Procedurally Generating Complex Structures

Oliver Biggar	Jiayi Shen	Yixiang Yin
u7381193	u7174055	u7112271
u7381193@anu.edu.au	*u7174055@anu.edu.au*	*u7112271@anu.edu.au*

1 Describe the research question (3 marks)

Procedural generation is the algorithmic construction of digital content, and it is an increasingly-important part of building novel and scalable video games and virtual worlds [5, 15]. Stated generally, our research question is the following: can we develop an algorithm to efficiently construct novel and complex architectural structures in games?

By 'architectural structure', we mean large buildings through which players can move and interact. By complex, we mean that these structures have more topological structure than say, a house or office building. Some examples include castles, libraries, government buildings, museums, airports, et cetera.

To be more specific, we investigate a combination of two methods: (1) a novel grammar-based approach for constructing the building's footprint[1] and (2) a cellular-automata-based method (adapted from [3]) for filling such a footprint with separate rooms. Our goal is to determine if these simple and efficient methods are sufficient to produce a diverse range of interesting structures, and to implement a proof-of-concept in the popular open-world game Minecraft.

2 Motivate the research question (3 marks)

Video game development is a huge and growing industry. The costs and revenues of major games continue to increase, and the user base expands year on year [5]. Much of the cost and development time of new games comes from creating the game's *content* [15], the requirements of which—in terms of detail, scale and immersion—are becoming prohibitive. More recently, virtual reality for entertainment and training has exacerbated the demand for complex game content.

One particularly important and challenging example of game content are *structures*, particularly complex structures. As the spaces within which players move, and focal points of player and non-player activity, they are of great importance. These structures consequently define key aspects of a game's storyline and gameplay. Conversely, they require detail and design proportional to their importance and size, making complex structures particularly slow and costly to construct.

Procedural content generation (PCG) presents a possible solution to the cost of creating game content, and consequently has been an increasing topic of research interest in recent years [5]. PCG has successfully constructed examples of small structures and large-scale world features [15, 2, 16]. PCG offers two benefits: (1) new content can be generated at near-zero marginal *cost*, and (2) they offer the possibility of players continually encountering *novel content*, extending the playable time of a game.

The *cost* appeal of procedurally generating structures is clear. Just as real-world complex structures require the time and effort of architects and engineers, so too do virtual complex structures require significant effort. Consequently, games are limited in how many such structures they can feasibly contain. This, in turn, affects *novelty*; generating complex structures would give a much greater breadth of man-made structures for players to explore. In combination, procedural content generation of complex structures has significant potential to change the video game industry.

The importance of novelty in games is reflected in our choice of test-bed: the procedurally-generated sandbox game Minecraft [11]. The terrain of each world in Minecraft is generated via a seed, producing a

[1] That is, the shape of its outer structure.

unique environment that is essentially infinite [11]. Thus, each play of the game is distinct. The power of this novelty is such that Minecraft remains the best-selling video game of all time [11].

Minecraft contains examples of man-made structures, including villages and fortresses. However, these are designed in advance and merely appear at random in the world, and so experienced players quickly become familiar with all possibilities. Hundreds of user-made "mods" exist which add further structures to Minecraft, demonstrating the depth of player demand for new content to explore. Again, however, the structures are finite in number and are human-designed in advance.

Minecraft's procedurally-generated nature, extensibility, user base and simple block-based design make it an appealing test-bed for our purposes. It is quick to prototype, and a solution capable of producing diverse and interesting content could potentially be released as a playable "mod".

3 What is the current state-of-the-art? (6 marks)

Building procedural modelling and structure generators are one of the most well-explored PCG areas [15], owing in part to the recent rise in virtual worlds and settings that need vast amounts of material. It remains highly active in research and includes everything from shape grammars to data-driven, self-learning techniques, as well as the reconstruction of models from images and videos of real architecture. Despite much study, there are just a few tangible implementations.

3.1 Building Generation

Most of the procedural methods mentioned in the research papers can be divided into building generation with and without interiors.

Methods without internal structure focus on ways of generating collections of structures or even entire cities. Kelly and McCabe introduce an automatic city generation system, Citygen, which generates the urban geometry of a modern city [6]. Muller et al. [12] created the shape grammar computer generated architecture (CGA) shape to generate and produce arbitrary-level-of-detail building hulls. Schwarz and Muller create CGA++, the successor of CGA [14], which provides additional context information about other shapes throughout the generation, as well as constructs whole shape trees or new ones.

The methods above describe generating the outer appearance of the buildings, such as their shape and building façade. In order to construct a full 3D building, both the exterior façade and interior of a structure must be developed. Procedural techniques for producing building interiors differ significantly from grammar-based approaches for generating façades. Floor plan generation and furniture layout solving are two aspects of this topic. Many researchers have focused on the procedural production of building floor plans, resulting in grammar, subdivision, graph layout, constraint-solving, and even machine learning techniques. Data-driven or constraint-based methodologies were used to create furniture arrangements.

Rau-Chaplin et al. [13] generates floor plans in the LaHave House Project using shape grammars. Hahn et al. [4] developed a method for generating office buildings in real time, where areas are only generated if necessary and all changes are persistent. Marson and Musse have used a 2D building outline and a set of rooms with the desired area and functionality to work out a different type of treemap, based on squarified treemaps [8]. Martin [9] a graph-based method, with a graph representing the connections of different rooms of a building as edges and rooms as nodes. Merrel et al. [10] used a Bayesian network to generate residential building layouts, from complete architectural programs to 2D floor plans. Tutenel et al. [16] used a generic semantic layout-solving technique to define room types and relationships between adjacent neighbors. Lopes et al. [7] evaluated a limited rectangular L-growth method of generating fully-connected rooms. Merrell et al. [10] developed an interactive furniture layout system, providing users with recommended arrangements based on interior design rules.

3.2 Minecraft

Minecraft [11] is a globally popular 3D sandbox video game. Players are free to choose how to play the game without any required goals to accomplish. They explore an open and blocky 3D world with virtually infinite terrain, and can build structures, earthworks and tools. Plenty of players are keen to build houses or settlements according to their own imaginations or famous buildings. A $1 \times 1 \times 1$ block is used to represent each

coordinate position in Minecraft. This model provides the basis for all the blocks, locations, and dimensions stated in this project.

The Generative Design in Minecraft Competition (GDMC) is a framework to test AI in Minecraft, which is designed to facilitate the exploration of new possibilities in the realm of 3-dimensional world generation and manipulation. The goal is to develop algorithms that can develop adaptive and "interesting" cities and towns in Minecraft.

