MQP MBJ 1600:
AUTOMATED MAP GENERATION SYSTEM

An Major Qualifying Project Report
submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Bachelor of Science
by

Eric Faust
John Guerra

April 28, 2016

Advisors:

Brian Moriarty, IMGD
Charles Rich, CS

Abstract

The Animated Module Map Generation system is a procedural generation system for game maps that allows geographic modules to create significant variants of themselves, thereby improving the automatic production of non-repetitive game content. The system can generate modules both at runtime or offline for custom tweaking. A simple game featuring dynamically curving paths was created to demonstrate and explore the system.

Acknowledgements

We would like to thank Dillon DeSimone for his art asset creation, Thomas Farro for his work on UI development and text compartmentalization, Connor Clang for music composition, and our advisors Brian Moriarty and Charles Rich, whose input helped shape this project.

Contents

1. Introduction

The game industry needs easier ways to create content for new products. Development costs are increasing as games get more complex, with player expectations rising as hardware power increases at an exponential rate.(Moore) To respond to these challenges, many developers are turning to procedural content generation (PCG).

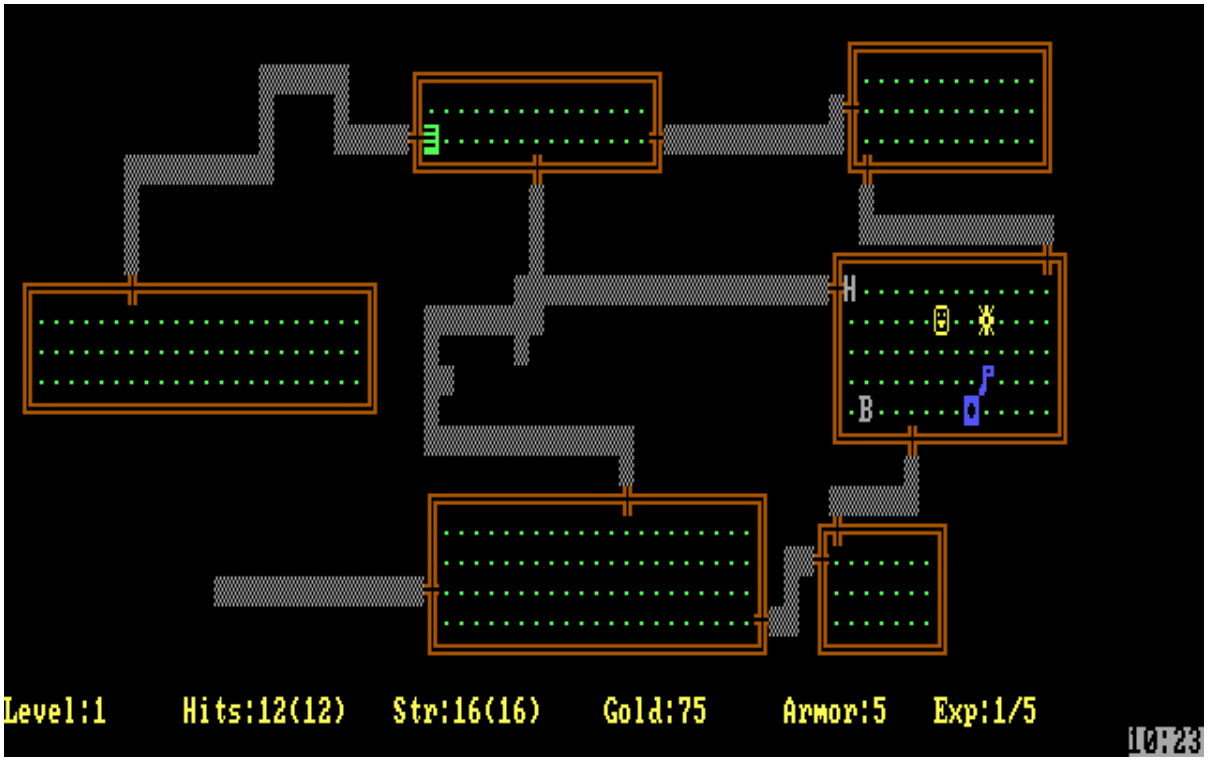
This MQP introduces a refinement to current PCG implementations. Our 'Automated Module Map Generation' system aims to avoid boring, repetitive game maps by allowing map pieces to automatically vary themselves, freeing developers to spend more time creating unique and novel pieces instead of slight modifications.

2. Procedural content generation

Algorithms have been used to automatically create game content since the earliest days of the industry, with examples found as far back as *Beneath Apple Manor* (1978).(Togelius, 172) While early PCG was done to save space — it was easier to fit a level-making algorithm on a floppy disc than many individual maps — currently it is most often used to increase a game's longevity. A player can play a procedurally generated game many times and still encounter fresh, exciting scenarios.

Some games rely on procedural generation as part of the creative process, tweaking the end result and adding content by hand afterwards, while others have their games wholly generated on the player's computer, stringing together assets with algorithms.

One of the most common uses of PCG is the creation of game maps for players to explore. This is the focus of the technology developed for our MQP.



Above: *Rogue* (1980), a classic game deeply associated with PCG.

Digital image. DOS Games Archive. Web. <http://image.dosgamesarchive.com/screenshots/rogue2.gif>. 11 Apr. 2016

2.1. Map generation techniques

There are a number of map generation techniques in wide use. Many rely on 2D grids or 3D voxels (volumetric pixels), treating maps as a structured combination of many smaller units. These methods are among the oldest.

