Interactive Modelling of Procedurally Generated Indoor Spaces

Susant Pant, Camilo Talero, Shannon Tucker-Jones

Introduction

Procedural modelling, within the context of urban and suburban buildings, has mainly become used for the large-scale modelling of cities. With its efficiency, what could have taken weeks to sculpt by-hand now renders in minutes, and each time completely differently. For example, Müller, Wonka et al.'s 2006 paper "Procedural Modeling of Buildings" displays a context-sensitive, rule-based procedural generation method that creates complex buildings from much smaller elements; even allowing the creators (albeit with a very large ruleset) to mimic the historic city of Pompeii without modelling an inch of it themselves. As early as 2001 we have seen papers such as Parish and Müller's seminal work "Procedural Modeling of Cities," which demonstrates the simple but outstanding power of procedural generation by building a network of streets and lining roughly 26,000 randomly generated buildings along them, creating a completely computer-generated skyline.

However, we feel that procedural architecture should be more than simply the output of a computer's algorithm. After all, there is the notion of human aesthetic to keep in mind. More importantly, many of the current implementations of procedural architecture tend to use it only as background rather than as the main content. It is in service of making the world feel larger, but not all too deeper. Many of the buildings are simply hollow shells. Technology has grown since 2001, however, and researchers have since made advances into making the interiors - not just the exteriors - feel like a genuine environment.

Unfortunately, interior procedural modelling has not had the focus, depth, or popularity in academia that the complex exterior designs, which researchers like Wonka push forward, have enjoyed. Most of this lies in the complexity of building interior structures which must obey the larger, more grounded rules (beyond those of procedural algorithms) of being a habitable, comfortable and realistic space, rather than simply mimicking their exteriors. As such, very little suitable material has been published on procedurally building an entire structure with the sturdy backbone of a believable floorplan; perhaps the closest and most visible was the one we used as the basis for our project, Jess Martin's "Procedural House Generation: A method for dynamically generating floor plans." In it, Martin describes a simple - as is all procedural generation, in concept - but illuminating 'algorithm' for creating indoor spaces reminiscent of houses that satisfy the basic requirements listed above.

Overview

In this project we aimed to create a procedurally generated 3D building, specifically a house, with an interior structure and well-defined floor-plan. Our aim was to create a step-by-step method

which carefully finds the best way to build a realistic house, with respect to the foundations laid down by Jess Martin's paper. On top of that, we wanted to integrate interactivity with the floor-plan that our algorithm developed so that the user's final result is not 'just' a cluster of rooms in close proximity to each other, but also has aesthetic appeal to the user. To achieve this end, we also developed a criteria of interactive features that such a program would need. This was necessary so that the interactivity of the program did not completely overshadow its procedural nature, and that instead of using the tool for its quick development of house designs, one may become too invested in the aesthetics of the house.

Our basic criteria is as follows:

1. The ability to relocate any room, as specified by the user.
2. The ability to relocate connections (doors) between rooms, if the user believes the current position is unfit.
3. The ability to scale change the dimensions of a room.
4. The ability to subdivide a room's walls to create rooms with more than 4 walls.
5. The ability to rotate rooms.
6. The ability to delete walls that connect with other rooms.
7. Change the type of a room.
8. Change the connectivity of the underlying graph structure.

This criteria was mainly used to guide and prioritize which interactive features should be included in our final project. Not all of them were implemented, but we believe the features that have been included are of a wide variety that allow the user to output interesting, unique, and well-designed houses at a high turnover rate.

From there, we used Martin's paper on floor-plan generation as a guideline to create our own basic floor-plan. However, Martin's paper did not provide an in-depth description on this method, and so we had to develop, through our own intuition on Martin's words, his 'algorithm'. We started by creating a graph as described in the paper, and then placing this graph in 2D space in such a way that every node was as far away as feasibly possible. From those node positions, the rooms expand in space either until they occupy an area equal to their size, or they have hit against other rooms. This provided a basic floor plan.

The interactivity of our tool plays its part here, as users can now move rooms around, move the positions of doors, scale room sizes and delete walls so that the house initially created under random circumstances resembles a more aesthetically pleasing and natural environment. From there, the user can change the 2D floor-plan to a 3D equivalent, with actual walls that have the ability to make gaps in them for windows or doors. To complete the image of a simple house, one may add a 3D roof to the top of the structure. The roof was constructed using Dahl and Rinde's paper on procedural generation of interiors, although the algorithm described in that paper has been appropriated from hallways to the triangular shapes most associated with roofs.

Martin’s Algorithm

In this section, we define our implementation of Martin’s algorithm.