3.3 Art Papers

3.3.1 Organic Building Generation in Minecraft[3]

Michael et al. introduces a simple method for floor-plan generation for Minecraft structures within the Generative Design in Minecraft competition (GDMC) framework.

Procedural Content Generation (PCG) has been applied in games to create levels, puzzles and other contents, while the similarity, functionality and time consumption are a bottleneck of certain PCG algorithms. To achieve a good balance of these factors, this paper combines constrained growth and cellular automata to generate a building with similar functions but diverse floor plans automatically in a given 3D space.

Methods in this paper are divided into two parts: a simple constrained growth algorithm for floor plan generation and a well-known strategy Cellular Automata (CA) for external wall generation.

A floor plan generator uses a single block, which is the natural Minecraft granular unit. The room placements are determined by the rounded cubed root of the total rectangle building area calculated as the number of rooms. Rooms are assigned initial starting locations randomly within the border of the buildings, then grow one by one block each round until none of them can develop any further. Next the generator goes on to the door placement phase of the procedure when the rooms have finished growth. Doors are installed to connect rooms, and a single door is installed in the exterior wall to allow access to the building.

After generating floor plan, external wall generation is accomplished by a procedure derived from the CA family of algorithms, which is used to self-organise the placement of solid blocks and glass windows.

Different from rectangular floor-plan generation, no rectangle growth is required, and the resulting buildings have an organic look full of rooms, doors and windows. Rooms are treated as separate entities and developed with single Minecraft blocks. The method is capable of constructing thousands of buildings in seconds.

This generator also has limitations. Larger buildings become confusing without internal room landmarks. "Small rooms" are created as a side effect of door placement which do not exist in the first produced floor plan. The future work is using a more procedurally generated framework to add furniture in a room as the interior landmarks for players in large buildings.

Summarized by Jiayi Shen

3.3.2 A Constrained Growth Method for Procedural Floor Plan Generation [7]

Lopes et al. explore the problem of generating floor plans for buildings. In particular, they note the specific challenges associated with constructing building *interiors*, the unique challenges of which they contend have hampered adoption in industry. This is in contrast with comparatively well-developed methods for building façades. The focus of the paper's contribution is on developing an efficient algorithm for constructing floor plans capable of meeting the strict constraints of a believable interior, specifically *reachability* and *connectivity*.

The authors contrast their work with the somewhat limited literature on floor plan generation. The identify two general flaws of existing methods, which are predominantly based on *constraint-solving*: (1) they are slow, because the criteria are complex, and (2) they tend to be inflexible, unable to handle irregular shapes. The method they propose is 'growth-based' and so is purportedly efficient and more varied. Their method proceeds as follows.

They begin by hierachically subdividing the given space into different conceptual 'zones'; the examples they provide are the 'public' and 'private' zones of a house. This breaks the constraint requirements into a simpler form. A grid-based algorithm is then used for room placement and room expansion within the building layout. It starts with a polygon as the blueprint and determines initial locations for rooms, which are required to meet the given adjacency constraints. Rooms then 'expand' from the initial room positions by appending adjacent grid cells and terminating when all the interior has been assigned. Predefined weights

govern the relative sizes of the rooms. In its first phase, the algorithm expands rooms as rectangles, which are then converted to not-necessarily-rectangular shapes in the second phase, to ensures all space is assigned to a room. Inner doors are placed procedurally on shared walls to connect the rooms as required by the connectivity constraints.

The approach has been validated in two ways. First, the floor plans generated by the method were observed to closely match real-world floor plans of North American houses. Second, architects and experts gave feedback on the output.

The algorithm produces fairly complex floor plans from the perspective of wall geometry, which seems to be a significant advance. However, the method is still somewhat lacking in other areas. The authors point out that some aspects are somewhat simplistic; an example is their door-placement method, which randomly chooses a location in the specified wall. Further, conceptual roles of rooms are not addressed explicitly in the algorithm, and so it plausibly creates houses which while connected, require, for instance, passing through a bedroom to reach another room. Such an example appears in Figure 5 in their paper.

Summarised by Oliver Biggar.

3.3.3 Procedural floor plan generation from building sketches [1]

Daniel et al. present an automated approach for reconstructing building interiors using hand-drawn architectural designs. Some of the ideas in this paper gave us insights on floor planning generation.

Models for computer graphic applications usually are crafted by designers, which is high-demand and time-consuming. It is common to use procedural algorithms in video games to reduce the cost by automatically generating cities and buildings. To compensate for the issue that the interior is usually lacking in the building generation, they opted to create a floor plans generator from an existing building sketch. They also tend to optimise technologies at the same time.

The suggested technique uses a segmentation algorithm and morphological operations to extract external walls and openings from a building's drawing, followed by a procedural generation phase to iteratively position and extend rooms to fill the available space. This method may be used to produce multiple interior models for a predefined building façades, or to assist in the early phases of architectural design work, such as providing room layout recommendations or determining what can be done with renovations after walls and openings are in place. First, the building's shape and openings are extracted via image processing. They preprocess the input image with the Bilateral Filter created by Tomasi and Manduchi, and obtain the exterior walls and the position of the doors and windows. They use a nonlinear combination of neighbouring image values, to smooth noise while keeping edges. Then, a growth-based method is used in the procedural floor plan creation stage. A grid is formed in the available floor plan area. Each room is placed in a single eligible cell in the grid and then enlarged to fill neighbouring cells until it reaches its ultimate size. When adjacent cells are occupied by separate rooms, the grid's edges become walls.

As a result, a procedural generating algorithm is proposed, which can generate a floor layout based on the user's requirements. It is also capable of dealing with a broad range of picture styles and building designs, including non-convex polygons. They made two contributions in this paper. They proposed an automated method to recover exterior walls and openings, and a procedural model for floor plan generation considering the exterior features of a building as well as user-supplied requirements.

Expansion to multiple-floor dwellings is treated as the future work, as well as incorporating diagonal walls in the grid generation to allow for diagonal interior walls. Another enhancement would be a wall position adjustment step, which would allow for more precise control than our current grid provides. In addition, for bigger projects, a dedicated corridor placement phase might be employed to ensure complete connection.

Summarized by Yixiang Yin

4 Report in detail what you have done to solve your research question (6 marks)

Our project was split into three parts, collectively proceeding from a random seed to a generated structure in Minecraft. The steps were the following:

1. Constructed a complex footprint from a random seed and adjustable parameters, represented as an array.

2. Fill the footprint with distinct rooms, joined by doors, using the array representation.

3. Translate the array representation into a structure in Minecraft.

4.1 Constructing a complex footprint

To meet our desired requirements, the produced footprints required both significant diversity and novelty, to make them interesting centres for gameplay, along with believable architectural structures. This method also had to be efficient, and so we needed techniques capable of producing output capable of mimicking the complex and highly-structured architectural layout of a building.

