Game Science Report : Floor Layout Generator

Pawan Alur
Rutgers University
pka21@scarletmail.rutgers.edu

ABSTRACT

There are many games with indoor levels that make use of randomized floor layouts. Alongside that, there are a lot of research that focuses on fields like evacuation, path-finding etc. that assume that they have a large data set of floor layouts. In this project, my goal was to build a floor layout generator that creates layouts following certain restrictions.

KEYWORDS

Procedural Content Generation, Floor Layout,

1 INTRODUCTION

To generate floor layouts by hand takes a lot of effort, time and expertise. While it is possible to get hold of floor plans from buildings that already exist, it is difficult to gather thousands of such plans. Thus, a lot of research for fields where there is a need for floor plans like evacuation in case of emergency, AI testing, crown simulation etc. need a source of floor plans. Procedural Content Generation is essentially generation of content using code rather than having a human make it by hand. This allows for a much shorter storage space than having to store that whole level data itself. A very simple example is how storing equations of 3 lines would take much lesser memory than storing an image of the triangle itself. Procedural Generation has been used in video games for a long time to generate content like dungeons, levels and even whole worlds like in Minecraft. This project was to build a floor layout generator that uses Procedural Generation of a floor layout efficiently. While there are papers that I found that generated floor layouts, most of the research in this field had to do with development of other game play elements. The reason procedural generation is so much more desired in the project rather than choosing from a bunch of pre-made floor plans are as follows :

- It requires much lesser memory, upto 1000 times lesser memory
- It requires less time to write an algorithm that generates floor plans rather than have a person constantly make new plans.
- In research that uses Machine Learning to get better, that more the variety, the better it's results will be at the end. Procedural generation can create an almost unlimited supply of plans that all have no bias towards specific features as a human is bound to have.
- It can create floor plans that a human would almost never think of, simply because it may be to complicated, or may break social convention.
- It is easy to mould it to just the required parameters and nothing else.

The report is divided into 4 sections : I will give an overview of research that inspired this project. I then write about the algorithm used to generate floor layouts and then showcase the different examples generated using this method. I end the report with the limitations of this project and the possible extensions.

2 RELATED WORK

For a lot of time, there wasn't much work done in the field of Procedural Content Generation. One of the reasons that this wasn't used too much was because it shifted the power to create exactly what a developer wanted to a program, which may make mistakes and there would be no way to detect it. However, ever since a few popular games have used Procedural Content Generation to generate dungeons successfully, there has been quite recently in this field. Related to my project, the one source of information I found very useful was [3]. This is a survey of all recent search-based Procedural Content Generators. This gave me a lot of basic information in the field and also introduced Search-Based Generators in detail. It explains the various types of Procedural Content Generation classifications and what are the differences between them. This allowed me to be conscious of what type of algorithm I wanted and focus on building it to the image I held in my mind. Two papers that I found about Floor Plan generators were [4] and [2]. They are two completely different ways to generate floor plans and each have their own pros and cons to the way floor layouts are generated. [1] was a really great algorithm I found online, and something I used as a basis for my project. While [1] is an dungeon generation algorithm, much of it could be generalized for anything that requires rooms. Floor layouts can be easily made as long as you get the rooms in the right place. The way this algorithm worked was to generate rooms as a square with a random height and length. These are then separated completely from each other to using a separation steering behaviour. As they use Park-Miller Normal Distribution, the sizes are skewed towards rooms with a smaller size. They then distinguish based on area(height*length), which rooms should become main rooms and which should become roads. There are various parameters here that could be changed to get a result more like how you want it. There is a ratio one could maintain between height and length to obtain rooms of specific sizes, cap how huge the whole thing can be by fixing spawn points and so on. After choosing the main rooms, the algorithm then goes on to use Delaunay Triangulation to construct a graph with these rooms as vertices. Then, they solve the graph for a MST and add all the smaller rooms in these edges as path. This works very well as a simplistic dungeon generator, but I needed to make a lot of changed to allow for it to look like a floor instead.

3 GENERATION METHOD

3.1 Representations :

There are three main objects that are used to make the algorithm easier. They are :

- Block
- Room
- Floor

3.1.1 Block. A block is treated as a basic unit in this algorithm and is a simple rectangle. It has the following fields :

- Length
- Width
- Position of center as (x,y) pair
- Minimum X position
- Maximum X position
- Minimum Y Position
- Maximum X position
- Door Location
- Direction

As the goal of the project was that every single block be connected to a wall, the direction specified which wall was this specific part of the floor was this block in(up/down/left/right). The Door position stores the point (X,Y) exactly opposite to the wall. So should this block be chosen to hold a door, this is where it would be instantiated. A sample block can be seen in Fig. 1.