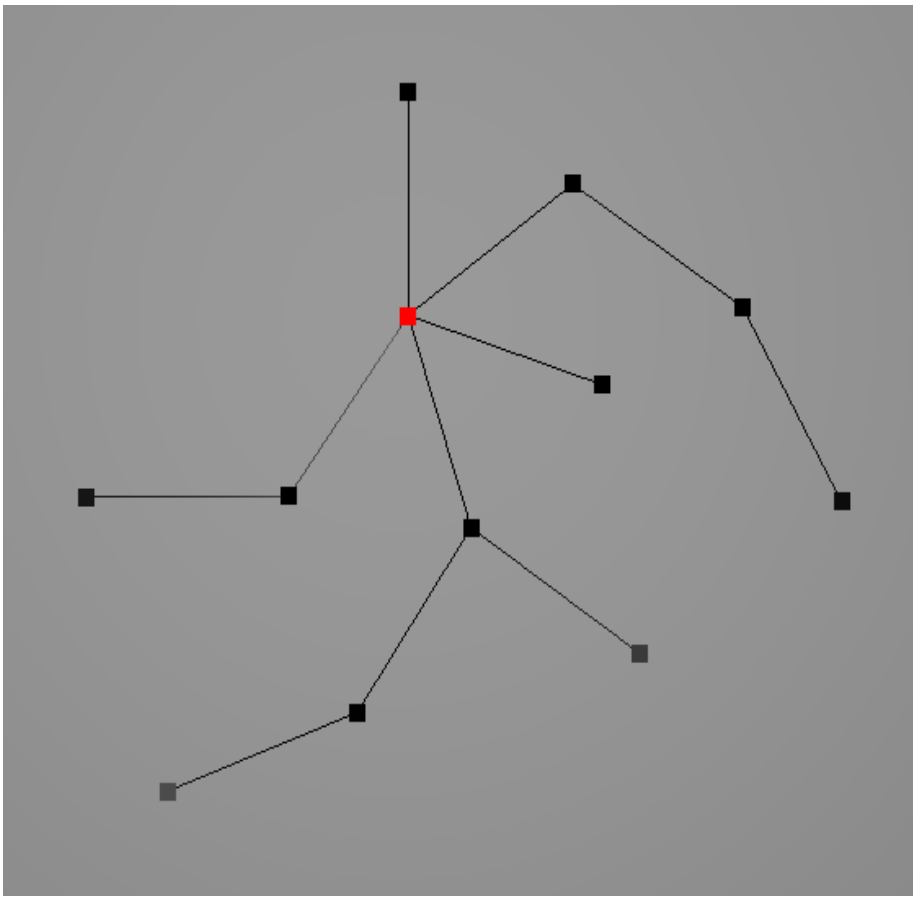
Martin starts his procedural house generation method by generating a graph, which is to hold all the connections between the rooms. We first add public rooms to the graph, which are defined as rooms that everybody in the house should have access to. These are locations such as kitchens, dining rooms, living rooms, etc. In our program, we have identified such rooms with the colour green. Martin adds the ability to specifiy a room’s function through statistics (such as giving one public room the label of the dining room and another one the living room), but we believe this went against our 7th criteria as we wanted the user to freely define the space that they were given to them.

Next Martin adds private rooms, such as bedrooms, which can only be accessed from public rooms. We identified such rooms with the colour red. Finally, the graph is connected with extra rooms, or “stick-on” rooms, such as closets or pantries, which may be added on to any rooms. Such rooms are yellow in our program. They also do not stop the expansion of public rooms, causing the public rooms to pass right through their walls, as we believed extra rooms gave an interesting twist to the idea of a room’s space (by being a separate room while still part of that room). Also, it provided an interesting method of developing room shapes that were beyond simply rectangular.

This graph structure has so far been simple, and was easy to implement. However, Martin’s paper chooses to gloss over an incredibly important matter in distributing those graph nodes across 2D space. This step ensures that, when it comes to expanding the rooms, none of them end up being tightly packed between two or three (or more) rooms and never finding the ability to grow beyond a paltry size. We describe our solution here:

1. Every node is randomly placed on a 2D space, except for the root which is placed at (0,0).
2. For the root node, we space out its children by 270 degrees.
3. We do a breadth-first search on the graph. For every new node being examined, we spread its children out over a 180 degree fan in the direction away from its parent.
4. We then normalize the distance from the parent to the children so that all rooms are roughly equally spaced form each other.

This simple algorithm may not satisfy every node’s distance from each other for all cases (i.e. some may still spawn right next to each other), but it is a largely consistent way of satisfying the necessity for distant nodes. Furthermore, Martin’s statement that every node is an “equal distance from other children and an equal distance from the root” specifies nothing about the process but matches the intuition of our algorithm.



The skeleton of the graph’s nodes, with the origin in red.

We now expand the rooms to fill their respective areas. Expansion occurs in all four cardinal directions from the position of the room’s node until the room hits another room, or its maximum area is reached. Martin discusses his “pressure” system, which compares the pressure inside and outside the room and accordingly increases the size of the room (if the outside pressure is bigger, the room simply stops growing). Martin recommends a Monte Carlo method to “choose which room to grow or shrink next,” but we found it more prudent to simply expand them all at once and see which collide first. Similarly, rather than enforcing a ‘pressure’ mechanic as discussed by Martin, we simply tell the room to stop after colliding with another as we found the effects to be near identical. However, to better ensure that a room manages to keep a connection with its parent (from whom the child gets its only door, at this point), we let the room expand at a naturally faster rate toward its parents, before being blocked or encountering the parent’s room.

At this point, the algorithm for finding the 2D floor-plan has finished its function, and is no longer necessary. Features such as expansion will continue to occur in the program as the user edits the floor-plan to their desire, causing new free space to open up; however, the positions of the nodes are not recalculated every time a node is moved.