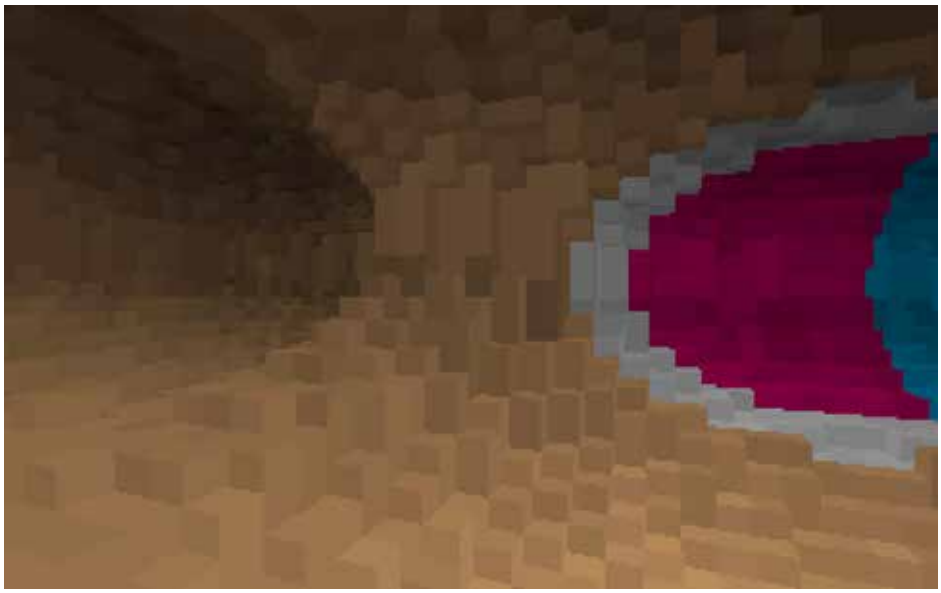
“Roguelike” games - named after the seminal 1980 title *Rogue* - are largely defined by their heavy use of PCG. Everything, from the items you find to the enemies you encounter, is distributed by algorithms rather than being placed by hand. This is not to say that such games aren’t heavily designed. Rather than specifying content explicitly, the craft lies in creating the ruleset followed by the generator. Such games are infinitely replayable, even by the original creators, as there are always new arrangements and challenges never seen before.

Above: Crypt of the Necrodancer (2015), a roguelike with grid-based level generation

Digital image. Gamespot. Web. <http://static1.gamespot.com/uploads/original/416/4161502/2853813-0001.jpg> 11 Apr. 2016

2.1.1. Grid and Voxel Based Generation

Despite their apparent richness, such systems do have limits. Most roguelikes employ grid-based geography, creating floors and walls from a pattern of repeating 2D tiles. Though different in outward form, such terrain is *functionally* locked to patterns of right-angled rooms and hallways.



Above: *Voxlap* (1994) demonstrates a voxel-based cavern.

Cave.exe. Digital image. Ken Silverman. Web. <http://advsys.net/ken/voxlap/cave.png> 11 Apr. 2016

Above: *Minecraft's* (2011) blocky aesthetic results from its voxel-based architecture.

Minecraft. Digital image. Ralph Vandenberg. Web. <http://minecraft.rvdbrg.com/wp-content/uploads/2011/09/2011-09-12_20.22.03.png>. 11 Apr. 2016

A similar problem occurs in games that use voxels (volumetric pixels) for terrain generation, including *Voxlap* (1994) (Silverman) and the more recent *Minecraft* (2011). Such building-block architectures are easily implemented, but often produce rigid geographies that do not feel organic.

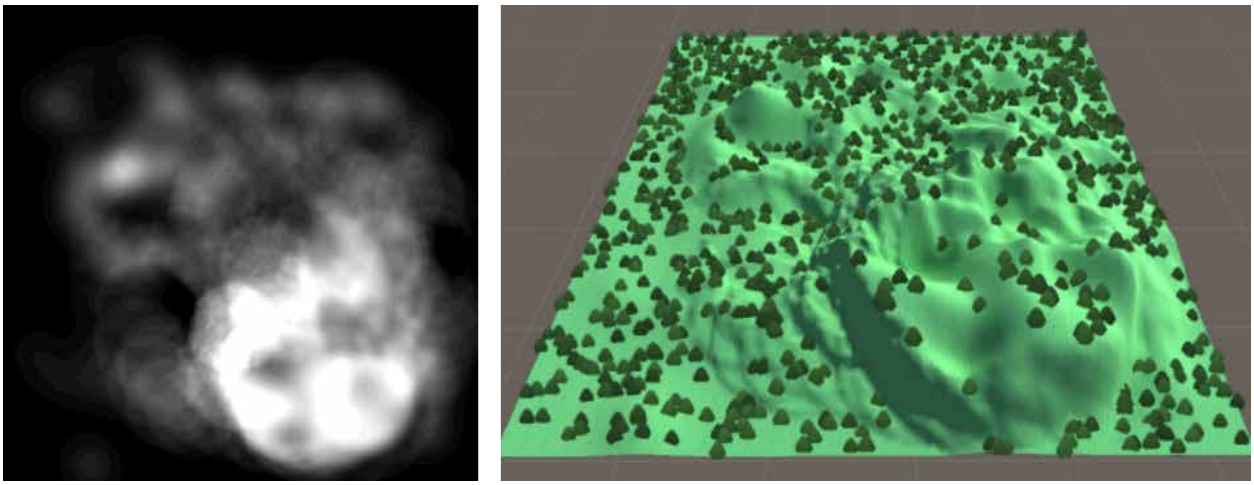
Above: An example of Daggerfall (1996)'s modular level design.

