Procedurally Generating Interiors for Video Games

Aleksis Kavalieris

July 19, 2013

Geoff Wyvill and David Eyers

Dept. of Computer Science
Otago University

Abstract

Video games have traditionally been built around static, hand-crafted worlds for the player to explore. While well-designed worlds of this nature make for an incredible play experience, a huge amount of work must go into designing and building each aspect of the world. This gives an experience similar to a movie in that the player will play the game the way the level designer intended them to, so the second and third times playing it will pan out largely the same as the first. I aim to build a system to generate large parts of levels cheaply at runtime, along with a simple game showcasing this technology. As the game itself will take a lot of time to develop, I decided to work with another student (James Brown). We are each working on a technical aspect of the game as our fourth-year projects, while working together on the game itself.

1 Introduction

1.1 Why did we choose to make a video game?

For many years now, the games industry has been growing rapidly. In 2012, the total revenue for the games industry was almost as much as the revenue for the entire movie industry. Of course, this means that there is heavy competition not only for technical skills, but for interesting ideas and intellectual property. As with movies and music, there is a huge variety of game genres and styles. Big-budget development studios can spend huge amounts of money on building a beautiful, realistic, and engaging game in whatever style they wish; we do not have that freedom. The game we are building must be engaging not through flashy graphics and interesting characters, but through an original gameplay experience.

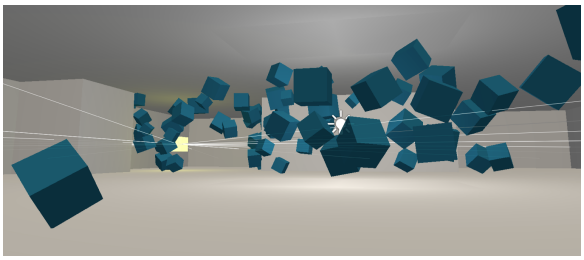

Figure 1: Cubes in their natural habitat

1.2 Gameplay concepts

In our game, the player must navigate through a maze-like building to find the exit. These interiors are procedurally generated, so each time the game is played the player has a different maze to navigate. They are pursued by an intelligent swarm of cubes, acting like a swarm of bees or the like. The swarm will prevent the player from easily navigating to the finish. We hope that this will make for a memorable play experience.

1.3 United against a common foe

Making a game is a very time-consuming process. There are many parts of the program that have nothing to do with the technical aspect I have chosen to work on, but still must be done anyway. By collaborating with James Brown, I can focus more on my procedural generation while we work together on the rest of the game. This way we can make a far more refined game than we could individually to better demonstrate what our technologies are capable of.

1.4 How we are building it

Creating games is a very well developed profession. There are many development tools available to use, which take care of vast amounts of complex engine programming to get a complicated game idea running. Writing a game like the one we have planned without the use of an existing engine would take too much time for a fourth year project, so we used one of the many powerful yet versatile and freely available game engines. We decided to use the Unity3d engine, as it gives us the freedom to make virtually any type of game using both C# and JavaScript. The majority of our game code is written in C#.

2 Procedural Generation

2.1 An exceedingly brief introduction to procedural generation

Procedural generation has been used in games, movies, and many other areas for a long time. From simple methods such as Perlin Noise and fractals to add random texture and detail to objects, to creating thousands of photorealistic trees populating the surface of Pandora[1], procedural generation is well established in movie graphics. Procedural generation can really spice up a game experience, for example creating entire worlds to explore in the incredibly popular game Minecraft[2] and the gargantuan flagship of procedural generation Dwarf Fortress[3].

2.2 Why I need procedural generation

After playing a game through even once, keen players will be able to memorize the layout of any handcrafted level. By generating the levels in this game procedurally, they can be different each time the game is played. Not only will this make it harder for the player to find the exit, but also the environment will be more interesting in subsequent rounds.

2.3 Lindenmayer Systems

The first generation method I looked at was the L-System grammars. These were originally developed to model the growth behaviour of plants, and were used to generate the trees and plants in Avatar[1]. The growth of the object is represented by a formal grammar; the current state is augmented with substitution rules. By adding a non-deterministic component to the grammar, objects can be randomly generated. This allows me to generate different layouts of rooms each time the game is run. Unfortunately, because L-Systems were designed to generate an approximation of a plant, the networks of rooms generated with my simple grammar had a 'root' room, with meandering, overlapping tendrils of rooms spreading out from that point. Obviously, this is not how buildings are built in real life. Although an L-System could be created to generate realistic building-like structures, I decided that it was not worth the trouble to devise such a grammar.

