, hence creating a gap where the walls once were. Each gap is then filled with a new room, which is tagged as a hallway.

If the mater room is not placed at any edges of the building a corridor will be generated to the closest edge of the building. This has been done in order to place the door that leads to the outside are in connection to the master room, by adding a door at the end of the generated corridor. However, if the master room is already located at one of the edges it will place a door on that wall instead, since no corridor has been created.

The corridor generation does however not take into account how much a room gets shrunk and thereby poses the risk of making one room very small or even remove it completely. Since the algorithm only traverses walls it does sometimes result in cases where the wrong room gets shrunk and thereby creates a connection to a neighboring room instead of the allowed goal room. This is, however, an infrequent problem and is fixed by rerunning the room generation, since the time it takes to generate a floor plan is minimal (which is been shown in more detail in chapter 6).

4.2.5 Doors

Doors are represented as a point on an edge of a room and the rotation that the final door will be facing. Doorways are added by finding the overlap between two rooms' edges and then placing the door in the middle the resulting line, which will connect the two rooms. However, the line needs to be at least as long as the doorway is wide, in order for the door to fit. If this requirement is not meet, no door will be placed at that point. The rotation for the door is then created by determining the rotation between the world oriented forward vector (the z-axis) and the vector that is perpendicular to the overlapping wall segment, where the perpendicular vector is oriented 90°clockwise from the wall segment. This will set the rotation of the door, so that it is pointing into the rooms. Doors are being placed from the master room to all neighboring rooms, while the rooms that aren't neighbours to the master room places walls to neighboring hallways. If a room can connect to multiple rooms it will only place another door if no other door is closer than a certain threshold. This eliminates the risk of having rooms that have an unnatural number of doors or doors that are very close to each other, while leading to similar paths.

4.2.6 Walls

Each wall, including interior and exterior, is constructed of multiple wall segments. The segments are one sided and supports dynamic deformation for accom-

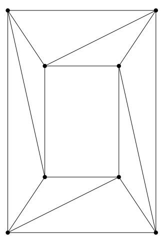


Figure 4.1: *Dynamic wall model*

modating windows and doors. This is performed by creating a rectangular hole within the wall that wraps itself around the object that is currently overlapping the it. The hole is created by placing each corner in the corresponding collision points, which are gathered by utilizing Unity's physics engine. The method does however have the drawback that it only supports one colliding object at a time and that it only supports rectangular objects. Both of these drawbacks could be addressed by implementing polygon clipping or at least a dynamic mount of vertices for the cut-out. The reason for not implementing this was that most doors and windows are rectangular, and the increased complexity would therefore only be utilized in a small subset of the total amount of cases. Figure 4.1 illustrates a dynamic wall model. Another limitation with this approach is that multiple wall segments are used, rather than a few large ones, which is not optimal. The reason for not optimizing this further has been due to the fact that the walls are not considered to be the main part of the implementation and since they were estimated have minimal effect on the overall performance of the system, further optimization would only result in minimal gains in terms of performance.

4.3 Exterior

This section covers how the exterior of the building gets generated.

4.3.1 Roof

The method utilized in the framework is based upon a generic parametric model, where adjusting the parameters will allow for different types of roofs. It is worth noting that the supported types do not include other types of roofs. The reason for this is both that the roof model is not the primary subject of this thesis and there is already a limitation in terms of the overall shapes of the buildings.

Parameterization

Figure 4.2 illustrates the five different roof types to be implemented, since they was considered to cover most common types of roofing.

The method proposed for procedurally generating buildings in this thesis supports a variety of different roof designs. The process initially started with inspect-

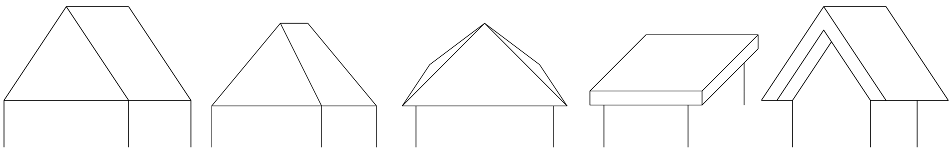

(a) *Front Gable* *(b)* *Hipped* *(c)* *Pyramid* *(d)* *Flat* *(e)* *Open Gabel*

Figure 4.2: *Roof types*

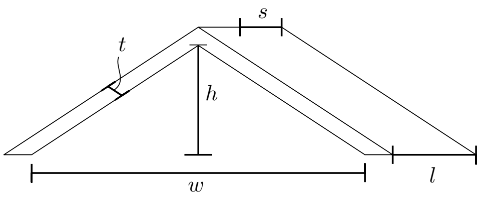


Figure 4.3: *Generic roof model with parameters.*

ing a set of common layouts in order to find common parts between them. By doing this, one can abstract the fundamentals of each design in order to create a generic model of a roof. By performing a parameterization of all fundamental features of a roof, the model can represent all of the targeted types of roofs. The types that have been targeted are the following:

- Flat Roof
- Front Gable Roof
- Open Gable Roof
- Pyramid Hip Roof
- Simple Hip Roof

The generic model is represented of main 12 vertices[1]. The model is symmetric along both the x- and y-axis. All supported types of roofs are then archived by moving one or more vertex pair in order to create the targeted characteristic. The characteristics are achieved by manipulating a set of parameters, which provides a simple way of creating different varieties. The model utilized during this thesis is based upon the following parameters, all of which are illustrated in Figure 4.3:

- Width ($w \in R_{\geq 0}$)
- Height ($h \in R_{\geq 0}$)
- Length ($l \in R_{\geq 0}$)

[1]Main vertex: A vertex is later divided into three or four identical ones, in order to properly generate vertex normals and texture mapping.

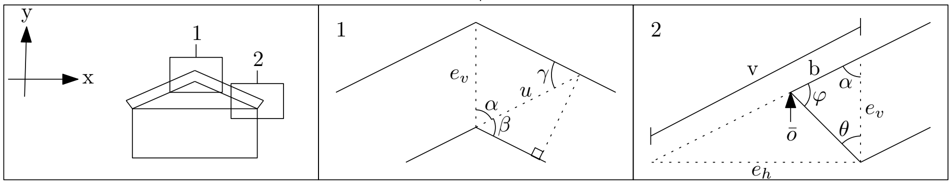


Figure 4.4: *Angles and variables for thickness and overhang calculations.*

- Thickness ($t \in R_{\geq 0}$)
- Slope Ratio ($\{s \in \mathbb{R}_{\geq 0} : 0 \leq s \leq 1\}$)
- Overhang Angle ($0° \leq \theta \leq 90°$)
- Gable ($g \in \{false, true\}$)

Roofs with a gable does have two addition triangular wall segments (consisting of three vertices each), that represents the actual gable. The gable-typed roofs do not allow for slopes along the length of the roof where the gable is placed and are also restricted to a 90°corner angle.

The vertical position of the topmost vertices is determined by the height h, width w and thickness t of the roof. The vertical extension e_v, that is the thickness measured along the y-axis, can be calculated by forming two triangles along the roof line, as illustrated in Figure 4.4. The point is calculated in 2D-space and then placed symmetrically along the length of the roof. By utilizing the law of sines, one will find that the extension can be described as in equation 4.1.

$$\begin{aligned} e_v &= u\frac{\sin(\gamma)}{\sin(\alpha)} = \frac{t}{\sin\beta} \\ \text{where } u &= \frac{t}{\sin\beta} \\ \gamma &= \beta = \pi - 2\alpha \\ \alpha &= \arctan\frac{w}{2h} \end{aligned} \tag{4.1}$$

The overhangs are controlled by the angle θ, which describes the angle between the y-axis and the overhang. $\bar{o}$ defines the other most vertex in the overhang and moves along the line that is defined by the top of the roof. The point $\bar{o}$ can be defined by equation 4.2.

$$\begin{aligned} \bar{o} &= r(e_h,\ e_v) \\ \text{where } xe_h &= e_v \tan\alpha \\ v &= \sqrt{e_h^2 + e_v^2} \\ \varphi &= \pi - \alpha - \theta \\ r &= \frac{e_v \sin\theta}{v \sin\varphi} \end{aligned} \tag{4.2}$$

Generation of Parameters

Values for the parameters that are used for the roof are either based on the building's size or randomized. The width and length of the roof are equal to the size of the building's bounding box, while the height is 75% of the height of the walls. The slope angle has a randomized value between 0 and 1, side angle has a value between 0°and 90°, while the gable is either true or false.