Daggerfall Module System. Digital image. Tuts+, Web. <https://cdn.tutsplus.com/gamedev/uploads/2014/01/animation-daggerfall-moduleshighlighting.gif>. 11 Apr. 2016

2.1.2. Modular Map Generation

Other PCG map systems operate on handmade modules of relatively high complexity (rooms, hallways, junctions, etc.), connected algorithmically into a seamless map. This technique goes back at least as far as *Daggerfall* (1996). Such maps can exhibit any architecture for which a module is provided, but can nevertheless begin to feel repetitive as players recognize new instances of previously encountered modules.

Other games rely heavily on topological mathematics (such as fractals) to generate their maps, especially for outdoor environments. This can produce reasonably organic-feeling terrain, but may lack the personality of hand-designed geography.



Left: The heightmap used for a piece of terrain. The lighter parts of the image correspond to higher parts of the terrain. **Right:** The terrain made from the heightmap, with procedurally placed trees.

2.1.3. Terrain heightmap generation

One last form of automated generation is through terrain heightmaps. Using a grayscale bitmap in which brightness is mapped to altitude, a 2D plane can be vertically deformed to create a detailed 3-dimensional surface. Other bitmaps can be used to "paint" the surface with various terrain textures (mud, paved roads), while other algorithms populate the plane with details such as trees and bushes. Large amounts of content can be created in very little time with such techniques.

However, there are drawbacks to this system. First, terrain can't fold over on itself, meaning it can't be used to create overhangs, caves, or anything of the sort. At most, it can create extremely steep cliffs, which can, depending on the game's mechanics, be unintentionally scaled to let the player climb to places they shouldn't be able to access. It's also limited to a very specific type of world creation. A gigantic seamless landscape doesn't lend itself well to a platformer.

Above: *Oblivion* (2006) used procedural generation to make its 22 sq. mile landscape.

Oblivion. Digital image. Keys.pk, Web. <http://www.keys.pk/wp-content/uploads/2015/08/elder-scrolls-oblivion-game-pic-1.jpg> 11 Apr. 2016

Most important, this terrain system doesn't actually make anything interesting for the player. It's a good place to start when creating a world — games like *Oblivion* (2006) created 22 square miles of terrain with this technique — but without further content placed over it, such terrain is not very unique or interesting. Often a developer will create many assets to procedurally place around the world. Roaming creatures may provide some combat, and hidden artifacts may be a goal to search for, but games that rely wholly on procedural terrain generation are often characterized by their wide open spaces, with assets that are modifications on things the player may have already seen, or different combinations of a set of pre-designed parts.

Above: *No Man's Sky* (2016) boasts 18 quintillion planets to explore

No Man's Sky. Digital image. Hello Games, Web. <http://no-mans-sky.com/press/no_man's_sky/images/NewEridu.png>. 11 Apr. 2016

2.2. Animated Module Map Generation System

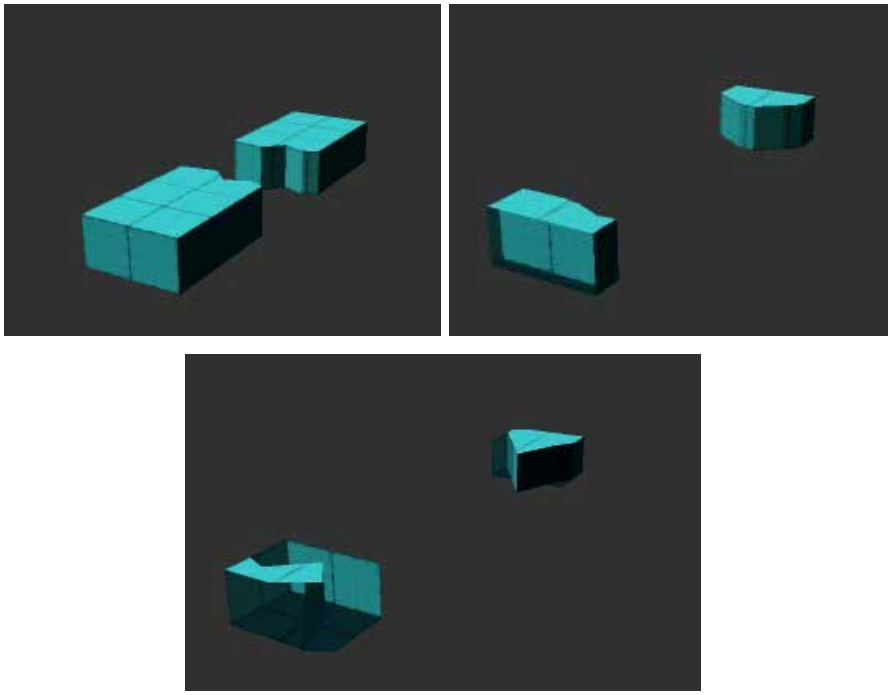
None of the existing procedural generation systems quite did what we were looking for. Terrain systems don't lend themselves to interesting gameplay scenarios; grid- and voxel-based levels lock the designer into a distinct and unavoidable style. Module-based systems have a limited number of possible module combinations.