2.4 Current working implementation

I began to experiment with tile-based representations of the world. Tile-based worlds have numerous advantages, especially for real-time games like this one. Constraining the world to discrete tiles will result in a faster and cleaner generation algorithm. A tile is 4m by 4m, and can be empty, have a wall running through it, or a doorway to allow the player through. Walls can run only from north to south or from east to west, so walls are never diagonal. A wall on

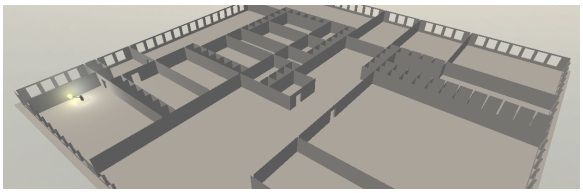

Figure 2: A map generated with the first tile-based approach

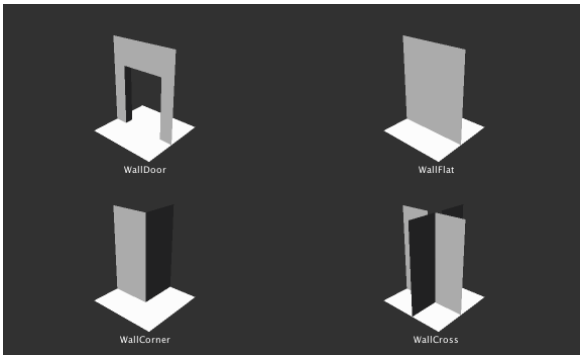

Figure 3: A sample of the different tiles used

a tile will join up to walls on adjacent tiles, and the correct geometry will be displayed.

To build a single floor of the building, which is currently all this implementation does, I first start off with a rectangle of empty tiles. Empty tiles still include a solid space to stand on, and the empty space outside the floor is not part of the array of tiles. I draw a number of overlapping rectangles of walls, creating an array of rooms. The size and position of each of these rectangles is determined randomly using Unity's build in random number generator. For each wall of each rectangle drawn, there is a chance that a door will be placed. In theory, each rectangle should have at least one door, but unfortunately the rectangles intersect to make multiple smaller rectangles with no doorways. This is the main problem with this approach, and means that the player will most likely not be able to reach every room.

Walls with windows are added around the edge of the floor to form the outside of the building. Each tile is given the correct mesh based on adjacent tiles. The meshes are then rotated to fit with the adjacent meshes, creating a complete system of rooms. These can be given correct physical behaviour to work with Unity's collision system.

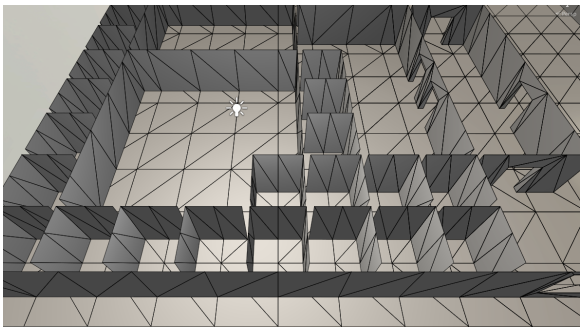

Figure 4: A multitude of tiny, wasteful rooms

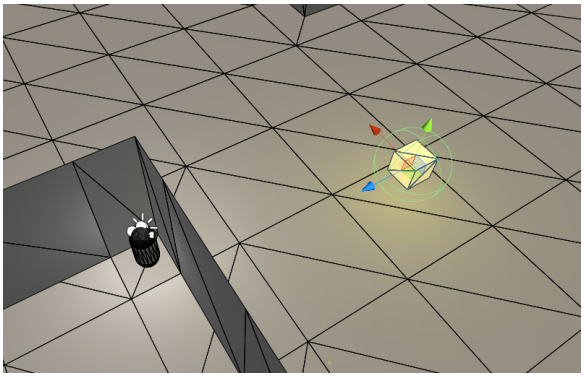

Figure 5: A basic search algorithm getting confused

2.5 Issues with this implementation

When two walls are adjacent, the walls will be melded together when the geometry is placed. If two rectangles are drawn so that there would be a narrow corridor created, instead it is filled with tiny 1 tile by 1 tile rooms with no way in or out. Not only is this a waste of space as this implementation requires a flat section of wall to build a door on, but this is not how buildings are built. It prevents narrow corridors from existing in these buildings, something that I would like to see in the game.