We made use of two ideas. The first one was the concept of a *grammar*. Grammars describe abstract relationships, and have been historically very well-suited to the modelling of plants and natural structures, due to their inherent *fractal structure*. This is appropriate because buildings too have fractal structure, being formed from basic shapes adjoined to smaller copies of basic shapes. The second idea is *symmetry*. Buildings, particularly important buildings, tend to have symmetry or near-symmetry. Some prominent Canberran examples are both the old and new Parliament Houses.

To incorporate both ideas at once, we constructed a building grammar which defined a 'Layout' as one of several shapes, with other Layouts adjoined to its edges, known as *child Layouts*. This recursive definition is what makes this a grammar. More concretely, we used two shapes, Rectangles and Courtyards, where Courtyards are Rectangles with an open rectangular space in their centre, see Figure 1. These are both instances of the Layout class.

Figure 1: The Rectangle and Courtyard Layouts

The full system works as follows. The user defines an initial bounding box of desired size. A Layout is then placed randomly within the box, and dimensions are randomly selected for the Layout. Each layout has a set of locations where a child Layout can be placed (on Rectangles and Courtyards, these are the sides). For each such place, bounding boxes are constructed to determine the maximum allowable size of the Layout's child Layouts. Then the shape for each child Layout is randomly selected, and each child is recursively constructed in turn. Layouts which are placed on the side of another Layout extend SideLayout, which ensures that they and their children are placed only on the 'exposed' sides. See Figure 2. For each possible child location there is a possibility no child is placed. Also, each Layout class has a lower bound on the size of a bounding box in which it can be placed, and so the recursion stops once the available space is too small.

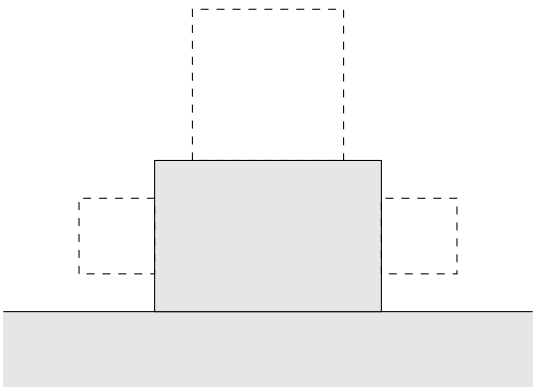


Figure 2: A Rectangle layout on the side of another layout, with the dashed rectangles showing the bounding boxes for the possible child layouts.

Thus far, we have not mentioned symmetry. This is handled in an extra step between placing a Layout and constructing its children. Each Layout class partitions the possible locations of child Layouts into *symmetry sets*. A single child Layout is constructed independently for each set in this partition, and a copy of the child Layout is attached to each element of that set. As an example, the Rectangle class partitions its sides into $\{\{UP\}, \{DOWN\}, \{LEFT\}, \{RIGHT\}\}$, $\{\{UP, DOWN\}, \{LEFT\}, \{RIGHT\}\}$, $\{\{UP\}, \{DOWN\}, \{LEFT, RIGHT\}\}$ and $\{\{UP, DOWN\}, \{LEFT, RIGHT\}\}$. These correspond to no symmetry (each side independently receives a child Layout), up-down symmetry, left-right symmetry, or both left-right and up-down symmetry. Some examples are shown in Figure 4. To make the symmetry better-matched to that of buildings, we must *reflect* copies of child Layouts which are opposite one another on the parent Layout. We explain this in Figure 3.

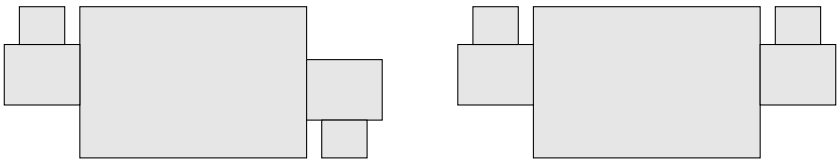


Figure 3: An example showing how copies must be reflected to produce the correct symmetry. The left is without reflection, the right is with reflection.

A valuable feature of this design is extensibility and ease of modification. First, all 'random selection' steps, including the dimensions, child shapes, and symmetry, are defined by beta distributions, which are controlled by two adjustable parameters for each choice. Each Layout class defines its own parameters, and they can also be modified on the basis of each instance. This allows us to easily construct very complex design rules, such as "Rectangles adjacent to Courtyards should have child Layouts that are very likely symmetric". In the language of grammars, this example is *context-sensitive*, which allows for considerable power in the rules implementable in our system. This allows us to tune our generator to better match desired architectural layouts.

Also, adding new Layout shapes is very straightforward, as they can simply extend the Layout class and define their own parameters. If the shape is significantly different from the defaults, one can override the required methods in the Layout class, without breaking any existing functionality.

A variety of examples of produced output are shown in Figure 4.

4.2 Room planning

Given a footprint of a complex structure, the algorithm must generate a floor plan with both diversity and novelty for the player to play in it. To make it diverse, the algorithm should generate a plan differently from each run even if the input is the same. To make it novel, the generated plan must include sufficient details in the plan, including at least rooms with a variety of sizes and shapes, the internal walls separating them, doors connecting between them.

We used a L-shaped constrained growth method researched by [7] except that it grows by one (Minecraft) block at a time. However, the original algorithm only supports rectangle footprint to start with, so we adapted the algorithm to accept almost any footprint. The adapted algorithm did the planning in a 2D matrix and works as follows. The user can define a room number or let the algorithm determine it by taking the rounded cubed root of the available area in the footprint, and a external door number(i.e. the number of doors to be placed along the external wall). The workflow of the algorithm can be separated into three phases:

- Initialize room position
- Grow rooms
- Place doors

Within the available area (i.e. the area covered by empty space in Figure 5a), each room is initialized with a random starting area which is a 2*2 squares (shown in Figure 5b). The algorithm ensures that they are not placed too close or overlapped to each other. For the room growth, one room is chosen randomly to grow by one block at each iteration until none of them can grow any more. When one room is chosen, one option will