Our solution was to make an improved version of the modular system that would vary things in each module, which we call an Animated Module Map Generation system (AMMG). This technique allows each module to appear in effectively infinite variations, significantly increasing the variety of levels the player can experience.

2.2.1. Using Morph Targets to Randomize Modules

Our first plan was to give each module a few different morph targets. Morph targets (also known as blend shapes or per-vertex animation) are a way of animating a 3D model by interpolating its vertices between a few different poses. This is commonly used in facial animation. For example, a character may have one blend shape where they smile, and another where they frown, and the game engine can interpolate between the two to change expressions smoothly. Our plan was to make a few different shapes for each room, and then interpolate between them randomly to make new variations.

There were a few snags with using morph targets. The first came up when we ran the animation randomizer multiple times. Our first implementation accidentally saved all morph target animations to the original model, which meant that we were permanently overwriting the original over and over, causing the morphs to stack up. Over time, this would lead to horribly distorted models, flipping inside out and twisting in all sorts of incorrect ways.



Above: Morph target effect being stacked multiple times on the same model.

The second issue was that, despite these models staying static and unmoving during gameplay, they were still technically "animated" models, which require about 100 times more memory than truly static models. This would be a problem if we wanted to have a large number of modules.

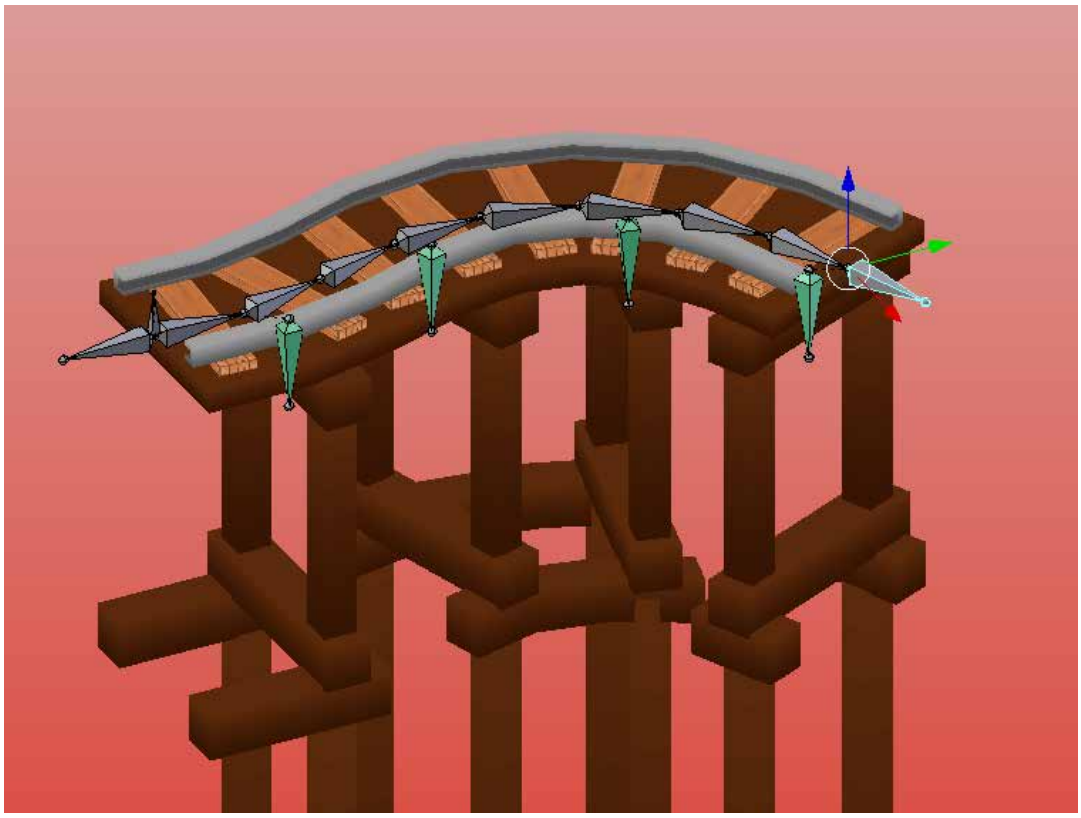
Last, and most important, was the issue of morphing entrances and exits to modules. The module system uses nodes to determine how to connect objects (which exist as points and rotations in space) to be referenced by the module-placing engine. If the entrance to a room was moved by morph targets, the node would be left behind, causing it to line up improperly. The best solution was the following lengthy process:

1. Find the vertex that is closest to the node.
2. Find the mesh's index value of that vertex, so we have a way to reference it in the code.
3. Run the morph target animations.
4. Use the index value from before to find the vertex again.
5. Move the new module to the position of the vertex. **(At this point, we have the new room in the correct position, but not the correct rotation.)**
6. Find the normal of the nearest face on the original module.
7. Rotate the new module (around the vertex's position) to match the rotation of the normal found above.

This is both a very lengthy thing to code and a lot to ask a computer to do quickly. We wanted to be placing around 100-200 modules as fast as possible on startup, and this would take far too long. At this point, we decided to look into alternative options to morph targets.