The walls run through the centre of the tiles, giving small areas of floor on either side of the wall that both count as being on the same tile. Normally, this would not be a problem, but James and I have decided to use a pathfinding method based on A*, which requires the world to be represented as a grid to search. Using my current method of building wall geometry, the search algorithm will have no idea which side of the wall it should be on.

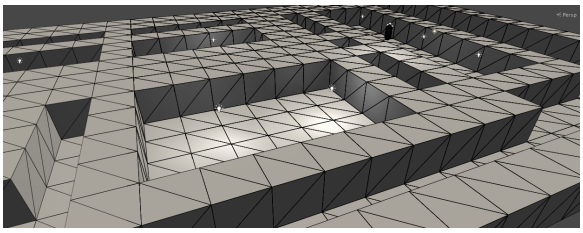

Figure 6: Simple method to represent walls

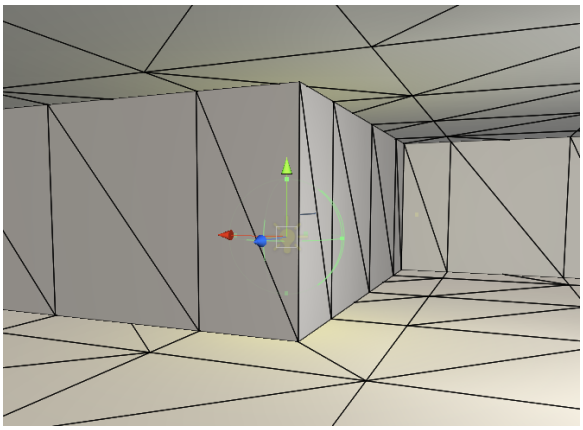

Figure 7: Yellow light from the queen cast through the walls

2.6 Make it simpler?

One alternative approach I tried out was having each tile either completely solid (A 4m by 4m block of solid wall), or completely empty (but still with a floor). This way, the search algorithm cannot get confused as to which side of the wall it should be on.

With the thin-walled approach, lights are cast through the walls. A side effect of having thick walls is that the lights casting through walls is much less noticeable. Although the free version of Unity is powerful enough for almost any use, there is a professional version called 'Unity Pro' which includes more features, such as real-time shadows. This would fix this lighting problem (at a performance cost of course), but Unity Pro costs $1500, and we are not prepared to pay that amount of money at this stage.

Unfortunately, the blocky nature of the walls is noticeable, and it makes the environment seem even more bland than it was before. There is also slightly less floor space to use as the walls take up more space.

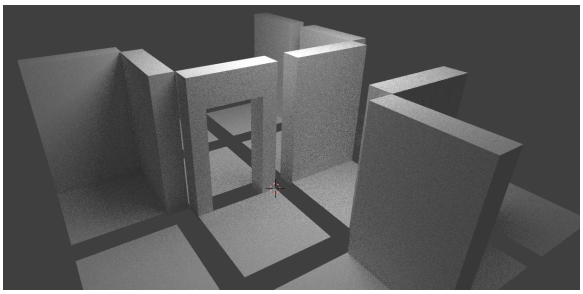

Figure 8: A mock-up of what the walls-between-tiles approach may look like

2.7 Make it more complex?

I plan on implementing a different approach to represent walls, floors, and doors. Instead of the tiles themselves representing the walls, walls will run between tiles, separating them from each other. This approach should still give a clean search space for James' search algorithm, as there will be no ambiguity over which side of the wall to be on. It will allow for narrow corridors, as the walls themselves will not merge together as in the original implementation, and will allow 1 tile x 1 tile rooms (e.g. a closet) with a door for access. It will also be more aesthetically pleasing than the thick-walled approach, as the walls will be a realistic thickness.

I also will of course implement a more intelligent generation algorithm, and at this stage I am looking at the 'constrained growth method' for procedural generation[4]. This method first determines starting points for each of the rooms to be generated; this will be a single tile for each room. One at a time, rooms are expanded to their maximum size, while still having a rectangular shape and without overlapping other rooms. From there, the rectangular rooms are expanded over the remaining spaces to create more complex shapes. This method should work well with the new thin-walled approach with walls between tiles.