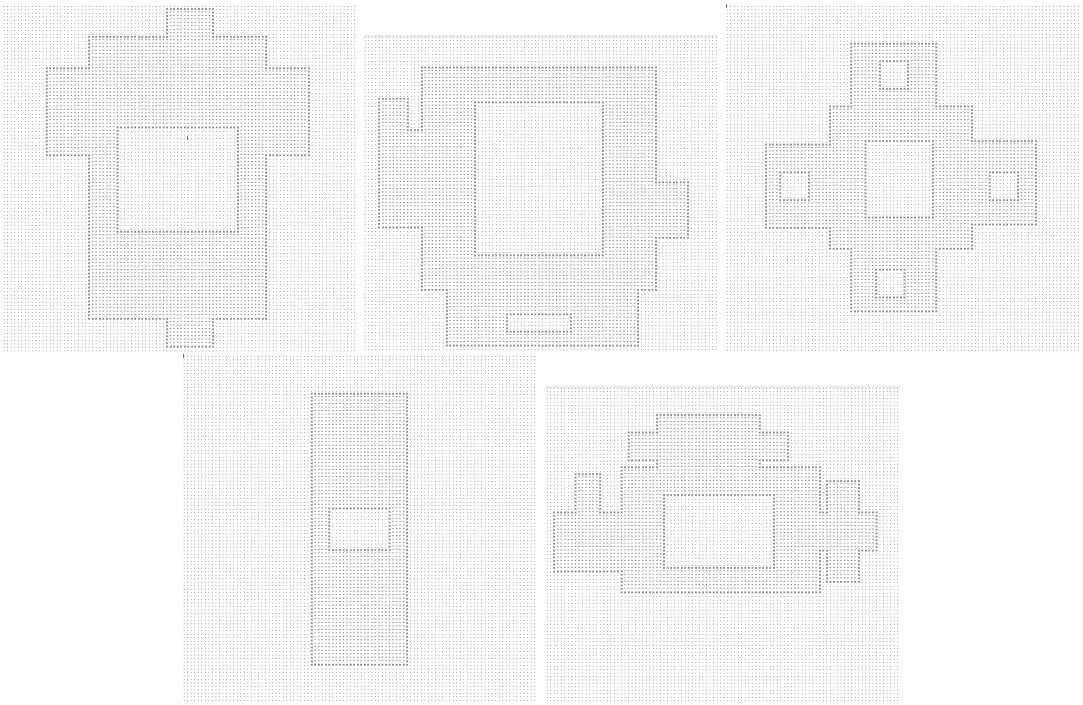


Figure 4: A collection of examples produced by the algorithm.

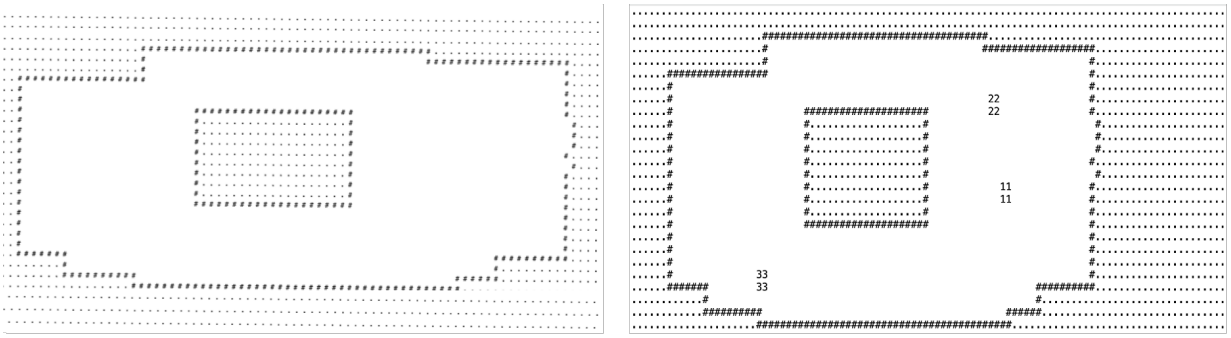
be picked from all possible growth locations around the current area of the room, but the diagonal adjacency is not considered. Rooms are constrained to only grow into empty blocks. Figure 5c illustrates what the floor plan shown by Figure 5b looks like after 10 iterations of growth. Figure 5d shows what the floor plan looks like after all growth. Then, the remaining empty blocks shown in figure 4 will be considered as internal wall after the growth. The plan after cleaning up is shown in Figure 5e. Note that we just changed the empty space in Figure 5c to # to represent internal wall but the outer lap of # still represents external walls. To ensure the transversality of the structure, the algorithm will overwrite an internal wall as a door if and only if there is at least one internal wall adjacent to it (diagonal adjacency is not considered). When a door is placed, it will end up overwriting either side of the door to the wall. More specifically, if it detects at least one internal wall is vertically adjacent, it writes one block up and down to the internal wall. Otherwise, if it detects at least one internal wall is horizontally adjacent, it writes one block left and right to the internal wall. After the internal door placement, the algorithm will place several doors(user-defined) randomly along the external wall shown as "#" in Figure 5a. A complete floor plan is shown in Figure 5f.

4.3 Conversion to Minecraft 3D Models

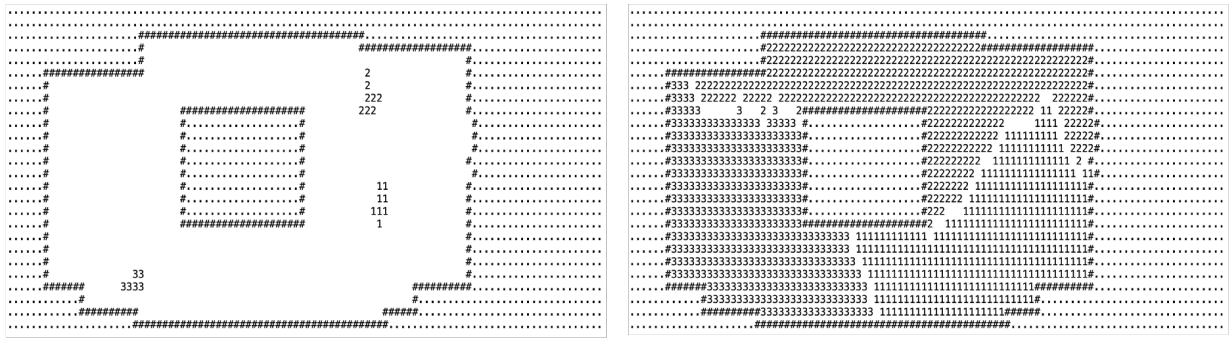
In this section, we discuss the implementation process of converting a floor plan to a Minecraft 3D model.

Buildings in Minecraft are constructed by block and block. We use the natural granular block units and develop the 3D models by placing appropriate blocks in corresponding coordinate positions. Each field is constructed from a block, and the meaning represented by each field in the numpy array is converted one by one in Minecraft to build the model we want. The correspondence between array fields and Minecraft building contents see Table 1.