2.2.2. Using Rigging to Randomize Modules

Rigging is the process of adding a "skeleton" to a model to allow it to be animated. By rigging our map modules, we could employ traditional bone-based animation to animate them, using a bone placed at the entrance/exits of a model as the connecting node.



Above: A rigged mine cart track. Highlighted on the right is one of the bones that will be used as a node for module placement.

This system was much simpler for most cases. The biggest problem was that, when animating on multiple axes — for example, trying to turn the rail left/right and also slope it up/down — the bones would be twisted, and ended up being rotated at angles that didn't match correctly. After a lot of testing, we decided the best way to handle this issue was to just not to let it happen, and only blend between animations that move along one axis.

2.2.3. Collisions

Colliders are the part of the game object used in all physics simulations and interactions with other objects. They can be very memory-intensive, so they need to optimized. A game object can have an extremely high-detail mesh for rendering purposes — thousands, perhaps tens of thousands of polygons — but for rapidly detecting collisions with other objects, the polygon count must be orders of magnitude smaller. When possible, colliders shouldn't use a unique mesh at all, but rather be made up of a few primitives like cubes or cylinders.

Unfortunately, rigging the terrain meant that our colliders would be relatively complex. There are methods for creating simplified collision meshes when importing a model, but we needed our colliders to be calculated at runtime, after they'd been procedurally posed. Using a full-resolution mesh wasn't an option, as each would require thousands of polygons, and might be placed hundreds of times.

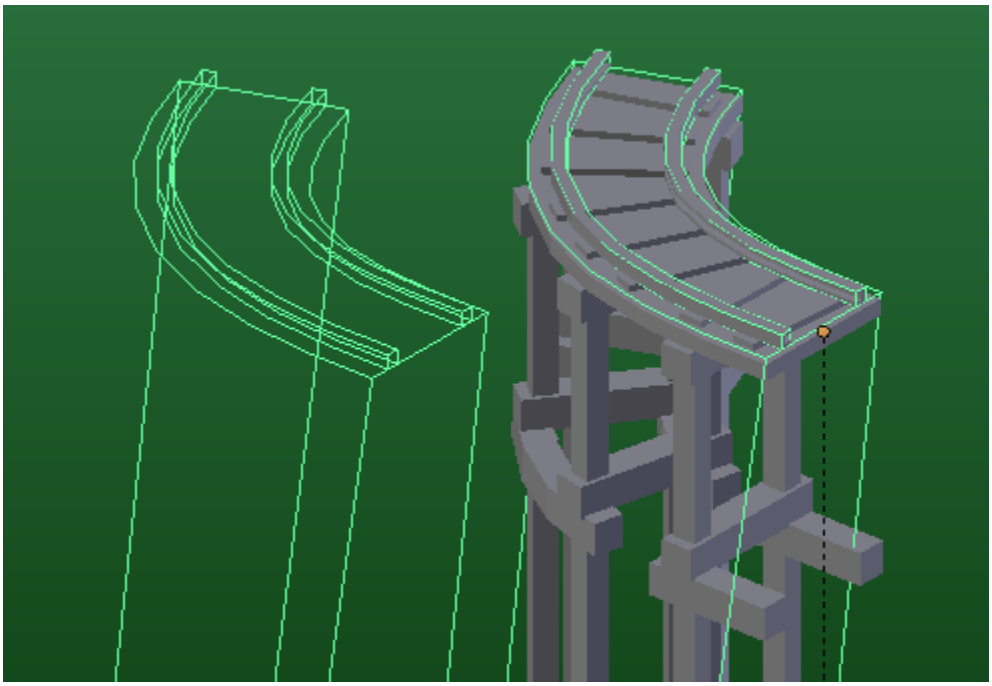
Above: A collider based on the original mesh no longer aligns after animations are baked in.

One option was to put a primitive collider on each bone. This would create a fairly close estimation of the mesh's bounds. We briefly considered implementing this, but decided against it for a number of reasons. First, there would still be inaccuracies. In the picture below, there are small gaps between the colliders at each corner, into which objects might fall or get caught. Making them larger would have the opposite problem, making objects seem to "float" on otherwise invisible platforms. Most important, adding colliders to every bone in every module would require a large time investment.

Above: Primitive colliders bound to bones. Closer, but still imperfect.

Our eventual solution was to make a highly simplified version of each mesh to use as a collider, rig the new collider mesh, and pose it in the same position as the main. At first, this seemed like overkill, but we soon realized that it would be easy to streamline the process.

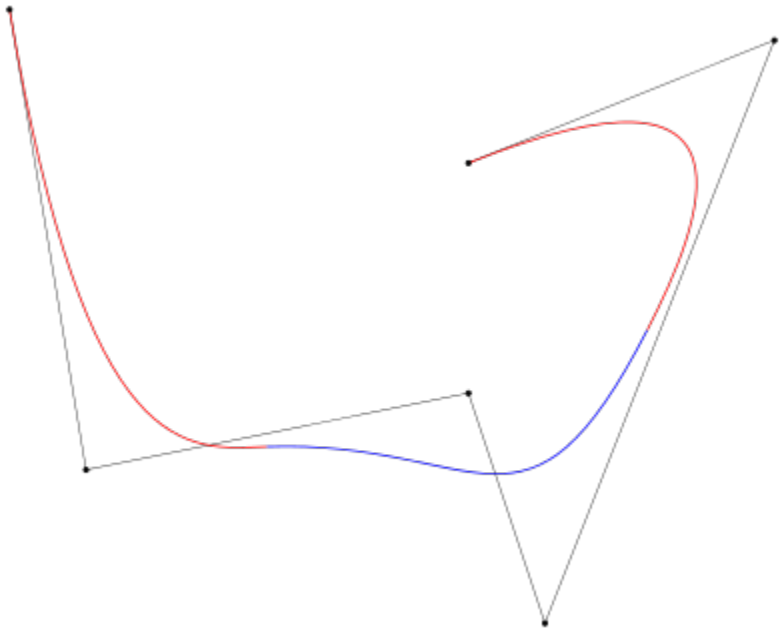
Making a simplified version of the original mesh was fairly simple. By deleting any small details, removing edge loops, and occasionally running a decimate modifier to lower the polygon count, we were able to quickly create a simpler mesh for collision purposes.