4.3.2 Windows

The number of windows that will accommodate each side is based on a random number between zero and the total length of the wall. The position p of a window is calculated in one dimension and is based on the relative distance to the beginning of the wall it is being placed along, in order to place them as evenly as possible. The distance is described in equation 4.3.

$$\begin{aligned} p &= \left\lfloor (i+0.5)\frac{l_{wall}}{N} \right\rfloor + 0.5 \\ \text{where } i &: \text{Current index} \\ l_{wall} &: \text{Length of wall} \\ N &: \text{Number of windows to place} \end{aligned} \tag{4.3}$$

The addition of 0.5 Moves the window into the center of the closest wall segment. This is done since the pivot point of the windows is placed in the center, while the wall segment has their pivot point at the left-hand side. If a window has been placed at the same location as the door that leads outside it gets removed.

4.4 Interior

This section covers how the interior of the building gets generated.

4.4.1 Furniture representation

Each placeable object is represented by the mesh's local bounding box, position, rotation, object type, a set of attachment points and a set of access areas. The positional properties are utilized for defining an objects central position and forward-facing direction. An attachment point is defined by a relative location, including position and rotation, to the parent object which defines a point at which another child object may be placed. The point also contains constrains regarding which object types it accepts and whenever a child objects existence is vital for the parent object's functionally (a dining table would for example not be functional without chairs). This approach of placing objects is similar to the one presented by Germer and Schwarz [3], where the children have a set of acceptable parents rather than the parents having a list of children.

The access areas define areas that has to be accessible in order for the object to be usable by a theoretical person in the room and are inspired by the use of

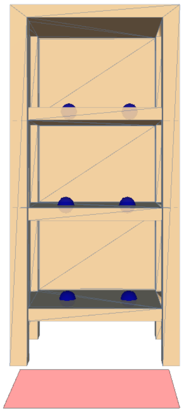


Figure 4.5: *Attachment points (blue) and a single access area (red) for a bookcase*

accessible space as described by Yu et al. [14]. This could be the space in front of a bookcase or the side of a sofa. Each access area, thereby has to be free from collisions and be accessible by a path leading from a doorway in the room to the access area. Figure 4.5 illustrates the attachment points and a single accessible area for a bookcase.

4.4.2 Furniture Placement

Furniture placement is handled by room basis. The algorithm consists of a set of furniture to place called $toPlace$, a list $open_W$ of attachment points belonging to walls, a list $open_F$ for the attachment points that belongs to place objects and a list $placed$ of the previously placed objects. The process is based on iterating through the list of furniture to place, until there either is no more furniture to place or none of them can be placed. The loop will thereby try all possible placements for all objects in $toPlace$. This means that objects that were previously unplayable will get be tested again once the loop wraps around and if no objects could be placed once it reaches the last placed object's position it will abort, since no more objects could be placed. During each iteration it will first check if there is an attachment point in $open_F$ where the current piece of furniture, if its has vital ancestors that can be placed without colliding and be accessible. It will also check if it can be a parent to any of the room's vital objects. If no such attachment point was found, it will perform the same check for $open_W$. The object and its vital ancestors will then be placed in the room if any suitable attachment point was found, during either of the previously described checks. The process is described more in depth in Algorithm 2. It is worth noting that all parts corresponding to the actual collision detection is omitted in the pseudocode and is described in-depth in section 4.4.3.

The approach is hence classified as greedy, since it is based on placing the

Algorithm 2: Furniture Placement Algorithm

```
$furniture \leftarrow$ Available furniture for room, with vital objects ordered first;
$open_W \leftarrow$ Attachment points belonging to wall segments;
$open_F \leftarrow$ Attachment points belonging to placed furniture;
$placed \leftarrow \emptyset$;
$tries \leftarrow 0$;
// Out parameter: Vital ancestors & their positions
$Collides(parent, ap,$ out $vitals<obj, ap>) \leftarrow$ Parent or vitals collides;
Function PlaceFurniture()
    while $(openW \neq \emptyset$ or $openF \neq \emptyset)$ and $tries < furniture.Count$ do
        $current \leftarrow furniture.First()$;
        $vitals \leftarrow \emptyset$;
        $fits \leftarrow Any\ ap \in open_F$ where !Collides(current, ap, out vitals) and
         ObjectsAreAccessible(vitals.accessareas);
        if !fits then
            $fits \leftarrow Any\ ap \in open_W$ where !Collides(current, ap, out vitals)
             and ObjectsAreAccessible(vitals.accessareas);
        end if
        $vitals \leftarrow vitals \cup \{(current, ap)\}$;
        if fits then
            foreach $(obj, ap) \in vitals$ do
                $placedObject \leftarrow Place(obj)$;
                $open_W.Remove(ap)$;
                $open_F.Remove(ap)$;
                foreach $a \in (obj.accessPoints \setminus vitals.values)$ do
                    $open_F.InsertAtRandomIndex(a)$;
                end foreach
                $placed.Add(obj)$;
                $furniture.Remove(obj)$;
            end foreach
            $tries \leftarrow 0$;
        else
            $tries \leftarrow tries + 1$;
        end if
    end while
end
```

objects in the first available location. This means that there is no guarantee that all, or even, any vital objects will be placed if either the room is too small or if objects are colliding as a result of one or more previously placed objects.

4.4.3 Collision handling

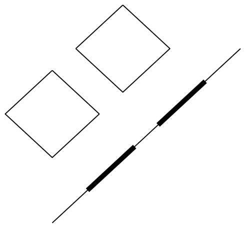


Figure 4.6: *Illustration of SAT, using two rectangles.*

Collision calculations are performed in order to ensure that no object is placed within another one. The detection does omit the parent object during the check in order to allow for less precise placement of the attachment points. An example of such a case is if a table should accept decorations to be placed on top of it, where the attachment point is placed slightly inside the table's surface. Collisions are checked by utilizing the meshes' bounding boxes, which are then represented as object oriented bounding boxes (OBB) in world space. The bounding boxes for all placed objects, excluding the parent, are then checked against the currently placed object's bounding box. It is worth noting that two objects have to intersect in order to be classed as colliding, which means that objects that just have touching sides will not count as colliding. The problem of calculating the collisions can then be solved efficiently with the use of the separating axis theorem (SAT), which is the same method which Germer and Schwarz [3] utilized for their collision handling. The reason for SAT being a suitable method is because the resulting cuboids are simple shapes with few sides. The theorem states that two convex shapes does not overlap if there exists an axis on which the shapes projection's does not no overlap [4]. Figure 4.6 illustrates how the theorem can be applied for checking if two rectangles collides. Note that the rectangles' projections do not overlap on the axis and they are thereby not intersecting.

The axes that must be checked are the three normals of each box (the other three are just flipped variants of the other ones) and the cross-product of the normals from the first box and the ones from the other box. The normals of the boxes are used to test whatever any vertex from one box intersects the second box. The results from the cross products are utilized for to checking whenever any edge intersects the other box. The result, hence is 15 axes that are checked against a total of 16 vertices for each pair of objects.

Most objects in a building will be axis-oriented, commonly because they are aligned to the walls, which means that SAT with OBB will bring an overhead compared to using axis-oriented bounding boxes (ABB) instead. However, since the attachment points allow for free rotation of the objects, ABBs would not be sufficient for objects where the difference between the OBB and ABB is big. An example of this would be a thin object that reaches diagonally across the room, which would result in an ABB that covers the entire room.

Each doorway gets an axis-oriented bounding box that is 1 m wide and expands 0.5 m into the room. The bounding box is then utilized in a stricter test where the objects ABBs are checked for collisions against the doors' ABBs. Collision checks that utilizes ABBs rather than OBBs requires less calculations to perform (check if one corner is within the other) which improves performance at the cost of being less precise. The less precise manner is something in this application can be deemed as positive, since the doors should never appear to be obstructed and hence can be less forgiving regarding collision. The windows do, however, not affect the furniture placement.

It is important to note that a pair of placed objects will only be checked against collision once, that is during the actual placement, since all placed objects remain stationary. The objects are also only checked against other objects in the same room, since all rooms are generated separately, which further decreases the number of checks performed. Each wall is constructed out of multiple wall segments, which in turn leads to many collision checks. Walls segments are therefore excluded from the collision checks and replaced by checking whether the object is contained within the room's bounding box. This check consists of checking whenever any of the objects' corners are outside of the room's bounding box.