The 8 Criteria of Interactivity

While adding interactive features to our program, we found ourselves considering the nature of our task. What was the point of editing a randomly generated floor-plan if one can edit it to the point where creating it from scratch may have been faster? In essence, we did not want the user to be so involved with editing their floor-plan that they lost track of the telos of procedural modelling: the rapid generation of content that would otherwise be long and tedious. As such, we limit the amount of interaction that the user is permitted, in accordance with at least a basic criteria of necessary interactions to make the editing features worthwhile.

Our basic criteria follows 8 simple modifications a user can make:

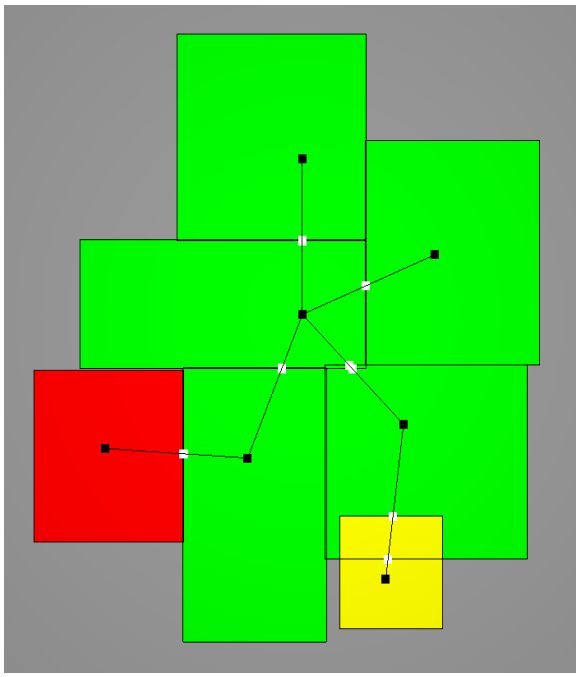
1. The ability to relocate any room, as specified by the user. This was perhaps the most significant, as we found the idea of a customizable indoor space that was bound to a particular shape was paradoxical and completely counter-intuitive.
2. The ability to relocate connections (doors) between rooms, if the user believes the current position is unfit. This comes from the relocation of rooms, which may cause certain door positions to resolve at nonsensical locations.
3. The ability to scale change the dimensions of a room. This follows from the same intuition as room relocation: if a room cannot be customized to even this degree, then there is little need for such a level of customization.
4. The ability to subdivide a room's walls to create rooms with more than 4 walls. This came from the idea that not all rooms need be rectangular, which we saw was the case with Martin's algorithm. However, we were unable to implement this feature due to time restraints.
5. The ability to rotate rooms. This follows from the argument for scaling and rotating rooms, but we found that without the ability to subdivide walls to account for particular shapes, rotating walls by angles less than 90 degrees looked awkward, and a 90 degree rotation itself could be simulated by simply rescaling the walls.
6. The ability to delete walls that connect with other rooms. This allowed greater flexibility in the number and shape of the rooms without outright erasing rooms from our graph, which would go against our principle of focusing on the procedural nature of the program first over its editing functionality. On the flipside was the ability to add rooms, which we also felt undermined the procedural quality of the final product.
7. Change the type of a room. With our current implementation this is a very simple addition to the code, which we decided not to prioritize while working on this project. We will discuss its implementation below as though it is part of our program, for it is - in spirit.
8. Change the connectivity of the underlying graph structure. This, like the previous point, would be a very simple addition that, at the point of submission, did not have a high priority for us. We will also discuss its implementation below. It mainly serves the purpose of adding more doors between rooms. Do notice however that every room must have at least one door leading to it; so if room connections are to be deleted, then at least one other connection must still remain.

As discussed above, not every single one of the nine criteria was implemented - mainly due to time restraints. We believe that the ones we did implement were the most significant to display the

possibilities of editing a procedurally generated space. We will now describe the general method required to satisfy the criteria we included in our program. To do this, we introduce one of our most significant classes with which so much of every house is built, Room.

Criterion 1

Room has a tuple that holds the position of the node, as well as two other tuples that hold the top right and bottom left positions of the rooms. In order to move a room around in 2D space, as described in criterion 1, the user moves the position of the node by mouse; this displacement is also added to the position of the top right and bottom left positions, thus moving the room as a whole.



The base floor-plan, the result of Martin's algorithm without any user editing.

Criterion 2

To satsify criterion 2, things get a bit complex. Firstly, it is important to understand that we have defined the position of a door between two rooms as the intersection between the edge that connects the two nodes, and the wall that separates the two of them. However, since each room has one wall each, each room must also have a copy of the door's location with respect to their own wall. To store the data of all of a room's possible doors, Room has a std::vector which holds the positions of the doors. This is visible in our program as the white dots between walls. So, if we would like to manipulate the location of a door, all the user must do is to manipulate the location of the node without manipulating the rest of the room, so that the intersection point (the white dot) is updated to a new location – which is to be the new location of the door.