Above: An example of a simplified collision mesh.

It was important to us that we allow the system use the original mesh as the collider if it was simple enough, as making a copy would be a waste of time and memory. Our solution was to make the generation system check for a mesh named TrueCollider in any given module. If TrueCollider (our simplified collision mesh) is found, the system uses it as the module's collider. If no TrueCollider is found, the system uses the module's original mesh. This enabled us to pick and choose which modules needed optimized colliders, which greatly improved performance.

2.2.4. Benefits over Splines

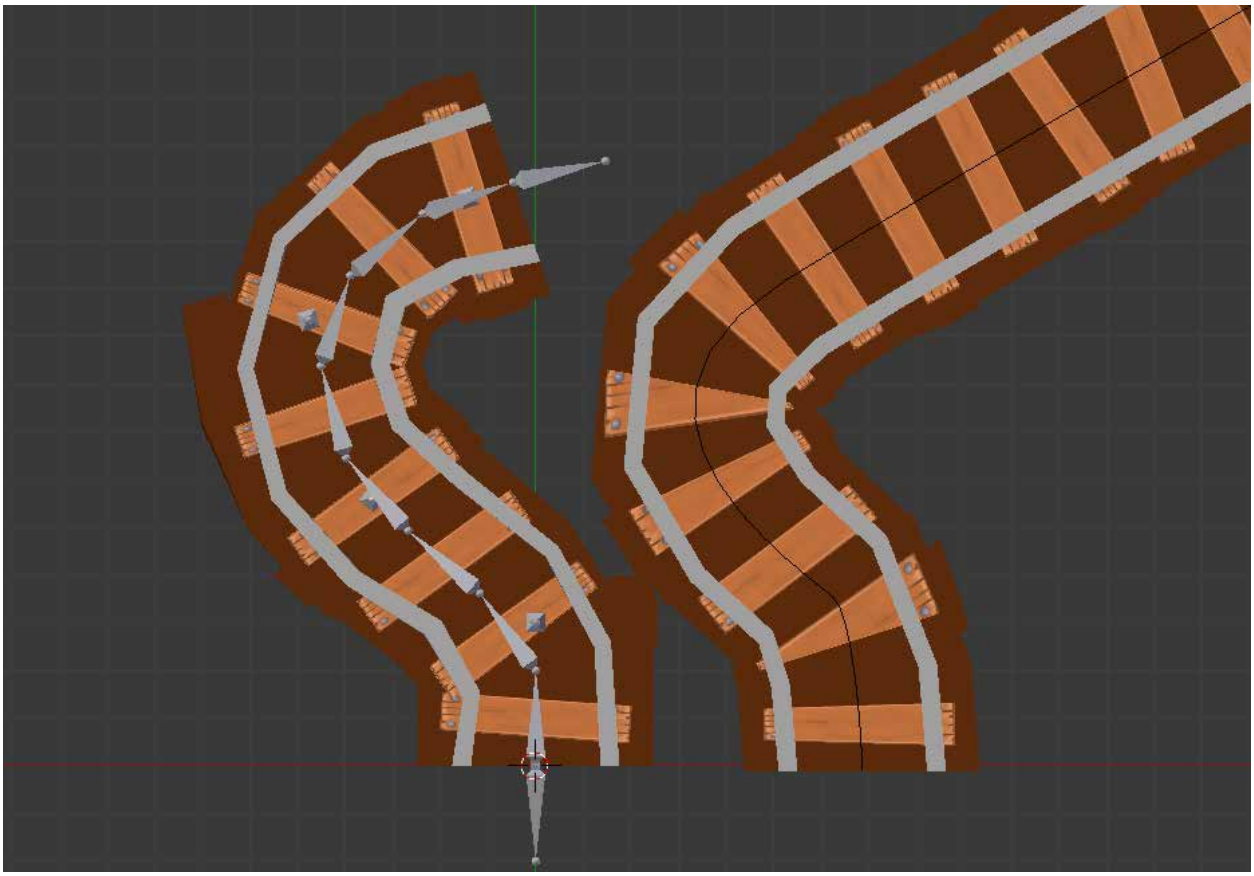


Above: An example of a spline.

Splines. Digital image. Scientific Gamer. Web. <https://upload.wikimedia.org/wikipedia/commons/thumb/1/18/B-spline_curve.svg/400px-B-spline_curve.svg.png>. 11 Apr. 2016.