Each access area of an object is also checked using the same SAT-based method in order to ensure that no accessible area gets covered during placement. This includes the previously placed objects' access areas and the access areas that belongs to the object that is currently being placed are free of obstructions.

The order of each collision check is the following:

1. Object is within room's bounds
2. Object ABB intersects any door ABB
3. Object OBB intersects any previously placed object's OBB
4. Object OBB intersects any previously placed access area

All collision checks regarding the furniture placement are performed on manually on the CPU and does thus not utilize any otherwise available frameworks such as the built in physics system in Unity, which in turn is based on Nvidia PhysX at the time of writing[2].

4.4.4 Accessibility

An access area is defined as an area which has to be accessible from one of the doorways, by a theoretical character that can traverse the room. The character's size is thereby relevant for determining how close objects can be placed until they are considered to block a given path. The solution proposed utilizes a traversable

[2]Unity update notes: https://unity3d.com/unity/whats-new/unity-2018.3.0

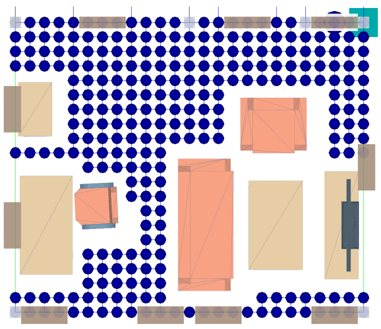


Figure 4.7: *Searchable grid without margins, used for accessibility checks, in a living room with an office area.*

grid that covers the rooms walkable space. A node is placed in all non-blocked areas within the room, while blocked nodes are omitted entirely. Each node is then linked to its neighboring node which defines the final grid. The grid is updated each time a new object has been placed in the room by removing the covered nodes and all connections that previously lead to them. Figure 4.7 illustrates an example of a grid in a living room. In order to accommodate the theoretical character's size each bounding box is inflated with the radius of the character, which is defined as a disk. The grid itself is placed in the middle of the room and has a margin its sides and the walls that corresponds to the radius of the theoretical character. The step size of the grid is adjustable for varying degrees of accuracy. The step size used for during development has been half of the character's radius where the character's radius has been 0.25m, which results an appropriately fine grid. It is worth noting that a large grid will impact the perform ace, but since the targeted buildings in this thesis are for accommodation purposes most rooms will be relatively small compared to e.g. industrial buildings.

Before a piece of furniture can be placed, the system makes sure that all previously defined and the current furniture's accessible areas are accessible after the object has been placed. Since each access area has been checked for collision in the collision stage it is ensured that the whole access area is accessible if one node within it can be reached. The object's inflated OBB poses the risk of covering the access area, which would make the area unreachable. This problem has been circumvented by marking all areas within the access area as walkable, even in places where it is placed within the OBB. Other objects' OBBs will however not create this problem since the collision check removed such cases.

The system will also perform a similar check between all the room's doorways in order to make sure that there is a path between them at all time. If no such path can be found, it the room is not considered to be functional and the proposed furniture placement is thereby invalid. The process used in this system is performed in a linear fashion by checking if one doorway has a path to all other doorways, which in turn means that all other doorways has a path to the other ones as well. The concept of enforcing paths between doorways was presented by

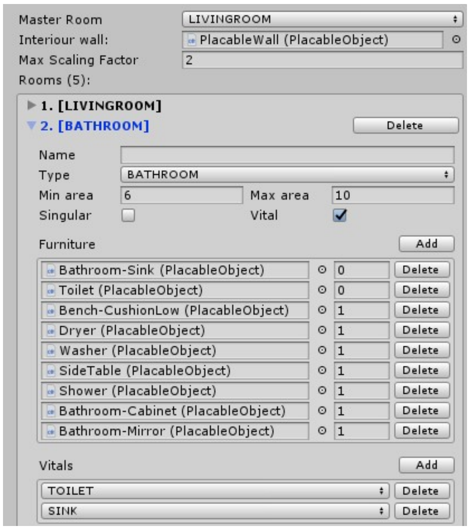
Yu et al. [14].

The actual search is performed using the A*-algorithm from one of the room's doorways to a random node within the checked access area. The used heuristic is either Euclidean and Manhattan distance between the current node and the goal node.

4.5 Floor and wall placement

The floor for each room is created by a single quad which covers the room's two-dimensional bounds. Each side of the room is then filled with the required amount of wall segments, which completes the final room.

4.6 Interface and Expandability



Master Room: LIVINGROOM
Interiour wall: PlacableWall (PlacableObject)
Max Scaling Factor: 2
Rooms (5):

1. [LIVINGROOM]
2. [BATHROOM] Delete

Name:
Type: BATHROOM
Min area: 6 Max area: 10
Singular: ☐ Vital: ✓

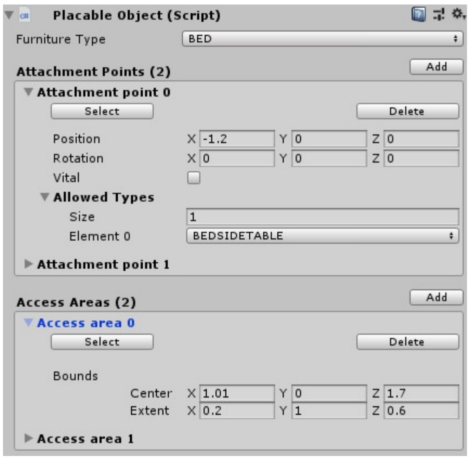
Furniture Add

Bathroom-Sink (PlacableObject)	0	Delete
Toilet (PlacableObject)	0	Delete
Bench-CushionLow (PlacableObject)	1	Delete
Dryer (PlacableObject)	1	Delete
Washer (PlacableObject)	1	Delete
SideTable (PlacableObject)	1	Delete
Shower (PlacableObject)	1	Delete
Bathroom-Cabinet (PlacableObject)	1	Delete
Bathroom-Mirror (PlacableObject)	1	Delete

Vitals Add

TOILET	Delete
SINK	Delete

(a) GUI for interior configurations



Placable Object (Script)
Furniture Type: BED

Attachment Points (2) Add
Attachment point 0
Select Delete

	X	Y	Z
Position	-1.2	0	0
Rotation	0	0	0

Vital: ☐
Allowed Types
Size: 1
Element 0: BEDSIDETABLE
Attachment point 1

Access Areas (2) Add
Access area 0
Select Delete
Bounds

	X	Y	Z
Center	1.01	0	1.7
Extent	0.2	1	0.6

Access area 1

(b) GUI for placeable objects

Figure 4.8: *Roof types*

Custom graphical user interfaces have also been developed in order to make the implementations user friendly and easily extendable in terms of configurations and objects.

Figure 4.8a illustrates the user interface used for creating and modifying interior configurations, which contains the objects and con taints used in the generation process. The top section contains the building's general settings, such as room type of the master room, interior wall object and the maximum scaling factor used for room generation. Below the general setting is a list that contains all possible rooms that can be placed in a building that utilizes the interior configuration. Rooms can be added and deleted from the list and contains all the necessary information that is required to add and furnish them. The general setting for the room is presented at the top, which includes an optional name, the room type, minimum and maximum area, followed by whether multiple instances of the room is allowed and if it is vital for the building's functionality. Each placeable object that the room can contain is dragged and dropped into the illustrated list. The number at the side of each object corresponds to the maximum number of instances of the object that the room is allowed to contain.

The interface for creating and modifying placeable objects is illustrated in figure 4.8b. The furniture type is the first setting that the user can set. Possible attachment points are presented in a list below that and contains the position, rotation, whether the object placed at the point is vital for the object's functionality and lastly a list of what items that are accepted at it. All potential access areas are displayed in a list below the attachment points and are defined by a bounding box that is placed relative to the objects pivot point. Both the attachment points and the access areas can be visualized in the 3D environment when placed by pressing the corresponding select button. Attachment points are shown as a move- and rotation tool which can be used for adjusting the point, instead of using the number inputs in the list. Access areas are visualized as a 2D plane in order to more easily see where the access area is located in the 3D space.

5 Evaluation Methodology

This section describes the methodology utilized for evaluating the resulting framework, both in terms of computational performance and subjective thoughts on the results.