Criterion 3

Scaling a room is very simple in that all one needs to do is grab the bottom left or top right corner, and directly manipulate its position to make the room larger or smaller. However, if a room is of

a smaller size than its area, then the room will start pushing back, as the expansion from Martin's algorithm is still in effect. Therefore, if one desires to make the room smaller than its actual size, they may find it more helpful to pause the expansion first (using the spacebar. See the User Manual for more details on usage). So, we have also identified an effective, if a little inelegant, method to satisfy criterion 3.

Criterion 6

Satisfying criterion 6 requires Room to have an array of 4 booleans, each of which is associated with a wall. When the boolean is false, the wall stops rendering, thus producing the effect that the wall has disappeared. This also stops rendering any doors on that wall, as the door would no longer serve a purpose.

Criterion 7

We know that every Room object must have a type, as that value is used for initializing the flor-plan. Therefore, all one needs to do to satisfy criterion 7 is simply to change that value of type according to user input.

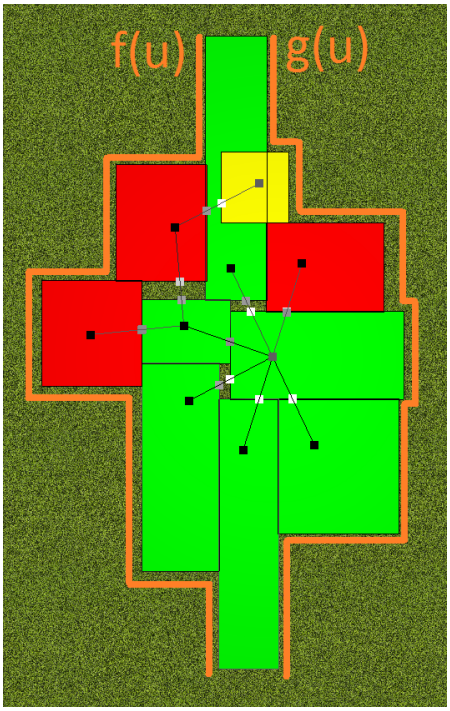
Criterion 8

Room also has a std::vector of Room obejcts with which it is a neighbour. This is used in calculating the position of doors and performing a breadth-first search in the adjacency list representation our program uses. In the case of criterion 8, if a user selects two rooms with a special modifier, both rooms would have the other added as a neighbour in the std::vector of neighbours. This automatically adds a door between these two rooms.

With these 6 criteria specifically, we believe the user will have a wide and varied playing field that will not distract them from the ulterior purpose of creating models quickly and elegantly. Of course, the other two would greatly increase the flexibility of design without sacrificing large chunks of time in carving out the details of a particular architecture, but these 6 in particular exemplify the ideal by being very simple transformations in and of themselves.

Determining Other House-Like Properties

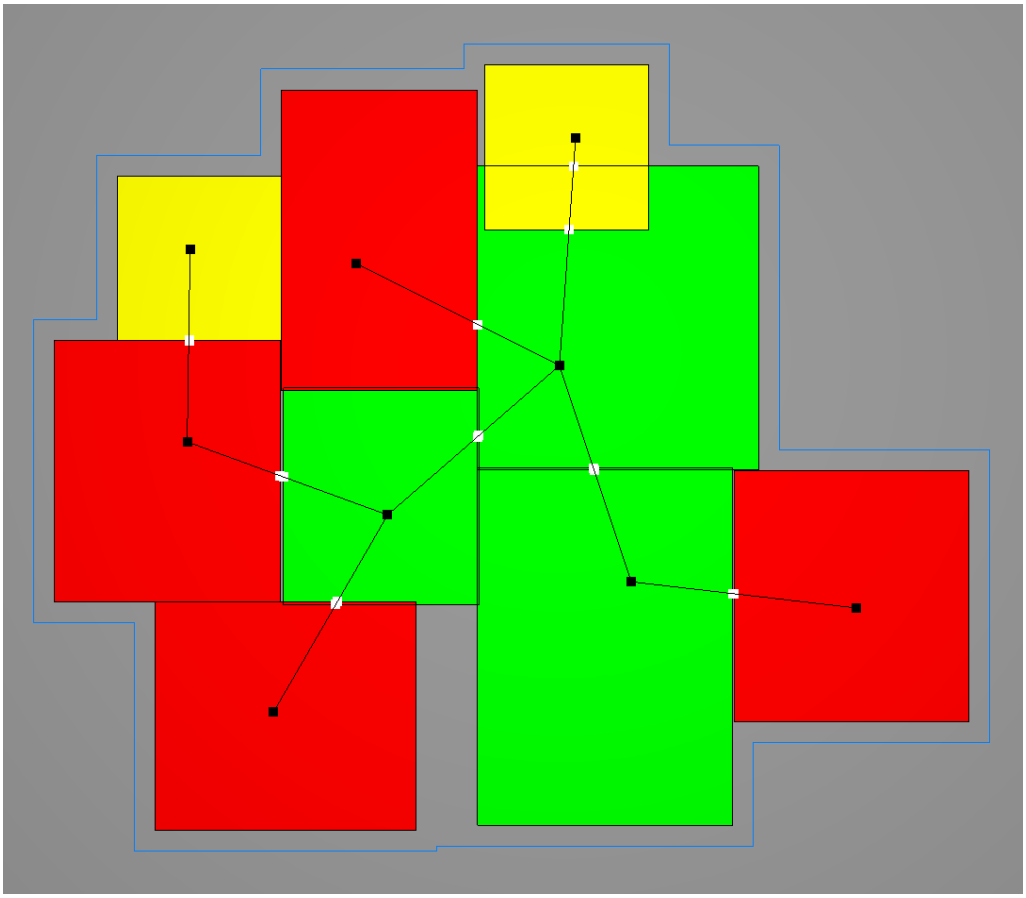
In this case, other "house-like properties" are defined as details that make sense to be in a 2D floor-plan but are not particularly relevant to the interior structure of the house. The most relevant example is windows, which are only relevant for walls that face the outside; not a lot of houses have windows that look into their toilets, after all. So the immediate problem to be solved is to find walls that face the 'outside'. To do this, we start at the bottom-most value of the floor-plan and increment upwards while finding the leftmost and rightmost values of the floor-plan. Separately, the left and right form two plots, like so:



Consider f(u) to be the line drawn by the leftmost positions, and g(u) to be the line drawn by the rightmost positions.

Now, if we connect the top and the bottom points, we have found the outline, or a collection of ‘outside walls’, on which to draw windows. From here,we found each individual wall by finding when the “function’s” value changed; i.e., every time the leftmost value changed for f(u), or the rightmost value changed for g(u). A lack of change would be a wall progressing upwards, and a sudden change indicated a wall going sideways. By piecing out the wall in this way, it is now possible to treat what was previously a portion of another wall (the room’s wall) as its own entity, so that we may now add windows to the structure.

This is the simpler part, as the only things to be done are to randomly provide the number of windows a wall should have, and then to subdivide that wall into equal lengths, adding a window position at each of those subdivisions. To save these positions, much like saving the doors in Room, we use a std::vector that holds the positions of these windows for the Room and for the outside wall structure.

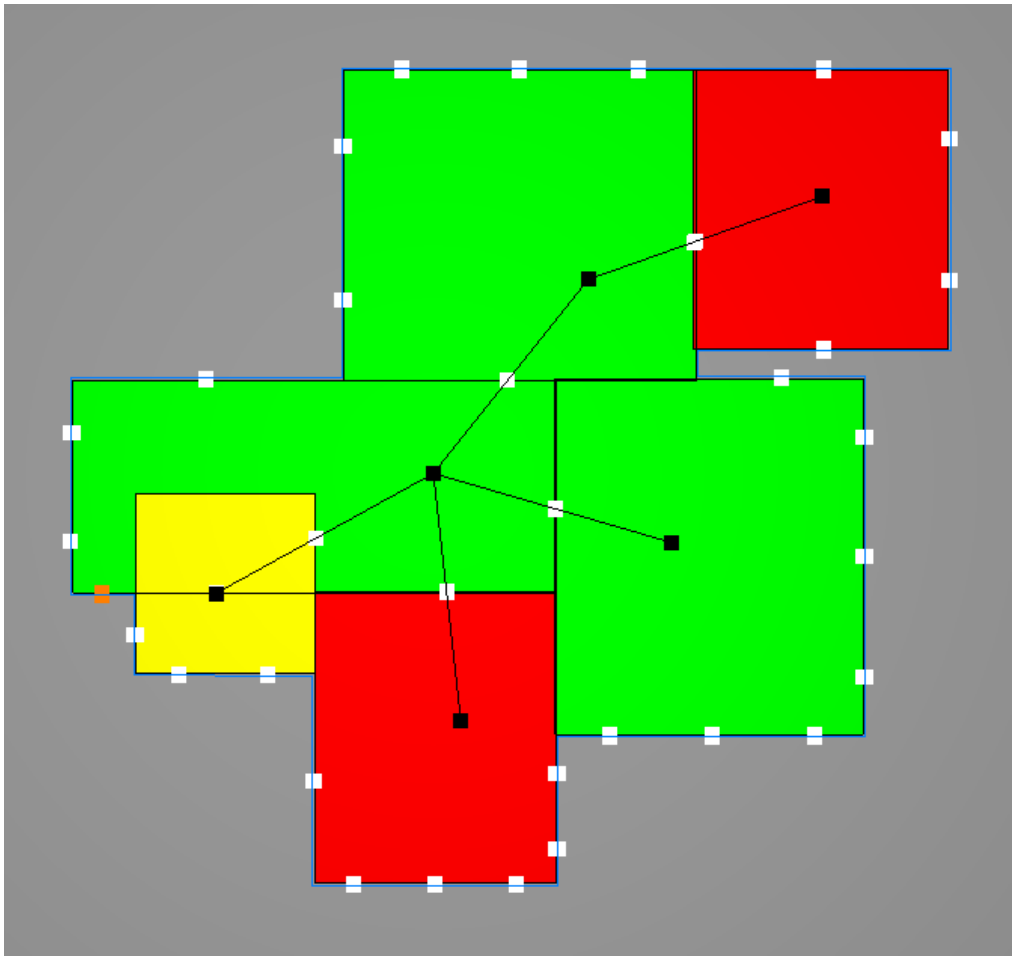


The faint blue outline describes an exaggerated display of the outside walls,
as constructed by our above method.