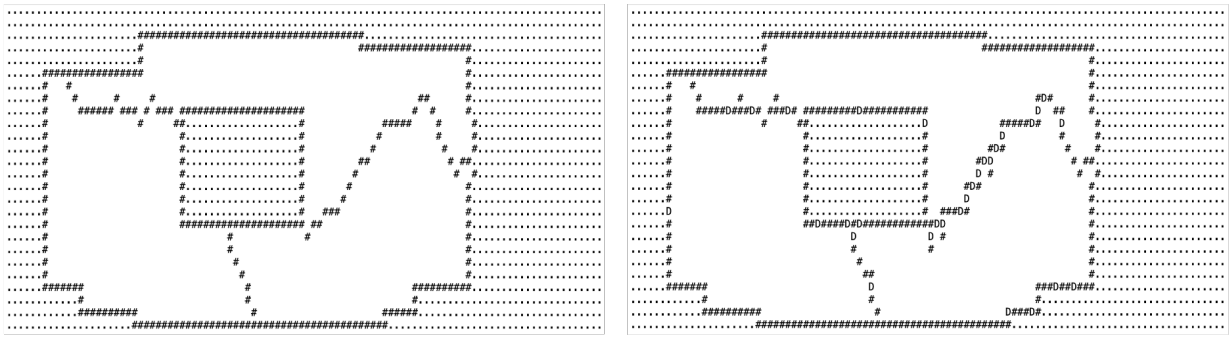
Assume we use x, y, and z to represent the length, width, and height of the 3D model, and we create a 3D space with a point in one of the plan's corners to represent the starting point (0,0,0). The distance from the beginning point on the x-axis positive direction is 1, the distance on the y-axis positive direction is 2, and the distance on the z-axis square direction is 3. We traverse each block of the Minecraft 3D model and find each point on its cross-section to correspond to the values in the array.



(a) (b)



(c) (d)

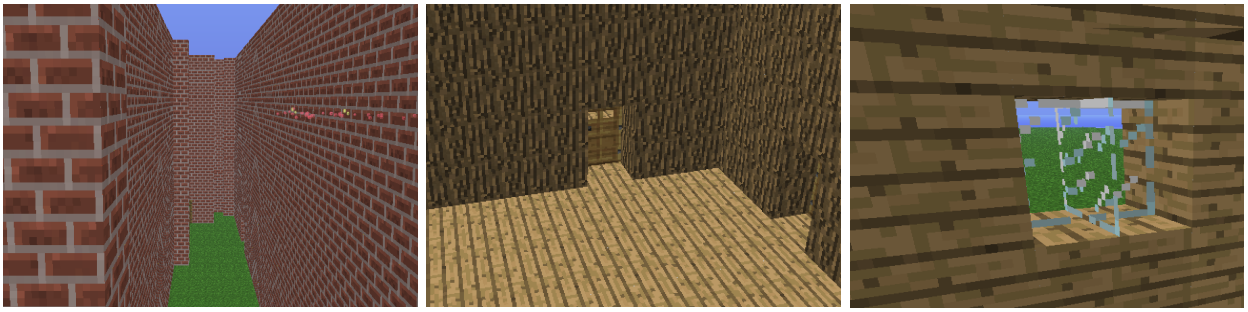


(e) (f)

Figure 5: A collection of examples produced by the algorithm.

Array fields	Meaning	Minecraft
0	Internal Wall	Wall
-1	External Wall	Wall
-2	Door	Wall
-3	Air	Nothing
-4	Corner of the external wall	Wall
-5	Window	Window

Table 1: Correspondence between Array fields and Minecraft building contents



(a) Walls (b) Doors (c) Window

Figure 6: Walls, Doors and Windows Generated in Minecraft

4.3.1 Wall, Door, Window

A block state is characterized as a -2 if it is a door, and -5 if it is a door, and -3 if it is air, and others are solid blocks. All the fields, except doors, windows and air, will be translated to the walls. The walls are initially generated with the same height of the generated building.

For the door generation, it has two conditions, one for the door itself, and one for the wall above it. The height of the door is initialized to a fixed number, hence during generating a door, it will check whether any walls are above or below. If not, the door will go up and down. Otherwise, the door will go left and right instead.

Windows are now generated randomly in the middle of the external walls at a suitable height from the ground. At window initialization, a matrix of the height at width/depth of the building is randomly generated with 75% of the wall being solid blocks and 25% being glass blocks.(See Figure 6)

4.3.2 Floor, Ceiling

Compared to contents above, generating floor and ceiling have easier rules. The blocks will cover the whole floor plan, the difference is only in the value of the y-axis. For the maximum height of the building, all blocks are constructed as the ceiling. Conversely, the minimum height, i.e., when it is 1, all generated blocks will be constructed as the floor.

4.3.3 Texture

Different parameters will be used to generate walls with different textures. Internally, Minecraft provides a diverse terrain for crafting construction materials like wood, glass, brick and other elements. Now we just select the texture of the decoration randomly according to the size of the building. In the future work, we hope to match the corresponding texture through the function of the room, so that the whole 3D model is more realistic and more immersive to the players.

All conversion codes are implemented in MCEdit through filters. Our filter can be applied to any part of the map, both in a brand-new map and an already saved map, to render our 3D model.

MCEdit is a free and open-source map editor in GDMC that allows users to create and edit map files. A filter is a command that has an effect, such as filling the whole space with air, or constructing buildings such as walls, staircases, or entire villages. We generate them by writing Python programmes that specify the filter's inputs or parameters (such as the materials, the orientation, the creator, and so on) as well as the code to be run, which is encapsulated in the execution function.

5 How did you distribute the work among your team? (3 marks)

The workflow of research is broken down into three main steps. The first phase involves constructing a complex footprint as a building area for the room generation using a random seed and adjustable parameters. The second phase is placing different rooms, doors, and walls in a building floor plan using the array representation. The next step is to use the GDMC framework to convert the floor plan into a 3D model in Minecraft.

Meanwhile, we have split the work for the research paper as well. Each of us was mainly in charge of certain questions, and there were some questions, such as Question 3, 4, 5 and 8, that we all worked on jointly. We are in charge of proofreading and double-checking any elements that were not authored by ourselves.

We allocate work in this way because it helps each person to concentrate more on their role and think more clearly and comprehensively. It also guarantees that everyone contributes properly to the project research, rather than having the opportunity to delegate their task to others.

Every member in this group contributes equally. The details of work distribution is shown in Table 2.