3 Limitations

3.1 Limitations within Unity

Unity comes with a fast built-in navigation system, which would be the normal way to have James's swarm navigate around the map. It creates what is known as a 'navigation mesh' to show which areas of the map are accessible and give information on how to move around the environment. Navigation meshes must be created from a static level before the game is run, so this system will not work with our procedural generation. We cannot run the navigation mesh builder from within the game on our procedurally generated maps. As a result, I had to devise a method to represent walls that would be easy to write a pathfinding algorithm for. This resulted in the tile-based approach above.

When building static maps with Unity, the normal way to achieve realistic lighting is to bake a 'lightmap' of all static lights, giving realistic soft shadows, bounce lighting, and such. On top of this, lights that need to move, cast dynamic shadows, or specular highlights are added separately to complete the illusion of physically accurate lights. This baking process typically takes at the very least 10 minutes on normal hardware for an average-sized map. As we are building our levels at runtime, we cannot bake lightmaps of the level ahead of time, and, as with the built-in navigation system, we cannot run the lightmapping module while the game is running. As a result, we have to light the map entirely with dynamic lights as we can place them at runtime. Not only does this create lighting artifacts bleeding through walls, but also the lighting has to be calculated per light, per pixel, for each frame, which really slows down the framerate on large maps.

3.2 Other limitations

Representing the world as tiles is good for straight walls, but makes it difficult to build anything not constrained to a 90-degree angle. Angled walls would also be difficult for the grid-based search algorithm to navigate around. For these reasons, I have decided to not use walls at other angles.

4 Up Next

4.1 Better generation

Using the 'constrained growth method' described above, I will be able to generate much more realistic floor plans, but as with my current implementation, the rooms generated are placed randomly. There is no overall structure to the floor, and it ends up feeling like a jumble of random rooms rather than a real building. For this reason, I plan on using a central corridor or hallway as a starting point for the generation. Before any rooms are placed, a corridor will be drawn through the floor, with enough corners to not make it too easy to traverse the floor quickly. Then, on either side of the corridor, I will begin generating rooms. This should give the floor enough structure to feel more realistic.

4.2 Complete buildings, and beyond

The obvious extension to generating a single floor of a building is to generate an entire multi-floored building, complete with outer walls and a roof. This should provide an interesting and exciting level to explore. From a gameplay perspective, this could prove problematic, however. Multi-floored buildings generally have very similar (or even identical) layouts for each floor, and while it would be realistic to generate them this way, it may yield better results if the layouts are quite different.

A similar problem arises with generating stairs between floors. In most buildings in real life, there is a central staircase, which can be used to travel to

any floor. Assuming the player is only trying to get from their starting point to the goal, generating buildings with a central staircase will mean most of the floors can be completely ignored. The player will only need to explore the floor with the goal. A possible solution to this would be to require the player to find a set of objects before they reach the goal, or to offer multiple ways between floors but make the swarm aggressive enough that is can block the players advance up the stairs, forcing them to take a detour.

If an entire building complete with outer faces and a roof can be generated, then there is no reason why a whole city of such buildings of different shapes and sizes could not be generated. Extending the generation algorithm to generate such cities may be outside the scope of this project, but it is nonetheless a logical continuation.

5 Conclusion

Building maps for video games by hand is an expensive part of the development of a video game. These maps are usually built around a 'main path'; anything outside this intended path for the player is usually inaccessible. I explored some simple methods of quickly generating playable maps at runtime. This could be used to build large parts of a map, or even the entire game world, without consuming development resources. It also means the player can play the game many times and not encounter the same map.

This is of course in progress; I outlined why my current simple implementations were inadequate, and proposed a generation method that should work well to give realistic buildings for use in video games.

References

[1] Renee Dunlop. Avatar: Perfection in blue. `http://www.cgsociety.org/index.php/CGSFeatures/CGSFeatureSpecial/avatar`, 2010.

[2] Marcus Persson. Terrain generation in minecraft, part 1. `http://notch.tumblr.com/post/3746989361/terrain-generation-part-1`, 2011.

[3] John Harris. Interview: The making of dwarf fortress. `http://www.gamasutra.com/view/feature/131954/interview_the_making_of_dwarf_.php`, 2008.

[4] Ricardo Lopes, Tim Tutenel, Ruben M Smelik, Klaas Jan de Kraker, and Rafael Bidarra. A constrained growth method for procedural floor plan generation. In *Proceedings of GAME-ON 2010, the 11th International Conference on Intelligent Games and Simulation*, 2010.