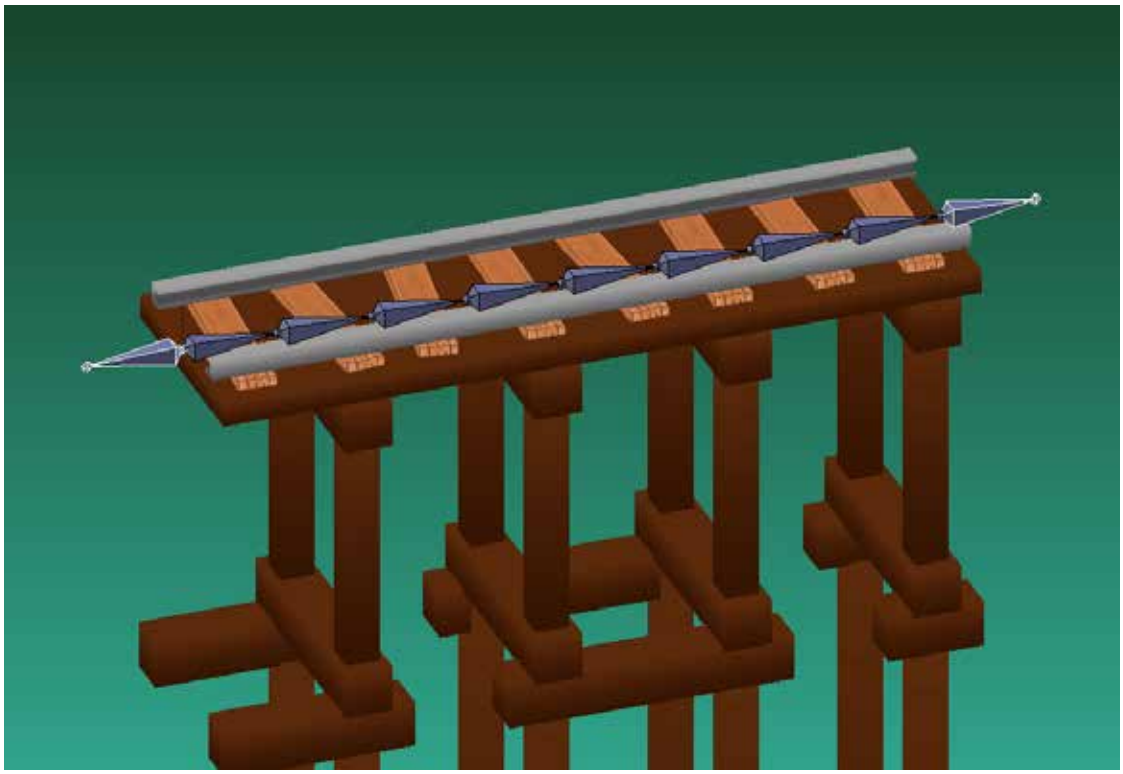
Splines are a type of mathematical function that are often used in 3D modeling to create curves. Making a model distort along a spline is an easy way to shape it- for example, defining the path of a winding road, or the way a vine curls around a pillar. We considered using splines instead of a skeleton, but decided not to for a few reasons.

The primary concern with splines was how they actually deform meshes. In the screenshot below, we show the same shape of module created through both bones and splines. Looking closely at the tighter bends, one can observe the wooden boards on the spline are bending into strange triangular wedge shapes. There's no way to avoid this, as making the model bend around the spline naturally makes sharp corners distort poorly. This is amplified more the wider the model is; a sharp turn on a wide hallway can make the geometry fold in on itself, which leads to all sorts of problems both visually and mechanically, such as colliders not working properly due to being partially inside out.



Above: Splines can distort the edges of models poorly.

Splines also can’t be easily used for more complex designs, such as rooms with more than one entrance. Multiple splines may work, but using multiple splines per module is no simpler than just using a skeleton- the biggest advantage of using splines is that just one can bend many modules, which saves time and resources. At the level of complexity we’re working with, splines don’t seem to have an advantage over rigged modules.



Above: An example of a rigged module. The node bones on each end are highlighted.

2.2.5. Implementing Modules

The final AMMG system was very complicated. The following is a list of the steps involved in creating and importing a module to a scene.

In the 3D modeling program:

1. Create a mesh for the module.
2. Create an armature for the module. Add one “node” bone facing out at each entrance/exit of the module, such as at either end of a hallway. Make sure the node bones line up exactly with the edges of the module, or the system will create seams.
3. Pose the armature in several variations and save them all as individual animations.
4. (Optional) Make a highly simplified version of the mesh, and rig it to the same armature as the module mesh. Name it TrueCollider. This will be used as the mesh for the collider.
5. Export to the game engine.

In the game engine:

6. Import the module and all its animations. Make sure each pose is saved as a different animation.
7. Create an animation controller for the module that blends between animations according to the parameters it is fed. (These are randomly assigned later.)
8. Add the module in the scene, using the master modular world generator's list of which modules it can generate from. (Use the instance of the module in the scene, rather than the original asset, to avoid the master generator pulling the module from disk every time it is generated.)
9. Add a "node" script to every node in the skeleton, and tell each node what type of node it is and what other nodes it can connect to (such as a hallway that can connect to rooms and other hallways).

At this point, the module has been successfully added to the generator. At runtime, the generator will still need to place all the modules, animate them, bake the animations to save memory, and bake the collider mesh into place, all in as little time as possible. This procedure is explained below.