Finally, there is also the need for a front door. Since it would be irrational to let the front door open to a private room or an extra room, we must have the front door be connected to a public room. Of course, the most obvious public room is the root node; the one at the origin. This method starts from that node, and continues with the following:

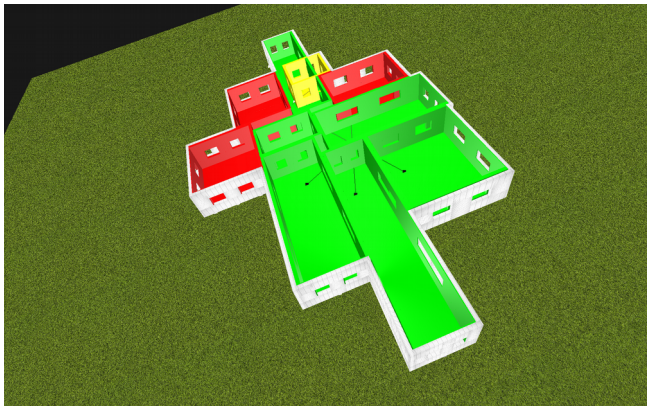
1. If there is a window in this room, find the first window and turn it into the front door.
2. Else continue to the next public room and repeat.

If no such public room is available, it is up to the user to edit the house such that a public room has the space for a front door.



The final output of our procedural 2D floor-plan generation algorithm. Orange shows the position of the front door, while white dots on the perimeter of the house show the position of the windows. Door positions are also shown in white, but are identifiable against the windows by their position. Notice that despite the public room on the right having far more windows, it is the center room (the node at (0, 0)) which gets the front door, as dictated by our algorithm.

Building a 3D Structure



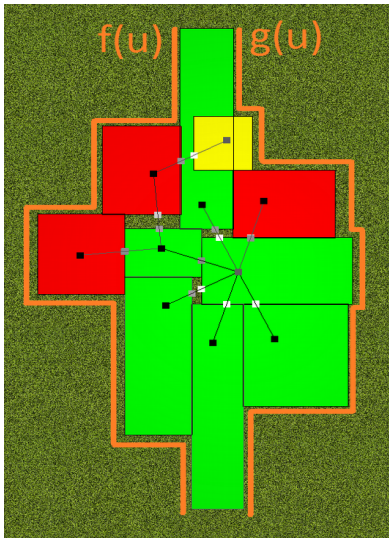
The rendered output of our basic 3D structure with subdivided doorway/window openings and the textured exterior being demonstrated.

A simple 3D wall is constructed in our program by passing a beginning and end position for a vector, which defines the direction of the wall. This is added to and offset that is perpendicular to that vector, to make the wall a 3D surface (with normals in both directions), and both vectors are added with a certain height. This box shape is then mapped to a simple triangle mesh, allowing OpenGL to render the shape simply as a collection of triangles. Said box shape is described in our Wall class, with which we render nearly all upright faces: from the rooms' walls, to the textured outer wall.

This 3D wall is extended to making openings for windows, using a very elementary subdivision method. For each wall, which is constructed from a vector to represent the x or z axis and a height to represent the y axis, we divide into thirds so that there are now 9 Wall objects which together constitute the wall. We then simply do not render the Wall object in the middle, so only 8 of those are rendered. Similarly for doors, instead of taking out just the middle, we also take out the bottom face to have the surface look more like a pathway. An optimization method would be to render the sides as an entire Wall object, so that only 4 Wall objects are made for windows, and 3 for doors. In our current program we only have the ability to show either doors or windows. It is currently set to displaying windows, which we constructed by subdividing our vector by fifths and instead take out the second and fourth subdivided face in the middle row.

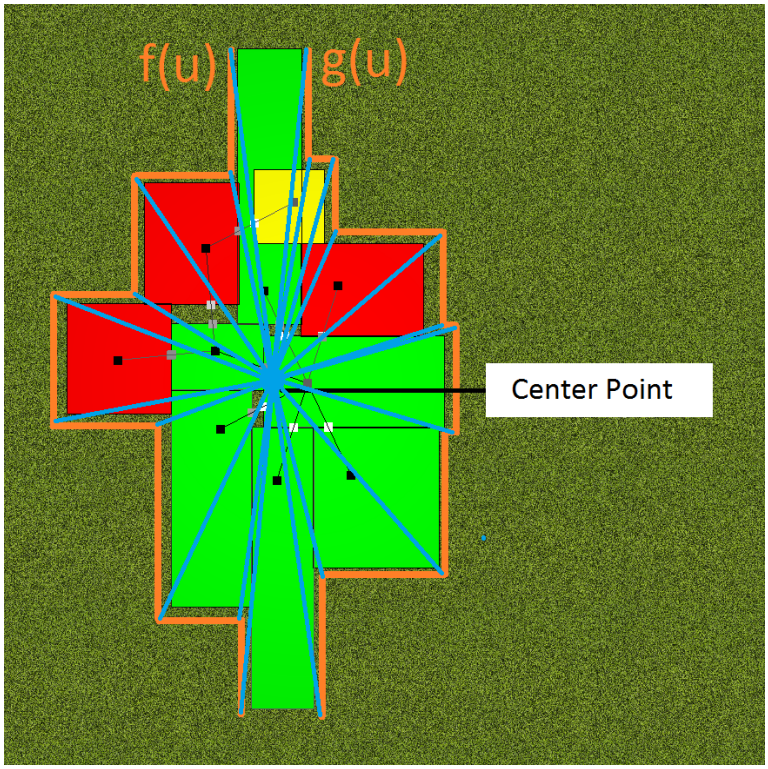
It was our intent at first to have the wall subdivide specifically at the positions specified and saved in the Room class, from the perimeter sequencing algorithm shown earlier as well as from the door positions saved as part of our 8 criteria. However, due to time constraints, we were only able to show that subdivision was a practical method for creating such shapes, and to display this have rendered every single Wall object in the house with this method just because we could.