Group Work Distribution		
Group Member	Research Project	Paper Questions
Oliver Biggar	Oliver was mainly responsible for designing and implementing the footprint construction algorithm.	1, 2, 3, 4, 5, 8.
Jiayi Shen	Jiayi was mainly responsible for researching the GDMC framework and converting the generated floor plans into 3D models in Minecraft.	2, 3, 4, 5, 6, 8.
Yixiang Yin	Yixiang was mainly responsible for researching the existing floor plan generation algorithms, adapting one of them to fit in more complex starting footprints.	3, 4, 5, 6, 7, 8.

Table 2: The work distribution of our group for research project and paper

6 Describe the outcome of your research (3 marks)

We implemented an algorithm that is capable of constructing a huge variety of novel and complex structures with sufficient details in it. Using the fractal method to generate complex footprints, the adapted L-shaped constrained growth algorithm to plan in it and finally the translation class to map it to 3D game content, we have proved that with a combination of these relatively simple and light-weighted algorithms, we were able to generate various, complex, and interesting structure in an open-world game like Minecraft. Even if in our research, we deployed it to Minecraft as a test-bed, it could be easily extended to all other different 3D games maps and structures. Since the whole algorithm is highly modularized, one can easily interchange the translator class to deploy it in other games without modifying anything else.

There are some negative outcomes related to the adaption of floor planning algorithm. One of them was that in certain circumstances the internal wall would be placed very close together, occupying some room space unnecessarily and it looks quite ugly and unnatural. It's highlighted in red in Figure 7. At this point, we were not sure if it was an artefact from our adaptation or just because the flaws in the original algorithm got exposed when deploying it to a larger shape.

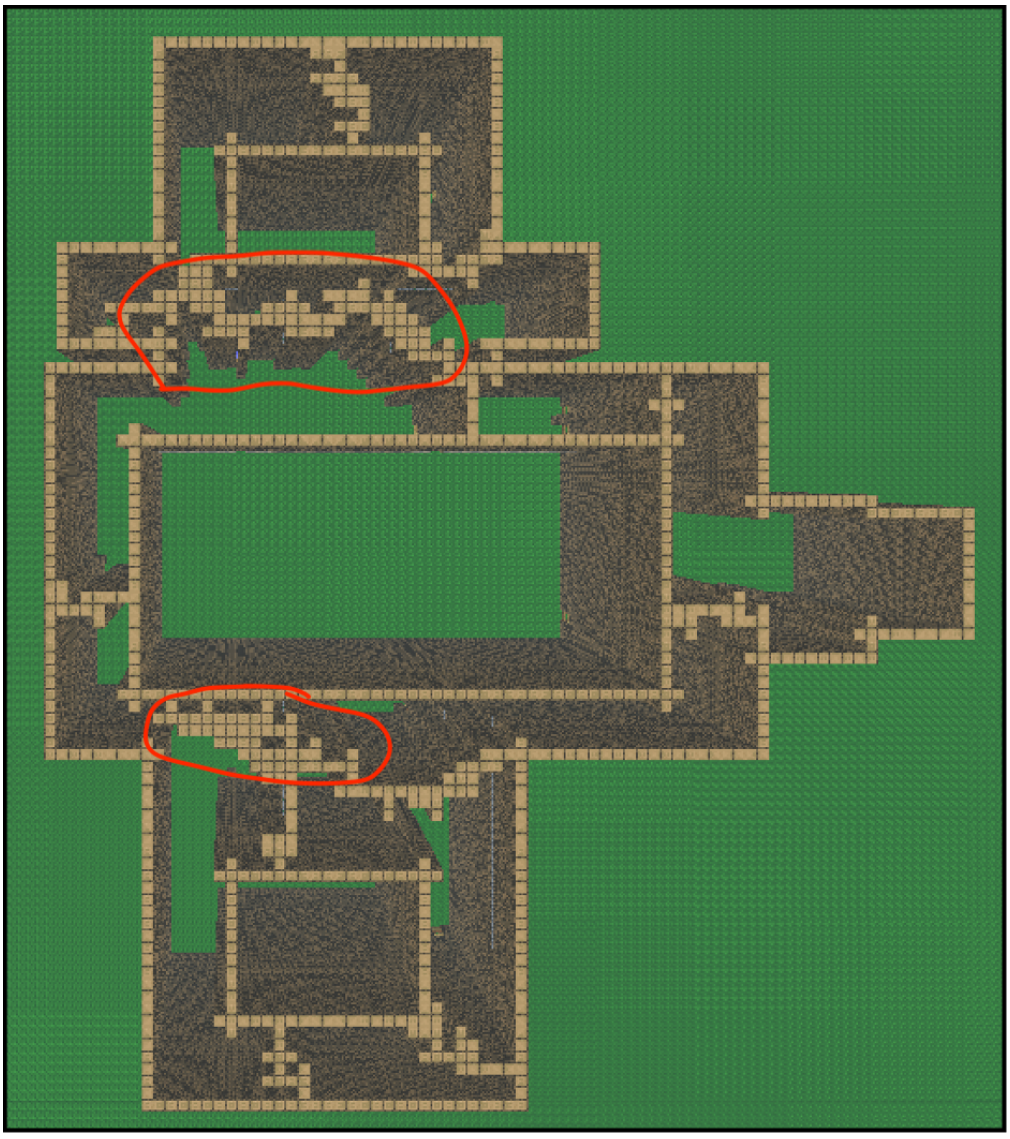


Figure 7: An example of the generated structure in Minecraft.

Also, another thing we did not expect was that if we run the floor plan algorithm on a footprint with the hole in it, the algorithm placed a single door on only one lap of the external walls. However, we need at least one door on each lap of the external walls for the player to traverse the whole area. Figure 8 highlights two different laps of external walls. Since we realised that quite late, we fixed that in some sense by putting more external doors randomly rather than just one so that it is more likely that each lap would have a door installed.