5.1 Computational performance

The computational performance has been evaluated both on individual rooms and complete buildings. Each major step, that is the generation of the floor plan and furniture placement, has been measured individually. The actual evaluation is based upon the time it takes to finish the whole building and the individual steps of the process. Multiple building sizes and amount of rooms have been covered in order to investigate both real world like setups and finding edge cases were the framework either performs better or worse than expected. Each test is performed over multiple iterations in order to investigate the average times and relevant percentiles. All measurements have been done within the source code, by using .NET's diagnostic tools. Any unused allocated memory from previous measurements are cleared by utilizing the garbage collector. This has been done in order to avoid that the timed results get affected by garbage collection regarding left over memory from a previous iteration.

Since the dynamic walls are dependent on the existing physics engine each generation is required to trigger the collision simulation, in order to wrap them around the colliding object. The physics engine runs multiple times during a frame as default. This will however add unnecessary overhead, since all objects are static. The default behavior has therefore been turned off and replaced with manual calls to the physics engine after a generation has been done. The procedure can be described as the following:

1. Start high-resolution timer.

2. Generate all parts of the building.
3. Create holes in the walls with the built-in physics engine.
4. Stop timer.
5. Save data about the generation.
6. Reset timer, remove placed items and clear unused memory.

The results of the evaluations regarding the performance are presented in 6.3.

The hardware utilized for all result gathering have been a PC with an Intel® Core™ i5-6600K, NVIDIA GeForce RTX 2060, 16GB RAM, running Windows 10. The version of Unity that has been used is 2018.3.7f1.

5.2 Subjective results

The subjective thoughts on the results has been gathered by generating a set of images of the finished result, which a complete building with an interior. Each set will contain a perspective top down view in order to clearly illustrate the results from the floor plan generation, as well as an isometric picture that illustrates the furniture placement from a different angle. Each illustrated result has then been shown to a set of people that have given their subjective opinion on the result, both in terms of the floor plan generation, furniture placement and the overall results. All participants have been shown the same set of illustrations in order to provide consistent data. Each illustration has been taken based from a random uninterrupted sequence of generations in order for the generations to remain unbiased. Furthermore, does the sets contain multiple sizes of buildings in order to investigate if some sizes provide better results than others. The complete evaluation set is constructed from four sets, containing five buildings each. For each building there where two questions, one regarding the quality of the floor plan and one regarding the quality of the furniture placement. At the end of each set there where two questions regarding the perceived variety in the floor plan and furniture placement. All questions were answered by a scale with the following alternatives:

- Excellent
- Very good
- Good
- Fair
- Poor

The final survey and the collected responses are presented in Appendix A.

Table 5.1 contains the specifications for each set that was utilized in the survey. Note that the amount of the total area taken up by corridors is greater when the rooms are sorted and with the master room placed first.

The results of the survey are presented in 6.4.

Table 5.1: *Sets utilized in the survey.*

Width [m]	Height [m]	Ordered & Master first	Corridor area
16	9	Yes	48%
16	9	No	18%
16	12	Yes	51%
16	12	No	12%

6 Results

This chapter will cover the computational performance of the system and subjective thoughts on the generated results. The interior configuration that was used had specifications for a living room, kitchen, bathroom, bedroom and a master bedroom. The living room is set to be the master room in all presented cases, since it represents what was deemed to be the most realistic choice.

6.1 General results

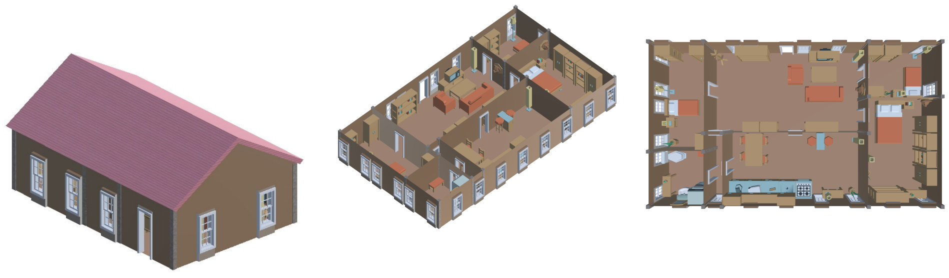

(a) *Exterior* ***(b)*** *Interior side view* ***(c)*** *Interior top down*

Figure 6.1: *Examples of a typical building*

Figure 6.1 illustrates examples of typical results from the system. Note that the exterior and interior illustrations are not from the same building. The exterior is constructed with a gabled roof and has a varying number of windows placed along each side. The interior has a living room with a TV area, storage and decorative pieces throughout the room. The kitchen has the vital objects in the form of a stove, sink and a dining area. This is accompanied by non-vital objects such as a bar table, kitchen appliances and decorations. There is one master bedroom and two smaller regular bedrooms, all of which contains a bed and storage. The bathroom has a toilet, sink, shower, washing machine and dryer.

6.2 Effect of constraints

This section will cover the results generated by the implemented system and illustrate the effect created by the enforced constraints on the generation.

6.2.1 Accessibility and collision

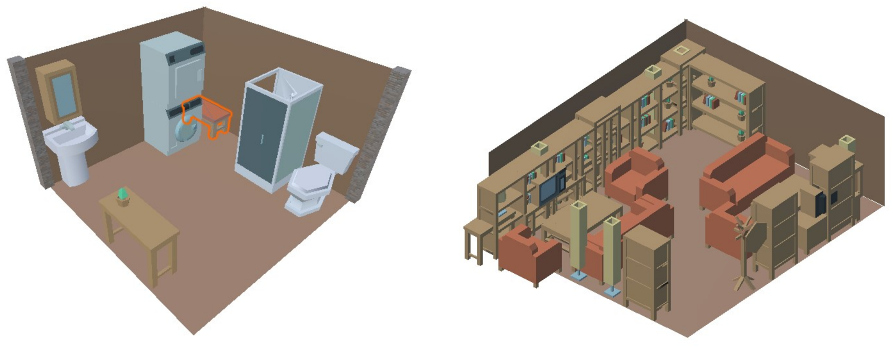
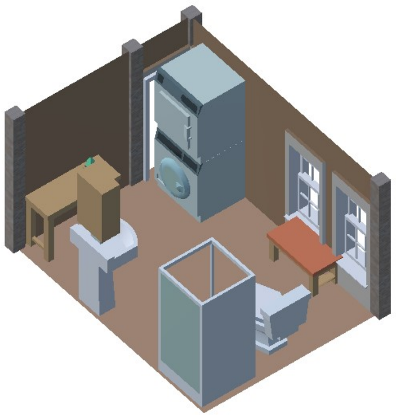

(a) No check for accessibility *(b) No collision checks*

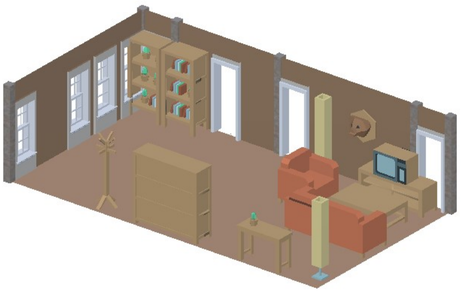
***Figure 6.2:** Rooms generated without implemented constraints.*

Figure 6.2a illustrates the effect of not utilizing a constraint that enforces the accessibility of objects. One can see that there is a bench (highlighted in orange), placed in the corner of the bathroom, which is not accessible due to the washing machine and dryer that stands in front of it. The implemented accessibility constraint would eliminate this type of situation since the access area would be colliding with the dryer. It would also eliminate situations where the object's access areas are inaccessible from one of the doors, which would occur if one or more objects blocked the path to it.

Figure 6.2b illustrated a room that was generated without any collision handling, which also includes removing the accessibility checks since they would perform similar collision tests. The result is a room that is filled with overlapping objects, since there are no constraints regarding this. The lack of collision handling is more noticeable compared to the lack of an accessibility constraint.



(a) A case when a washing machine blocks a doorway.



(b) A case when a TV bench blocks a doorway.