In our procedural model, the ceiling is a layer between the roof and the actual house, so that people inside the house do not look up to see the underside of the roof directly. In the case of our project, the ceiling is the blue surface that sits on top of the rooms but below the roof, and is clearly visible to anyone inside the rooms if they looked up. To create the ceiling, our immediate instinct was to use the f(u) and g(u) functions that defined the house's perimeter to create a ruled surface across the top of the rooms. To recall the functions:



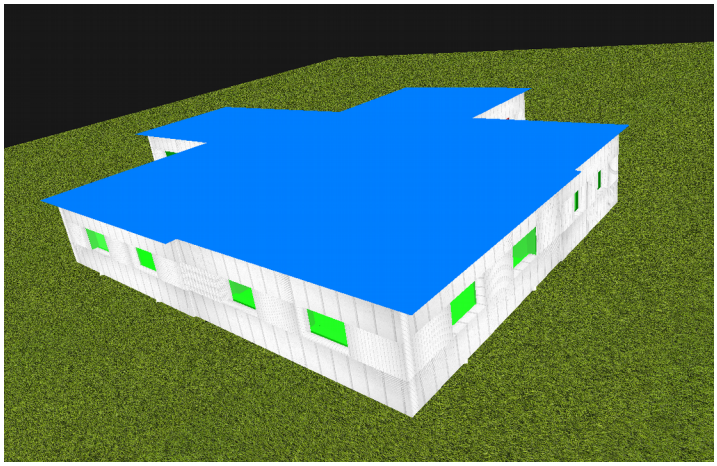
The potential lines required to draw a ruled surface which would mimic a ceiling.

We see that a ruled surface is definitely within the realm of possibility, as long as the cases where the ‘function’ jumps to the left or the right are accounted for. We attempted to pursue this, but found that this method consumed too many resources and started lagging our program a small amount. We decided to increase the increment value of u so that fewer points would be processed, but this ruined the effect of the ruled surface. In the end, we decided to sacrifice aesthetics at this juncture for the sake of computational power, and instead gave OpenGL the vertices of the ‘corners’ of the perimeter to render in a Triangle Fan format:



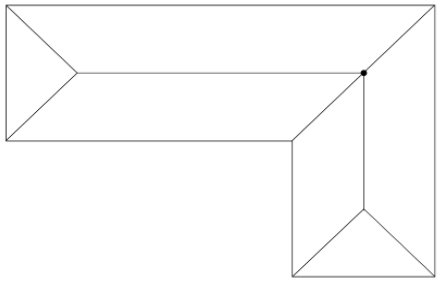
Every vertex creates a triangle with a central point and the next point being rendered, thus ‘fanning’ the ceiling out.

As those familiar with Triangle Fan will know, a central point to ‘fan out’ from must be given before any of the points being fanned. In this case, the original choice was to pick (0,0), with the argument that it would be at the closest to the largest number of places in the perimeter. However, it would make more sense for the center of a building to be the room which has the largest number of connections with other rooms. Of course, this could be the root, but it might not be. Either way, our central point was shifted to instead be the room with the greatest number of neighbours, and that proved to reduce the number of times triangles cut outside of the perimeter of the house to draw a triangle with far-away vertices.



As seen here, the ceiling has coverage over the entire area that the rooms cover, while minimizing the number of times the perimeter's corners are cut while drawing triangles. It still happens, just not as frequently as if the central point were (0, 0).

Finally, no house is complete without a roof. In order to implement this model, we pilfered through what sources we had but could not find one specifically about roofs. However, Rinde and Dahl's paper included a hallway construction method that so remarkably resembled a roof that we felt we had to try to build our roof using this method.