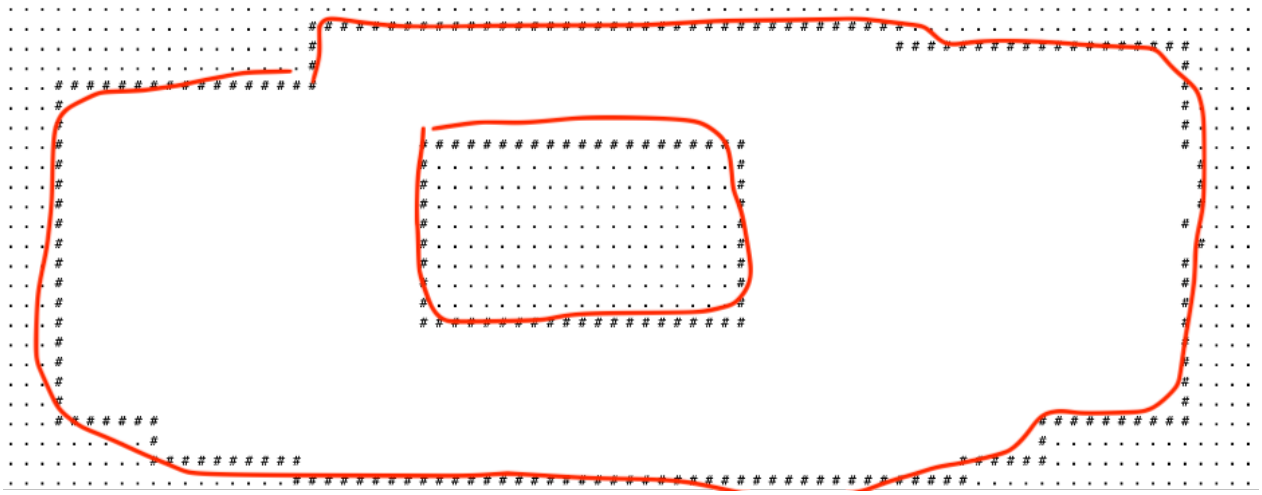


Figure 8: An example of the footprint with 2 laps of external walls.

Lastly, the running time of the adapted algorithm turned out to be quite slow when applied to large structures. The algorithm was run for several times with the same 100×100 footprint but different numbers of rooms. A set of data was collected and plotted in Figure 9. We observed a linear relationship between running time and the number of rooms. Roughly, the running time would increase by 10 seconds for each extra room. This implied that growing rooms is the most time-consuming process in the workflow. Therefore improving it would boost the performance of the algorithm.

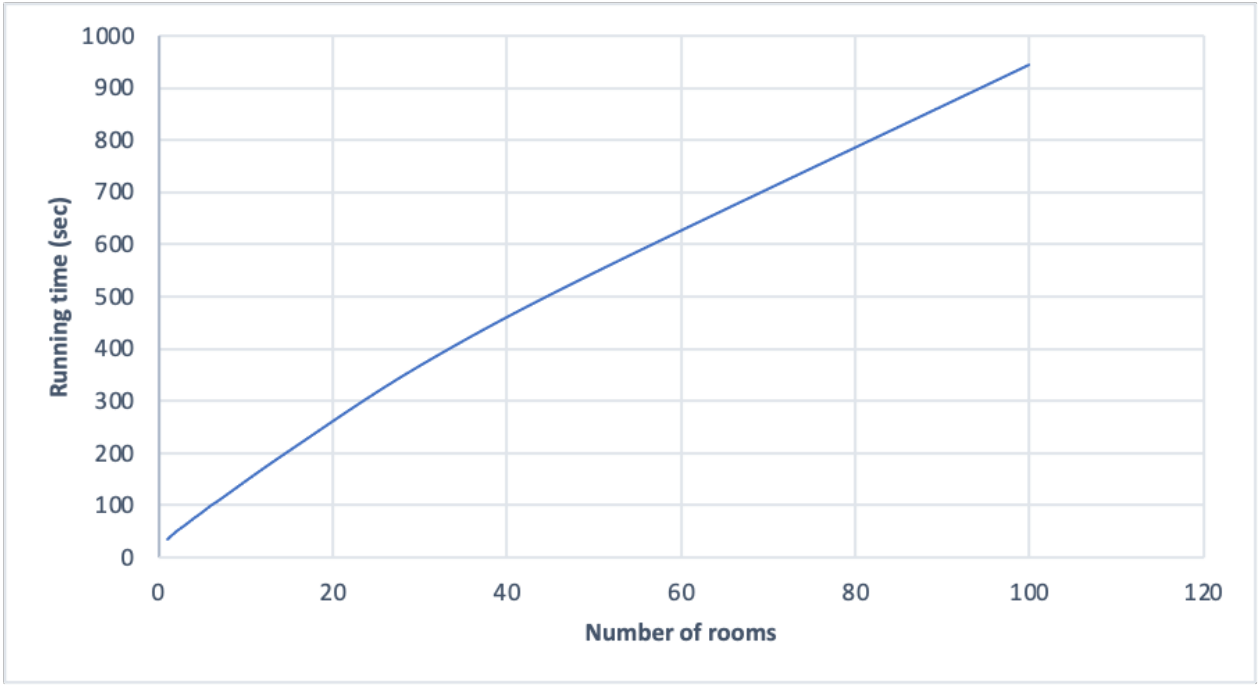


Figure 9: This is a plot of running time of the algorithm against the number of rooms to start with.

7 What would be the next steps for your research? (3 marks)

There are many potential future work to do for our research. As we talked about above, we can dig deeper into the algorithm to see what's causing the wall distortion problem and fix it accordingly. For the external placement problem, we can fix it by adding one more function to record the location of different laps of external wall in the matrix and then place one door along each of them.

Apart from that, we think one of the most important improvements we should do is to smooth the wall to segments of line after the growth. As a result, the shape of each room will be more regular and natural. Figure 10 shows the idea of fixing the internal walls. We believed that it could be done with some smart boundary checking and replacement. More concretely, it starts from one end of the path, checking if there is a block down the path. If there is, it moves to that one. Otherwise, it checks if there is any block adjacent (horizontally) to the empty block. If so, it moves that to the empty block. Else, it puts one there and so on.