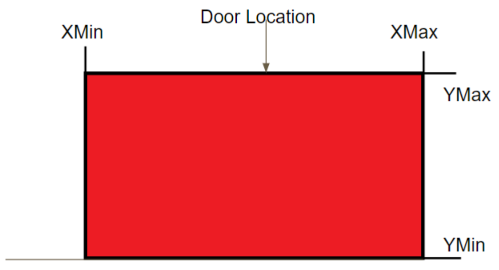


Figure 1: A block

3.1.2 Room. A room is simply a collection of blocks that intersect. By having many blocks intersect, we can generate interesting shapes to form rooms. So, every single time a block encounters a room that it is intersecting with, it simply merges itself to form part of the room by expansion. For example, in Figure. 2, we can see that there are 4 blocks. Block 1,2 and 3 form part of the same room. However since block 4 does not intersect with then, it forms a room to itself.

We then remove all borders in the room to get the final figure as can be seen in Figure 3.

The fields in a room are :

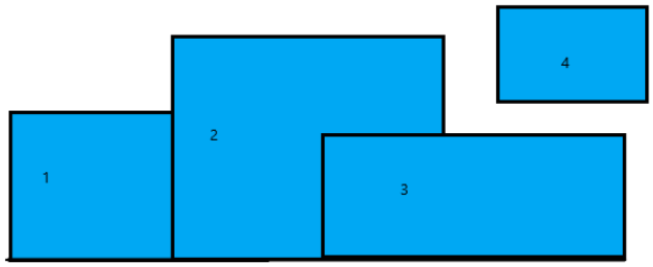


Figure 2: Merging Blocks to form a room

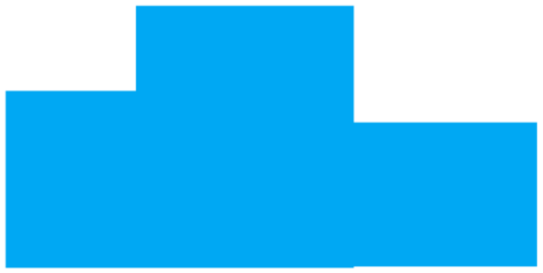


Figure 3: Final Room Created

- A collection of blocks
- A block where we must place doors.

The way the number of doors in a room work is based on the number of edges of the floor it covers. Thus, a room can have at most 4 doors. The way I decide which blocks have the door is fairly simple and intuitive. For every wall that the room is connected to, we find the the door positions of every block connected to the wall. We then choose the block that had the position furthest away from the wall. Thus, for the room in figure 3, we choose the door location indicated by block 2 to be the door for our room.

3.1.3 Floor. A floor is the simplest data structure yet. The floor is set to to be rectangular and it spans between the largest and smallest X and Y position of each block. It consists of the following values :

- Min X direction
- Max X direction
- Min Y direction
- Max Y direction
- Center of the room as (x,y) values

3.2 Algorithm

Now that we have the required knowledge about the 3 data structures I used in the algorithm, I will proceed and explain the algorithm. At it's base, the algorithm is very simple. It makes sure that there is nothing left from previous iterations by clearing the buffer and deleting everything from all lists we have created. Then it generates everything we need : Blocks, floor, Rooms, Doors, and Walls.

```
Algorithm 1: Main Loop of the Algorithm
  Data: Number of Blocks
  Result: Floor Layout
1 Create_Blocks();
2 Create_Floor();
3 Move_Blocks();
4 Create_Rooms();
5 Create_Doors_And_Walls();
```

3.2.1 Create Blocks. To create blocks, We use 3 inputs : A given radius, a maximum length and a maximum height. This allows us to draw a blocks as follows :

```
Algorithm 2: Create Blocks
  Data: Radius, max. Length, max. Height
  Result: Block
1 while number of blocks to be drawn do
2     find a random point inside circle;
3     choose random $x, y$ such that x is less than max. length
        and y is less than max. height.;
4     Draw a block at point with size x and y.;
5     Calculate All parameters.;
6 end
```

To find a random point in a circle, we simply choose a random angle between 0-360 and choose a random length between 0-radius. An example can be seen in Figue 4.