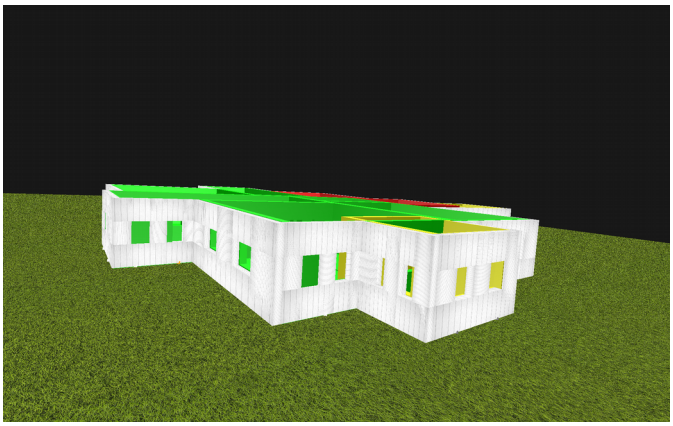


Rinde and Dahl suggested an indoor procedural model that resembled the above figure. Figure credits to Rinde and Dahl, 2008

We satisfied our aim, but not to the degree that we wanted. Much to our chagrin Rinde and Dahl are as coy about details as Martin in his paper, and so we had to intuit most of the idea behind the algorithm through the figures attached. While the algorithm asks for an iterative approach that ultimately finds the "root" of the roof, we ended our algorithm will only the first iteration to try and save on processing power.

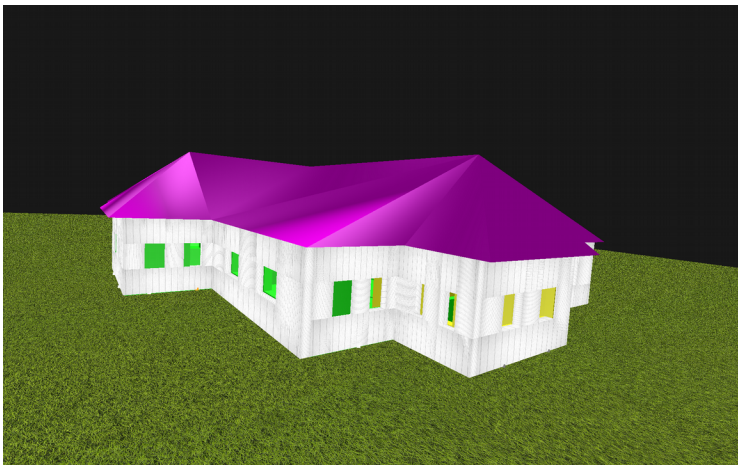
The algorithm in question creates "skeleton edges" from every corner of the space that is being dissected, so to speak. The skeleton edges extend at an angle that is equidistant to both of its corresponding walls. It intersects with any skeleton edge such that the other edge shares one wall with this one. The point of intersection is saved. In our algorithm, from there it starts drawing triangles to all edges in the ceiling if the distance from the midpoint of the edge to the intersection is the shortest one against all other intersections for that edge. From there, we try the rendering with Triangle Fan again, except this time the central point is actually just (0,0). This is serviceable, since all the intersections and this new central point have a height boost over all of the points in the ceiling. The purpose of the

Triangle Fan is simply to fill in gaps left by the drawing of the intersections, although it is not always the case that there will be gaps left in the first place.



Without a roof

With a roof. Even without a texture on the roof, by its presence the house is already beginning to look habitable and realistic.



Discussion & Future Goals

Our project was the exploration and development of introducing interactivity with procedurally generated indoor spaces. I think we have obtained a much richer and deeper understanding of the implications of allowing the user to edit, toy, and get invested with what is, in the principle of procedural generation, considered disposable and low-cost. We established the 8 criteria of what the user should be allowed to do within the context of our project, and decided that doing any more would defeat the purpose of procedural generation. As for our goal of developing a system to put them to use, we believe that our program gets very close to achieving this goal by satisfying the most important criteria, and by developing a framework where the rest are well within reach. In terms of generating a well-modeled procedural environment, I would say that we have also progressed incredibly far, by modifying both Jess Martin and Dahl & Rinde's algorithms into much more understandable and cost-efficient alternatives that produce similar results. Our ability to edit the indoor spaces in our floor-plan puts us a step above the efforts of both of these papers, and our progress in making said indoor spaces feel livable or habitable, such as by providing windows and doors at the very least, has the momentum to be completed in a very short period of time. Overall, we are pleased with our results, and we found it

very interesting to look outside class material for information on a field that was fascinating to jump into.

As hinted above, our future goals lie mainly in finishing and tidying up unfinished business, such as the matter of every single wall having two windows. We also intend to develop the user's ability to interface with the program by developing some form of UI; maybe, for example, which room they currently have selected. However, it is paramount that more interactivity not be added without considering the 8 criteria, and seeing if said interaction may make editing one of those a richer experience. Another straightforward, if difficult, way of advancing the project would be to expand on its procedural generation capacities, to better mimic housing structures both externally and internally. This includes furniture, an option that we had had on the table since the very start of the project but we ultimately decided allowed the user too much freedom, turning the project from procedurally generated houses to instead some kind of interior design application, which is obviously not our intention.

Finally, it seems semantically very easy to add a second floor to the set up we currently have. Every floor would simply be treated with a new graph of Rooms, with its own root. Stairs, on the other hand, may be a problem, but they may be edited and modified as door positions were modified in this project.

Resources

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

Müller, Pascal, et al. "Procedural modeling of buildings." *Acm Transactions On Graphics (Tog)*. Vol. 25. No. 3. ACM, 2006.

Parish, Yoav IH, and Pascal Müller. "Procedural modeling of cities." *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. ACM, 2001.

Rinde, L., and A. Dahl. *Procedural generation of indoor environments*. Diss. Master's Thesis, Chalmers University of Technology, 2008.