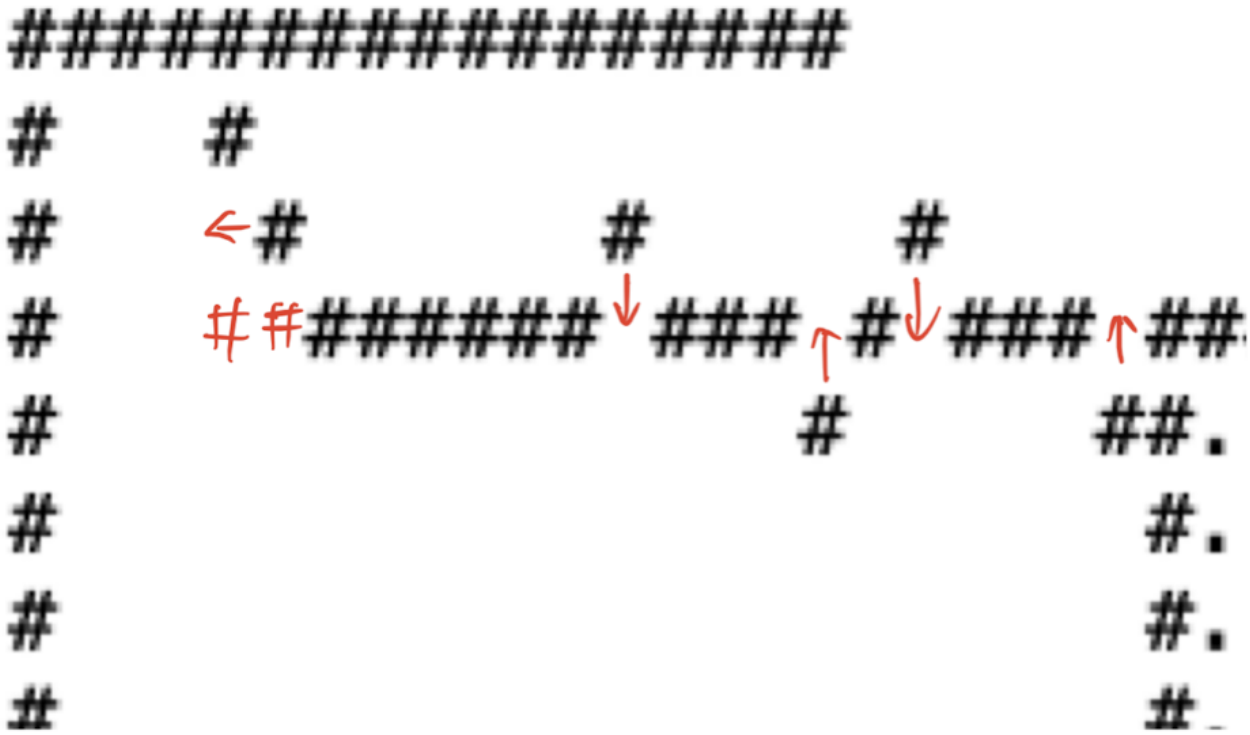


Figure 10: An example of fixing some (distorted) internal walls.

Another interesting improvement is the novelty of the generated structure. We intend to improve the algorithm and assign each room a certain function, such as bedroom, living room, kitchen, laundry room and etc. Based on that role, we might assign the living room a larger weight than laundry room to make it more likely to be chosen to grow because the living room tends to be bigger than laundry. Other than that, we might put customized item in rooms with different roles. For instance, one 2*1 bed and one 2*2 window must be placed in the living room. To do that, we have to introduce new symbols in our floor plan matrix. However, if we want to encode that window information in our floor plan matrix, we must change the encoding from 2D matrix to 3D matrix. This will open us to more decoration options on vertical space including but not limited to hanging various painting on the internal wall, putting the fireplace in the internal wall, placing cabinets in the internal wall and so on (given these items are supported in games).

8 What have you learned from your research project? (3 marks)

Thanks to our study effort, we have a thorough grasp of PCG, its application in games, and the present status of procedural building generation. We have surveyed a lot of PCG methods aiming at generating buildings and structures for virtual worlds. We have observed that procedural building generation is a well-developed

PCG area. The majority of approaches employ a formal rewriting system to generate a 3D building model from a 2D parcel shape, such as an L-system, a split grammar, or a shape grammar. These techniques may be used to produce intricate and believable structures, but they demand a significant amount of time and effort. Alternative approaches try to automatically recreate grammars from real-world data sets like photos of building façades.

Despite the fact that there has been a lot of study done in this field, there has yet to be many practical implementations in commercial games. The challenges of controlling output and the disparity of separate specialised procedures have hampered the acceptance and application of PCG methods. In modern games, time, repetition, similarity, functionality, and other limitations make it difficult to generate high-quality content to meet the expectations of players for a variety of structures.

These flaws also indicate that there is still possibility for development, and successful PCG techniques in game have shown significant commercial value. We'll concentrate on improving and refining current algorithms.

In terms of general skills, this research project has practiced us on the ability to find a research direction, quickly familiarise ourselves with the literature and techniques of a field in which we had little prior experience, and come up with a new idea. Additionally, collaborative research is challenging in this challenging time, and this project gave us many opportunities to learn how to work on one research project collectively and how to communicate effectively to maintain complementary work without overlapping or conflicting.

References

[1] Daniel Camozzato, Leandro Dihl, Ivan Silveira, Fernando Marson, and Soraia R Musse. Procedural floor plan generation from building sketches. *The Visual Computer*, 31(6):753–763, 2015.

[2] Jonas Freiknecht and Wolfgang Effelsberg. A survey on the procedural generation of virtual worlds. *Multimodal Technologies and Interaction*, 1(4):27, 2017.

[3] Michael Cerny Green, Christoph Salge, and Julian Togelius. Organic building generation in minecraft. In *Proceedings of the 14th International Conference on the Foundations of Digital Games*, pages 1–7, 2019.

[4] Evan Hahn, Prosenjit Bose, and Anthony Whitehead. Persistent realtime building interior generation. In *Proceedings of the 2006 ACM SIGGRAPH Symposium on Videogames*, pages 179–186, 2006.

[5] Mark Hendrikx, Sebastiaan Meijer, Joeri Van Der Velden, and Alexandru Iosup. Procedural content generation for games: A survey. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 9(1):1–22, 2013.

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

[7] Ricardo Lopes, Tim Tutenel, Ruben M Smelik, Klaas Jan De Kraker, and Rafael Bidarra. A constrained growth method for procedural floor plan generation. In *Proc. 11th Int. Conf. Intell. Games Simul*, pages 13–20. Citeseer, 2010.

[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, 2010.

[9] Jess Martin. Procedural house generation: A method for dynamically generating floor plans. In *Proceedings of the Symposium on Interactive 3D Graphics and Games*, volume 2, 2006.

[10] Paul Merrell, Eric Schkufza, and Vladlen Koltun. Computer-generated residential building layouts. In *ACM SIGGRAPH Asia 2010 papers*, pages 1–12. 2010.

[11] Mojang. What is minecraft?, Apr 2020.

[12] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. In *ACM SIGGRAPH 2006 Papers*, pages 614–623. 2006.

[13] Andrew Rau-Chaplin, Brian MacKay-Lyons, and P Spierenburg. The lahave house project: Towards an automated architectural design service. *Cadex*, 96:24–31, 1996.

[14] Michael Schwarz and Pascal Müller. Advanced procedural modeling of architecture. *ACM Transactions on Graphics (TOG)*, 34(4):1–12, 2015.

[15] Ruben M Smelik, Tim Tutenel, Rafael Bidarra, and Bedrich Benes. A survey on procedural modelling for virtual worlds. In *Computer Graphics Forum*, volume 33, pages 31–50. Wiley Online Library, 2014.

[16] Tim Tutenel, Rafael Bidarra, Ruben M Smelik, and Klaas Jan De Kraker. Rule-based layout solving and its application to procedural interior generation. In *CASA workshop on 3D advanced media in gaming and simulation*, 2009.