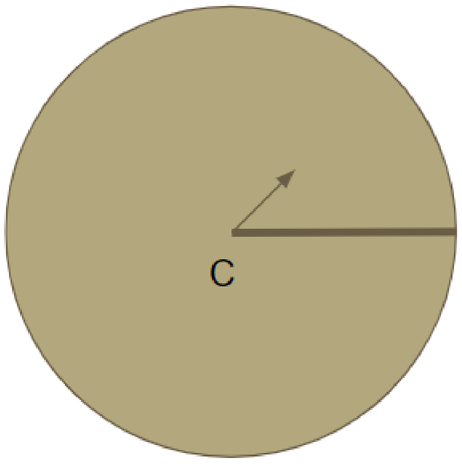


Figure 4: Choosing a point inside a circle

3.2.2 Create Floor. We create a floor by calculating the furthest limits based on all blocks and drawing a rectangle with these as values.

```
Algorithm 3: Create Floor
  Data: List Of Blocks
  Result: Floor
1 while i is in number of blocks do
2     minX = min(minX, block[i].minX);
3     maxX = min(maxX, block[i].maxX);
4     minY = min(minY, block[i].minY);
5     maxY = min(maxY, block[i].maxY);
6 end
7 Draw Floor with these 4 as floor walls;
```

3.2.3 Move Blocks. We move blocks by comparing the min and max positions of the block by the floor min and max and then moving it in the desited direction.

```
Algorithm 4: Move Blocks
  Data: List Of Blocks
  Result: Blocks In Final Position
1 while i is in number of blocks do
2     minX = abs(floor.minX - block[i].minX);
3     maxX = abs(floor.maxX - block[i].maxX);
4     minY = abs(floor.minY - block[i].minY);
5     maxY = abs(floor.maxY - block[i].maxY);
6     min,Direction = min(minX,maxX,minY,maxY);
7     move block in Direction by min;
8 end
```

3.2.4 Forming Groups. We form groups by checking if the block intersects with any group member or not. If it does not, then it forms a new group of it's own.

```
Algorithm 5: Create Group
   Data: List Of Blocks
   Result: List of Groups
1  while i is in number of blocks do
2      if block[i] intersects with a member of group then
3          if block[i] is already a member of a group then
4              delete block[i];
5          else
6              add block[i] to said group;
7          end
8      else
9          create new group and add block[i] to new group;
10     end
11 end
12 For all groups, find door locations;
```

The reason why we delete a block if it exists in another room is so that there are no conflicts between room. Another possible alternative is to merge the two groups, but doing so would result in too few rooms

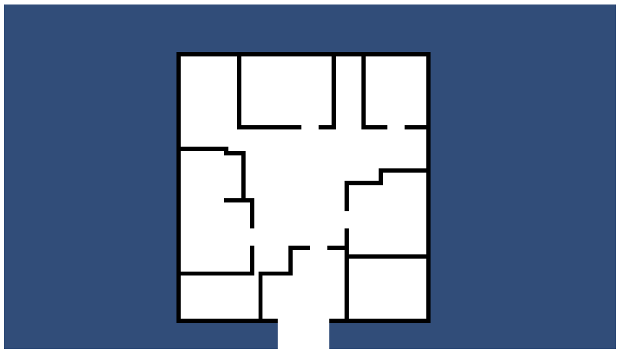
3.2.5 Drawing Doors and Walls. We simply calculate for every room the location of the doors as mentioned above by choosing the max distance from the walls. Drawing walls is slightly more tricky. We have to use the 4 vertices or each rectangle to find the equations of the 4 sides. Then for every pair of of blocks in the room, we find points of intersection. Find a set of points of intersection and vertices. Drawing all vertical and horizontal lines between pairs in this set give you walls for the group.