Figure 6.3: *Rooms generated without the constraint that enforces the doorways to be unobstructed.*

6.2.2 Door obstruction

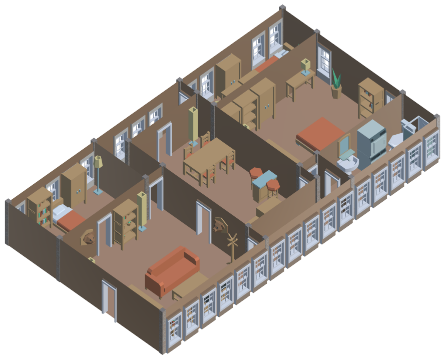
Both Figure 6.3a and 6.3b illustrated the effect of removing the constraint that enforces furniture to not obstruct the door. The washing machines blocks the door in the first figure, while the TV bench obstructs it in the second figure. The collision handling for these cases are handled separately from the object to object collisions, which is the reason for why the objects do not collide in any of the figures. Furthermore, has the accessibility check been turned off, since it would enforce a similar behavior to the door collision algorithm.

6.2.3 Room order

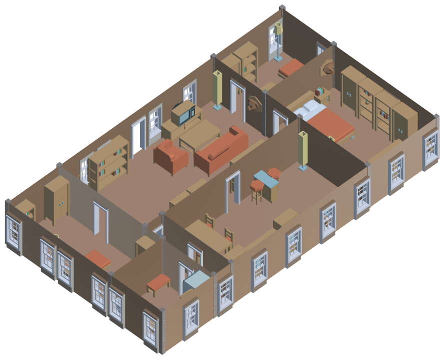
The effect of utilizing sorted or randomized order when generating the floor plan is illustrated in Figure 6.4. In Figure 6.4a the master room is placed first, which will result in it being placed in the first corner in all generations. Figure 6.4b illustrates a building where the order of the rooms has been randomized, which makes the master room appear in the center of the building. It is worth noting that the number of doors to the master room is larger in the unordered case. Placing the master room first do, however, increase the chances of getting more corridors, since less walls are shared with other rooms.

6.3 Computational performance

This section will cover the computational performance of the implemented system, in terms of computational time for one or multiple steps of the process. All



(a) Ordered rooms, with the master room placed first



(b) Randomized room order

Figure 6.4: *Furnished buildings generated with sorted and randomized room order.*

Table 6.1: *Computational performance results in terms of rooms, objects and running time, given a fixed footprint size.*

Width [m]	Height [m]	Rooms	Objects	Time [ms]	p95 [ms]
16	16	10.4	145.1	804.2	1127.9
16	9	5.5	83.7	502.9	701.6
12	12	5.5	86.8	500.5	731.0
9	9	4.5	57.2	158.5	228.2

presented results are based on sets of 1000 generations that has created in an uninterrupted season.

6.3.1 Performance of full building generation

Table 6.1 presents the results regarding average rooms, objects and running time in regard to fixed footprint sizes when utilizing sorted data and where the master room is placed first. Large buildings will have more rooms and objects compared to the smaller ones and does thereby take longer time to generate, which is expected. The 95th percentile is also included, since it provides a better insight in how the system would perform in general. The average computational time is below 1 second in all cases, while the larger 16m by 16m building has a 95th percentile that exceeds 1 second.

The squared building with dimensions 12mX12m has the same area as the building that is 16mX9m. Both gets the same amount rooms on average, which

Table 6.2: *Computational performance for floor plan generations.*

Width [m]	Height [m]	Ordered (Y/N)	Rooms	Time [ms]
16	16	Yes	10.4	11.8
16	16	No	10.3	7.5
16	9	Yes	5.5	1
16	9	No	5.5	0.6
12	12	Yes	5.5	0.7
12	12	No	5.6	0.4
9	9	Ye	4.5	0.5
9	9	No	4.5	0.4

Table 6.3: *Computational performance for furniture placement.*

Width [m]	Height [m]	Rooms	Objects	Time [ms]
16	16	10.4	144.4	806.5
16	9	5.5	83.8	481.2
12	12	5.5	87.3	468.0
9	9	4.5	57.2	148.9

is to be expected since the room generation only works based on the footprints area. The number of objects and the running time is also highly similar between the two.

6.3.2 Performance of floor plan generation

The average time for generating a floor plan is presented in Table 6.2. The results that are marked as ordered also placed the master room first. Half of the presented tested floor plans took less than one millisecond to generate. The largest floor plan was the one that took the longest to generate. The reason for this is because it has more rooms and has to generate multiple corridors in order to connect all of them. Floor plans that utilized a sorted room order took longer to generate compared to the ones with a randomized order. All tested floor plans do, however, have low average computational time, when compared to how long a complete building takes to generate.

6.3.3 Performance of furniture placement

Table 6.3 presents the average computational time for placing the furniture. Most of the time spent during the generation is spent on furniture placement. Larger buildings will not only have more rooms to fill, they will also be larger and thereby take longer time to populate.

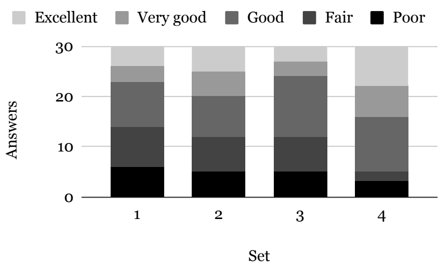
6.3.4 Performance of dynamic walls and roofs

The measurement of the average time for performing the collision detection and reshaping of a wall segment was 0.04ms. The amount of time for creating the walls are thereby very small and will have a little effect on the overall computation time. When investigating the running time for generating a roof, including the randomization and deforming process, the average running time was 0.07ms. Which will provide an even smaller impact compared to the walls, since a roof is only created once per building.

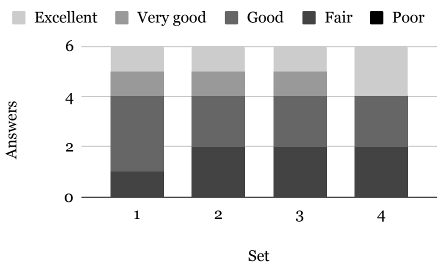
6.4 Subjective evaluation of results

The results from the survey regarding subjective thoughts on the result are presented in this section. The survey was answered by six different volunteers. None of the volunteers had any information regarding the sets' specifications. The complete results from the survey are presented in Appendix A.2.

6.4.1 Floor plans



***(a)** Perceived quality.*



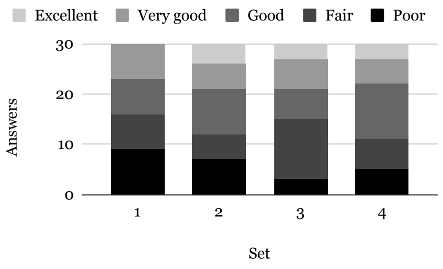
***(b)** Perceived variety.*

***Figure 6.5:** Survey results regarding floor plans.*

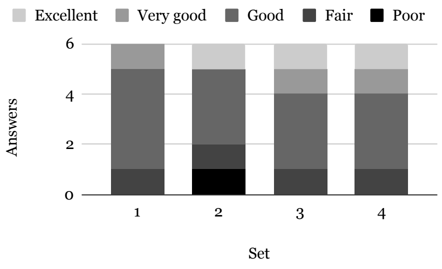
Figure 6.5a illustrates the summarized of the answers for each set. The participants did prefer the unordered version of the sets over the corresponding ordered one. The largest building was considered to be the best one in regard to the floor plan. Multiple participants did, however, remark that the number of corridors was too large in some cases for their liking. The sometimes uneven distribution of space (e.g. a large bedroom in a house with a small kitchen) was also mentioned to be a deciding factor for lower scores. It is thereby important to note that the sets that placed the master room first and used a sorted room order did have larger amounts of corridors than the unordered ones.

The variety of the floor plans was considered to be relatively good by the majority of the participants. A third of the volunteers did, however, perceive the variety as fair.

6.4.2 Furniture Placement



(a) Perceived quality.



(b) Perceived variety.

Figure 6.6: *Survey results regarding furniture placement.*

The results from the survey regarding the furniture placement are presented in Figure 6.6a. The furniture placement were considered to be the best in the last set and worst in the first one. The distribution was however more even in this case compared to the floor plans. A common remark was the fact that furniture was placed too close to doors, covered windows or was placed in narrow spaces.

The variety regarding the furniture placement is significantly better than the one for the floor plans. This indicates that similar setups are either not present or have not been pronounced enough for the participants to notice them as repeating.

7 Discussion

7.1 Results

The computation time for the system has been shown to be good, since the average time for all tested buildings has been less than one second. This would make it possible to utilize the system in real time applications, where the buildings could be generated when needed. The computational times presented by Germer and Schwarz [3] are matched by the implemented system. However, since they did not provide exact numbers or hardware it is hard to compare the two on a deeper level. The solution presented in this thesis does have the extended constraints of accessibility and vitality, while lacking the ability to place objects in a symmetrical manner. The system does, however, have a less computational time than the solution presented by Yu et al. [14], while also being less precise in its results.

Germer and Schwarz [3] noted that their implementation sometimes placed objects in unlikely spots or had an uneven distribution of the objects. Both of these limitations have been identified in the implemented system that is described in this thesis and is mostly due to the fact that the objects have no notion of the distribution of objects within a room. The addition of an accessibility constrains does eliminate the risk of having objects that are unusable because the path to them is blocked.

The implemented system does, compared to the one developed by Marson and Musse [8], not risk carving into any of the rooms' corners when generating the corridors. This means that all rooms will keep a rectangular shape and the corridors will thereby be relatively straight, since they only are allowed to traverse along whole walls. The limitation with this approach is the fact that corridors might become long and thereby taking up more space than necessary.

The position of the master room was thereby deemed as an important factor regarding the quality of the resulting floor plan. While placing the master room first will provide more room for furniture to be placed, since more walls are free of doorways, it also often leads to more corridors. The main limitation of the system was concluded to be the large amount of corridors, since they often lead to unnatural behavior or wasted space. This is mainly due to the fact that all rooms have to be connected to the master room and the number of corridors would thereby decrease if other connections were allowed.

The utilized method for generating walls is very simple compared to the ones developed by Wonka et al. [13], which was able to gain details in an iterative fashion. The walls do, however, have a dynamic nature which enables them to work with multiple types of window- and door models, without requiring any changes to the behavior of the wall. This does thereby provide a more dynamic solution compared to the once utilized by Marson and Musse [8], which was based on placing a wall segment above the door and one segment above and one below a window.

Multiple volunteers remarked on the furniture being too close to the doorways. The reason for this has been that the minimum distance between the front of a doorway and an object has been set to the 0.5 meters, which results in narrow gaps. The solution would therefore be to increase the margin between doorways and objects to the desired minimum distance. Another remark was the fact that some paths were narrow, because of some objects' placements. This would be resolved by increasing the theoretical characters size, that is utilized for accessibility evaluations.

The results are heavily dependent on how the placeable objects have been set up, both in terms of attachment points and access areas. Creating such points and the areas does require manual interaction. A small amount of available attachment points that are available during object placement will lead to less varied results, which is dependent on the object setup process. Omitted or poorly placed access areas can also affect the results, since it can potentially result inaccessible objects.

The feedback regarding the variety for both the floor plan generation and the furniture placement was overall positive in the survey. Furniture can be placed in a lot more places compared to the rooms, which also benefits the variety of the placements. The floor plan does seem to be the largest limitation in the system, both in terms of realistic structure and variety.

7.2 Implementation

The implementation has fulfilled its purpose of being a proof of concept of a system for generating buildings. The system has, as discussed in the previous section, multiple limitations that affect the outcome. Furthermore, was it not created with optimizing its performance in mind and could thereby be developed further for better performance. It did, however, show promising results in that regard.

The limiting factor within the implementation was the choice of method for generating the floor plan. Since squared tree maps relies on rectangles and squares, it will only be able to work with rectangular or squared footprints. It will also only generate rectangular rooms, which is not desirable in all situations. This thereby constrains the buildings overall shape and the rooms within it.

Different parameters such as the maximum room scale and theoretical character's size, have not been tested due to time constraints. It is possible that different values of some parameters would yield better results, since some of the given feedback regarded such cases where the current values caused abnormal layouts in some situations.

7.3 Evaluation methodology

The timing solution for evaluating the computational performance of the system has provided results with a good enough accuracy to ensure the system's general performance. The goal of the solution has been to provide data that can be compared with previously implemented systems, which has fulfilled. Each data collection was performed on 1000 generations. The amount of iterations was not increased, since more accurate data was not needed for evaluation purposes and increasing it would hence only increase the run time of the tests.

The survey helped to identify the most vital limitations in the system and was able to give an unbiased view of how the system performed. However, increasing the number of participants and the number of illustrated sets could have given clearer data in the cases where the answers were scattered.

7.4 The work in a wider context

The developed implementation provides a solution for generating 3D environments in the form of buildings with interiors. Such environments could then be utilized in real-world applications such as computer games, in order to provide a large amount of content for the user to explore. The endless amount of buildings such a system can generate will be more than what is possible to do with manual modeling.

The developed system is, however, not free from limitations and while the result overall is decent and might work in some scenarios does not mean that it is applicable everywhere. The system still requires manual input in the form of the models that needs to be prepared before they can be utilized. Furthermore, the system does not provide the level of detail that e.g. a level designer would be able to accomplish. While the level of variety was shown to be good in the brief testing that was made in the survey, this does not mean that the variety will be good enough for a computer game, since the user can explore hundreds or even thousands of buildings during gameplay.

8 Conclusion

The developed system did provide a good solution for procedurally generating buildings, which created comparable results to previously implemented systems within the field. The appearance of hallways was concluded to be an important part of how the quality of the building was perceived. Cases when the master room was placed in the center greatly reduced the number of hallways and the results were considered to be better than those with multiple or long hallways. For this reason, future work would benefit from utilizing a different method for generating floor plans, since the constraints will most likely affect the quality of the results in a negative way.

The system showed good performance in terms of running time and was able to match previously made implementation, while retaining an acceptable degree of quality in its results. The results of the furniture placement were overall positive and were concluded to have a good degree of variation.

Appendix

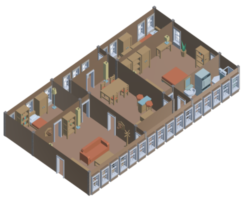
A

Servey

A.1 Survey: Illustrations

The following sections contains the illustrations that created the four sets that were utilized in the survey.

A.1.1 Set 1


(a) *Side view*

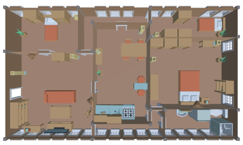
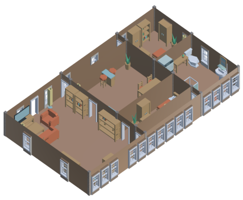

(b) *Top view*

Figure A.1: *Set 1.1*


(a) *Side view*

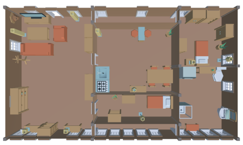
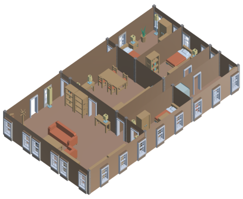

(b) *Top view*

Figure A.2: *Set 1.2*


(a) *Side view*

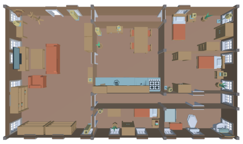
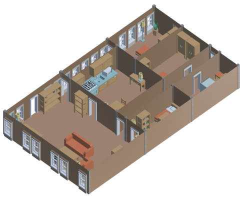

(b) *Top view*

Figure A.3: *Set 1.3*


(a) *Side view*

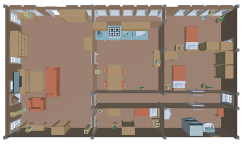
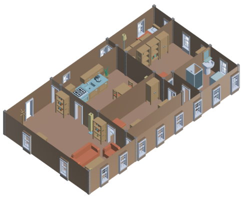

(b) *Top view*

Figure A.4: *Set 1.4*


(a) *Side view*

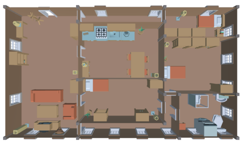
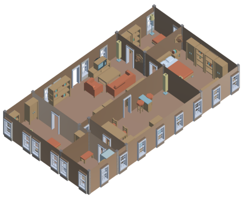

(b) *Top view*

Figure A.5: *Set 1.5*

A.1.2 Set 2


(a) *Side view*

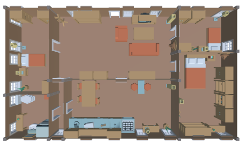
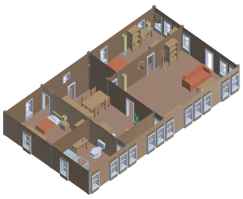