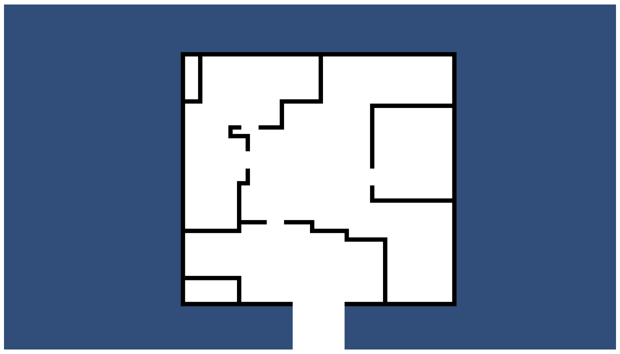
Algorithm 6: Draw Walls and Doors

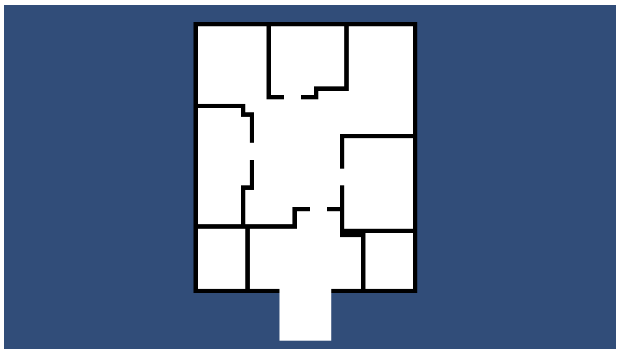
```
Data: List Of Rooms
Result: Walls and doors
while i is in number of Rooms do
    Draw doors of block[i];
    Let SetOfPoints be set of all vertices of all blocks;
    while j,k in blockset of Room[i] do
        if block[j] and block[k] intersect then
            Add points of intersection to SetOfPoints;
        end
    end
    while j,k in SetOfPoints do
        if j.x == k.x OR j.y == k.y then
            Draw a line from j to k
        end
    end
end
```

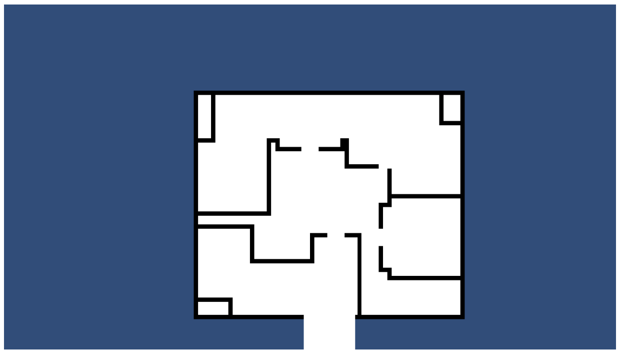
4 IMPLEMENTATION

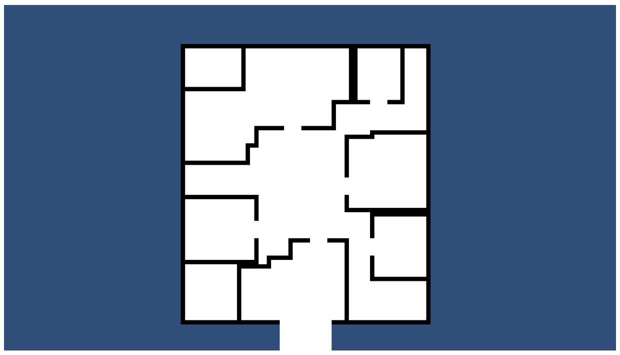
This project was implemented in Unity. This is because it trivialized the visualization aspect of this project and I could focus on the core algorithm instead. Also, it had a feature called layers that allowed me to choose what would be displayed above what(for eg. walls, doors and rooms must be displayed above the floor). The following figures show some of my results.











5 CONCLUSIONS AND FUTURE WORK

There are many limitations to my project, which can be improved on.

- There are very few ways to constrain the design : Other than max height/length and radius of a circle, I have randomized everything else. To add more constraints(certain rooms must be next to each other and so on) less randomization is needed. Also, as I did not study any rules in floor layout construction, it is very possible that we get fewer rooms than needed.
- It is possible to get 'closed areas' when two rooms block the area completely(see the bottom left corner of the last example). This should be avoided. A simple way to fix this would be to check all places if they are visitable and if not, add a door at the boundary.
- I have fixed the location of doors on a box, making it more dynamic could add to the algorithm.
- It only works for rectangular floor layouts. However, many buildings are not rectangular and thus I need to improve the program by allowing the existance of non-rectangular areas. This could be done by not snapping rooms in the 'Move Block' stage, but instead avoiding 1D collisions in both directions(first horizontal and then vertical).

REFERENCES

[1] A Adonaac. 2015. Procedural Dungeon Generation Algorithm. https://www.gamasutra.com/blogs/AAdonaac/20150903/252889/Procedural_Dungeon_Generation_Algorithm.php. (2015). [Online; accessed 03-September-2015].

[2] Ricardo Lopes, Tim Tutenel, Ruben Michaël Smelik, Klaas Jan de Kraker, and Rafael Bidarra. 2010. A Constrained Growth Method for Procedural Floor Plan Generation.

[3] Julian Togelius, Georgios N. Yannakakis, Kenneth O. Stanley, and Cameron Browne. 2011. Search-Based Procedural Content Generation: A Taxonomy and Survey. *IEEE Transactions on Computational Intelligence and AI in Games* 3 (2011), 172–186.

[4] Martin D. F. Wong and C. L. Liu. 1986. A New Algorithm for Floorplan Design. *23rd ACM/IEEE Design Automation Conference* (1986), 101–107.