(b) *Top view*

Figure A.6: *Set 2.1*


(a) Side view

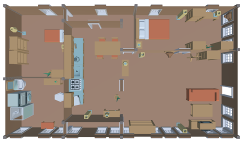
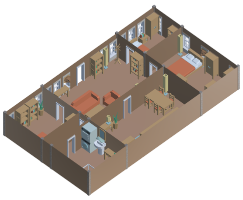

(b) Top view

Figure A.7: *Set 2.2*


(a) Side view

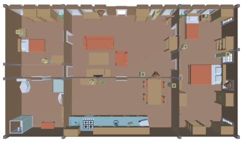
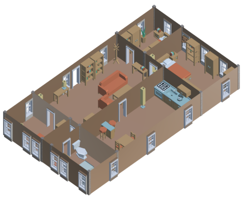

(b) Top view

Figure A.8: *Set 2.3*


(a) Side view

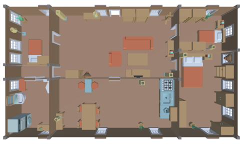
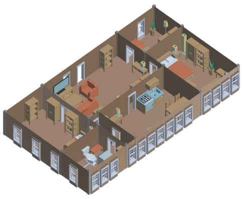

(b) Top view

Figure A.9: *Set 2.4*


(a) Side view

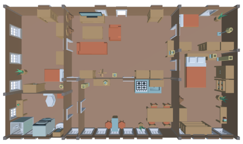
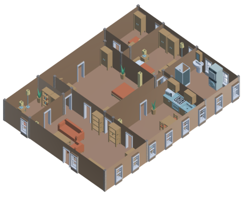

(b) Top view

Figure A.10: *Set 2.5*

A.1.3 Set 3


(a) Side view

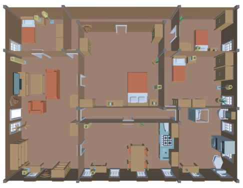
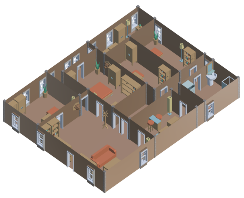

(b) Top view

Figure A.11: *Set 3.1*


(a) Side view

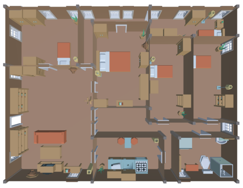
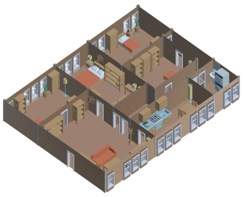

(b) Top view

Figure A.12: *Set 3.2*


(a) Side view

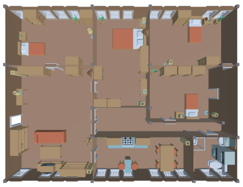
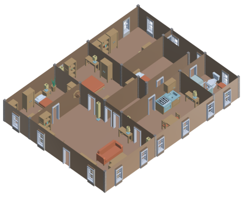

(b) Top view

Figure A.13: *Set 3.3*


(a) Side view

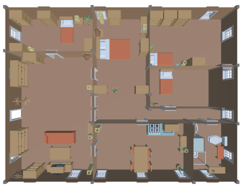
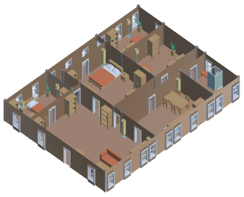

(b) Top view

Figure A.14: *Set 3.4*


(a) Side view

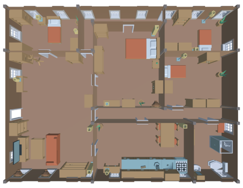
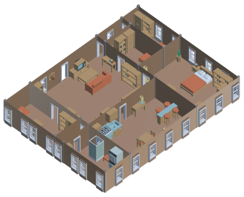

(b) Top view

Figure A.15: *Set 3.5*

A.1.4 Set 4


(a) Side view

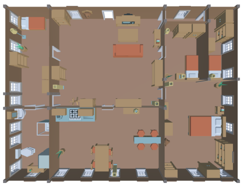
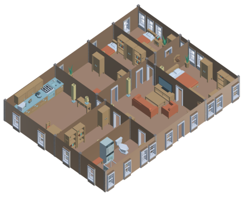

(b) Top view

Figure A.16: *Set 4.1*


(a) Side view

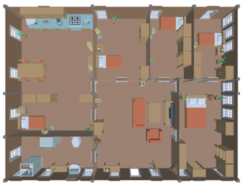
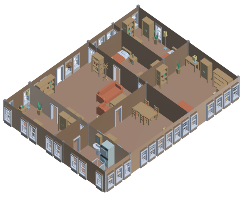

(b) Top view

Figure A.17: *Set 4.2*


(a) Side view

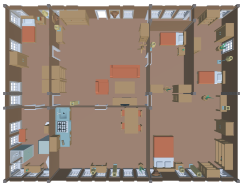
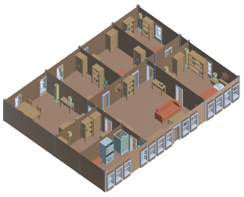

(b) Top view

Figure A.18: *Set 4.3*


(a) Side view

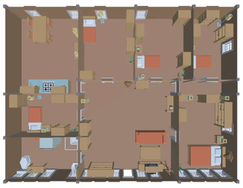
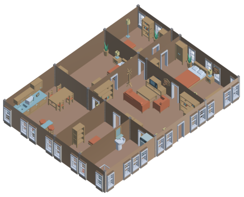

(b) Top view

Figure A.19: *Set 4.4*


(a) Side view

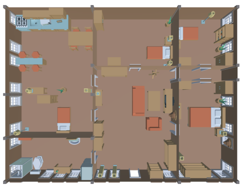

(b) Top view

Figure A.20: *Set 4.5*

A.2 Survey: Answers

Table A.1 contains all the answers, except from the free text one, from the survey. The questions used are described in the list below and the number within the parentheses corresponds to the set that the question was used in.

- Q1 = How do you perceive the floor planning?
- Q2 = How do you perceive the furniture placement?
- Q3 = The variety in the floor plans was:
- Q4 = The variety of the furniture placement was:

A.2.1 Answers for multiple choice questions

Table A.1: *Answers from the survey. P = Participant*

Question	P1	P2	P3	P4	P5	P6
Q1 (1.1)	Excellent	Very good	Poor	Fair	Fair	Poor
Q2 (1.1)	Very good	Very good	Poor	Good	Poor	Fair
Q1 (1.2)	Excellent	Good	Fair	Good	Good	Fair
Q2 (1.2)	Very good	Good	Poor	Fair	Poor	Poor
Q1 (1.3)	Excellent	Very good	Poor	Good	Poor	Good

Q2 (1.3)	Very good	Good	Poor	Very good	Poor	Good
Q1 (1.4)	Excellent	Good	Fair	Good	Poor	Good
Q2 (1.4)	Very good	Good	Fair	Fair	Very good	Fair
Q1 (1.5)	Very good	Good	Fair	Fair	Fair	Poor
Q2 (1.5)	Good	Fair	Fair	Poor	Good	Poor
Q3 (1)	Excellent	Very good	Fair	Good	Good	Good
Q4 (1)	Very good	Good	Fair	Good	Good	Good
Q1 (2.1)	Excellent	Very good	Fair	Very good	Good	Good
Q2 (2.1)	Excellent	Good	Good	Good	Poor	Fair
Q1 (2.2)	Excellent	Fair	Fair	Fair	Fair	Poor
Q2 (2.2)	Excellent	Good	Poor	Poor	Poor	Poor
Q1 (2.3)	Excellent	Good	Fair	Good	Good	Very good
Q2 (2.3)	Excellent	Very good	Good	Good	Fair	Poor
Q1 (2.4)	Excellent	Good	Fair	Good	Very good	Very good
Q2 (2.4)	Excellent	Very good	Fair	Fair	Very good	Good
Q1 (1.5)	Excellent	Very good	Fair	Good	Good	Good
Q2 (1.5)	Very good	Very good	Good	Fair	Poor	Good
Q3 (2)	Excellent	Very good	Fair	Good	Fair	Good
Q4 (2)	Excellent	Good	Good	Good	Poor	Fair
Q1 (3.1)	Excellent	Good	Fair	Poor	Good	Poor
Q2 (3.1)	Excellent	Very good	Fair	Fair	Good	Poor
Q1 (3.2)	Very good	Fair	Poor	Poor	Fair	Fair
Q2 (3.2)	Very good	Fair	Poor	Fair	Good	Fair
Q1 (3.3)	Excellent	Good	Good	Poor	Good	Good
Q2 (3.3)	Excellent	Fair	Good	Fair	Very good	Good
Q1 (3.4)	Excellent	Good	Good	Fair	Very good	Good
Q2 (3.4)	Very good	Fair	Fair	Fair	Very good	Fair
Q1 (3.5)	Excellent	Fair	Fair	Good	Good	Good
Q2 (3.5)	Excellent	Good	Fair	Good	Very good	Poor
Q3 (3)	Excellent	Good	Good	Fair	Very good	Fair
Q4 (3)	Excellent	Good	Good	Good	Very good	Fair
Q1 (4.1)	Very good	Good	Fair	Poor	Good	Excellent
Q2 (4.1)	Very good	Good	Good	Good	Good	Good
Q1 (4.2)	Excellent	Good	Good	Very good	Good	Excellent
Q2 (4.2)	Excellent	Very good	Fair	Very good	Good	Excellent
Q1 (4.3)	Excellent	Very good	Good	Poor	Good	Excellent
Q2 (4.3)	Excellent	Good	Good	Poor	Fair	Good
Q1 (4.4)	Excellent	Very good	Good	Good	Poor	Excellent
Q2 (4.4)	Very good	Good	Fair	Poor	Poor	Fair
Q1 (4.5)	Very good	Good	Good	Very good	Fair	Excellent
Q2 (4.5)	Very good	Good	Fair	Poor	Poor	Fair
Q3 (4)	Excellent	Good	Good	Fair	Fair	Excellent
Q4 (4)	Excellent	Very good	Good	Good	Fair	Good

A.2.2 Remarks

The following list contains the answers from the last question, regarding any remarks.

- P1
    - The overall thought around these layouts are that they are great and sometimes even wonderful. However, I remarked that the generator has a tendency to build small rooms and then place objects in tight places, in a realistic world this would not be possible and thus one could have problems reaching for example the bed or wardrobes. However, regards to plants and lamps put into these tight situations I believe that this is more realistic as they are smaller objects. I also reacted on the bear-heads which had a tendency to be placed directly on windows. However, I do not find it an issue for furniture to do this.
- P2
    - No comment
- P3
    - The biggest issue for me was the many narrow corridors, which can be a waste of space. Without them I would have rated the floor planning higher
- P4
    - "Some general feedback on the floor plans: Most of the rooms had somewhat awkwardly placed corridors. Where simply a bigger room might have been better there was a corridor that was taking up that space. In some cases the bathroom was only accessible through a bedroom which would in general be considered poor room planning. The sizes of the rooms could sometimes be unfair to their use (big single bedroom, tiny kitchen). But in general the floor plans where good, access to all rooms was ensured and there was nothing too crazy going on. The later examples without corridors greatly improved the overall floor design.

      Some general feedback on furniture placement: In many cases furniture blocked windows. A lamp in front of window would be acceptable but a hanging wall decoration would not. A bed might be in front of window but a bookcase might not be well placed there. The other this is that sometimes doors where partially blocked by furniture. I don't think the placements were considering the walking flow of the house, which might explain why some of the placements where obstructing the paths. Some concrete examples was when a sofa group was blocking the walking path from one room to the other or when a bed was blocking the door when it could have been moved more center to the room.

In general, if considering that these floor plans and furniture placements where computer generated, I would call them acceptable or great in some cases. Keep in mind that it is very easy to turn an excellent furniture placement to a poor furniture placement by simply introducing an obstructive furniture in one of the rooms. Because of this, I think that this problem is in general very hard solve and thus I want to give some credit to this work. From what I have observed in these examples I think that with some minor improvements this could become very good. "

- P5
  - No comment.
- P6 (Translated from Swedish)
  - Furniture is often i the way of door ways. Book cases and hunting trophies are some times in front of windows. Coat hangers are some times in the wrong part of the room. Unnecessary or weird corridors could be removed.

Bibliography

[1] Mark Bruls, Kees Huizing, and Jarke J. van Wijk. Squarified Treemaps. pages 33–42. Springer, Vienna, 2000. doi: 10.1007/978-3-7091-6783-0{_}4. URL `http://link.springer.com/10.1007/978-3-7091-6783-0_4`.

[2] Petr Felkel and Stepan Obdrzalek. Straight Skeleton Implementation. *Proceedings of Spring Conference on Computer Graphics*, pages 210–218, 1998. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.131.7175`.

[3] Tobias Germer and Martin Schwarz. Procedural Arrangement of Furniture for Real-Time Walkthroughs. *Computer Graphics Forum*, 28(8):2068–2078, 12 2009. ISSN 01677055. doi: 10.1111/j.1467-8659.2009.01351.x. URL `http://doi.wiley.com/10.1111/j.1467-8659.2009.01351.x`.

[4] Stefan Gottschalk. Separating axis theorem. Technical report, Technical Report TR96-024, Department of Computer Science, UNC Chapel Hill, 1996.

[5] Kenney. Furniture Kit, 2018. URL `https://www.kenney.nl/assets/furniture-kit`.

[6] Robert G. Laycock and Andrew M. Day. Automatically generating roof models from building footprints, 2003. URL `https://dspace5.zcu.cz/handle/11025/991`.

[7] Ricardo Lopes, Tim Tutenel, Ruben M. Smelik, Klaas Jan de Kraker, and Rafael Rafael Bidarra. A Constrained Growth Method for Procedural Floor Plan Generation. *Proceedings of GAME-ON 2010, 17-19 November, Leicester, United Kingdom, 1-8*, 2010. URL `https://repository.tudelft.nl/view/tno/uuid:7828c749-ddcd-4aa0-b14f-cdce3a164cf5`.

[8] Fernando Marson and Soraia Raupp Musse. Automatic Real-Time Generation of Floor Plans Based on Squarified Treemaps Algorithm. *International Journal of Computer Games Technology*, 2010:1–10, 10 2010. ISSN 1687-7047. doi: 10.1155/2010/624817. URL `http://www.hindawi.com/journals/ijcgt/2010/624817/`.

[9] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. In *ACM SIGGRAPH 2006 Papers on - SIGGRAPH '06*, volume 25, page 614, New York, New York, USA, 2006. ACM Press. ISBN 1595933646. doi: 10.1145/1179352.1141931. URL `http://portal.acm.org/citation.cfm?doid=1179352.1141931`.

[10] Yoav I. H. Parish and Pascal Müller. Procedural modeling of cities. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques - SIGGRAPH '01*, pages 301–308, New York, New York, USA, 2001. ACM Press. ISBN 158113374X. doi: 10.1145/383259.383292. URL `http://portal.acm.org/citation.cfm?doid=383259.383292`.

[11] Nuno Rodrigues, Márcio Dionísio, Alexandrino Gonçalves, Luís G. Magalhães, João P. Moura, and Alan Chalmers. Incorporating legal rules on procedural house generation. In *Proceedings of the 24th Spring Conference on Computer Graphics - SCCG '08*, page 59, New York, New York, USA, 2010. ACM Press. ISBN 9781605589572. doi: 10.1145/1921264.1921279. URL `http://portal.acm.org/citation.cfm?doid=1921264.1921279`.

[12] Scouser. Free Urban Textures: Roof, Tiles, 2015. URL `https://opengameart.org/content/free-urban-textures-roof-tiles`.

[13] Peter Wonka, Michael Wimmer, François Sillion, and William Ribarsky. Instant architecture. In *ACM SIGGRAPH 2003 Papers on - SIGGRAPH '03*, volume 22, page 669, New York, New York, USA, 2003. ACM Press. ISBN 1581137095. doi: 10.1145/1201775.882324. URL `http://portal.acm.org/citation.cfm?doid=1201775.882324`.

[14] Lap-Fai Yu, Sai-Kit Yeung, Chi-Keung Tang, Demetri Terzopoulos, Tony F. Chan, and Stanley J. Osher. Make it home. In *ACM SIGGRAPH 2011 papers on - SIGGRAPH '11*, volume 30, page 1, New York, New York, USA, 2011. ACM Press. ISBN 9781450309431. doi: 10.1145/1964921.1964981. URL `http://portal.acm.org/citation.cfm?doid=1964921.1964981`.