3. Procedural Generation and Our System

Once we decided to create our procedural content generation system, we needed to determine how it could be used with existing techniques and concepts in content generation. The major procedurally generated content of the game is the map system and, to a lesser degree, the weapon system. Our goal was having maps that are not identifiable as a series of parts, which would keep the game feeling fresh to a player.

3.1. Common Topics

In following sections, various approaches to procedural content generation are compared. We explore both how PCG is commonly deployed in the industry, and how we might implement it using our technique.

3.1.1. Evaluation Functions

When developing a program or game with procedural content generation, it is important for the generator to be smart enough to determine when something is correct or "good." This would be particularly important for a project utilizing our AMMG system. Infinitely generated maps are of no benefit if many of the maps generated are no fun to play. This is why it is vital that a strong evaluation function is written, regardless of what approach is used.

Evaluation functions are generally run at the end of a procedural content generation algorithm. If a result isn't good enough, it simply runs again until it finds one of high enough quality. Some designs, such as evolutionary search and genetic algorithms, use evaluation functions during runtime, avoiding the need to re-run the entire algorithm. In these cases, the evaluation is vital to ensure that bad content is detected and discarded as soon as possible.

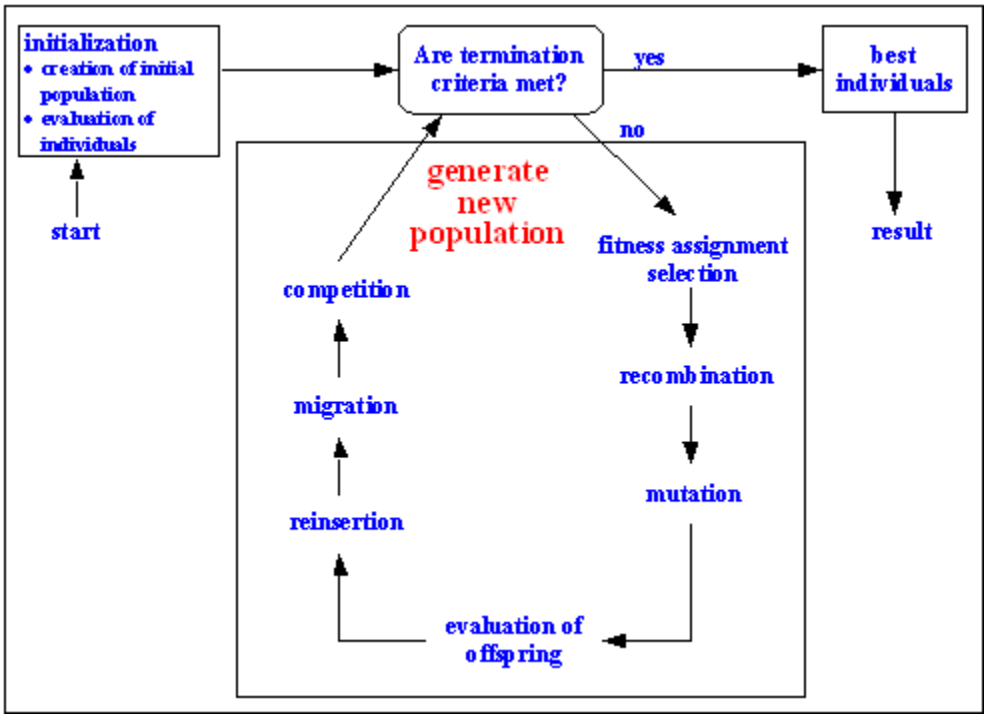
Evaluation functions are generally unique to each project. The more tailored it is to the needs of a particular game design, the faster it will detect poor content and finish generating. Our system, being similar to module-based content generation, lends itself to evaluation functions that count the number of modules, types of modules, number of connections and similar data. Unlike grid- and voxel-based generators, which can simply measure areas and volumes to estimate quality, our more flexible AMMG system poses a challenge. Each module can potentially be bent, stretched, or squashed, meaning there is no way to calculate the area or shape of a module until the content generation has completed. Accurate determinations of area require many calculations.

As an alternative, it is simpler to follow the lead of traditional terrain generation functions, and base evaluations on the *shape* of the navigable terrain. The general shape of a generated map is easily calculated by the shape of the rigged module's bones, and can be a useful indicator on how the generated terrain will play out.

3.1.2. Evolutionary Search Algorithms

One method for creating procedurally generated worlds is search-based evolutionary algorithms, a type of genetic algorithm. Evolutionary search algorithms look for a set of desired qualities in map pieces or other geometry. By employing an evolutionary approach of tweaking a piece of content's attributes and iterating multiple times, it can find the optimum means for

achieving desired qualities. This technique can improve on simple random attribute changing in cases when the chance of finding a good choice is low, such as when the search space is large.(Shaker)



Evolutionary Algorithm. Digital image. Geatbx. Web. <http://www.geatbx.com/docu/algindex-3.gif> Retrieved 11 Apr. 2016.

In practice, an evolutionary search algorithm works by generating a large set of content, either by random selection, or by using data created from previous iterations of the algorithm. Each example of content is passed to an evaluation function to determine its quality. Then the content list is sorted based on quality. Some of the lowest-ranking content is removed, and new content is generated in its place. This cycle continues numerous times, ensuring that the best content stays around, and poor content is removed. The new content that replaces the poor content can be generated randomly, improving the chance of a new approach that may work better, or it can be a modification of existing data, which makes it more likely that the new content will be of high quality, but potentially better or worse than before.

With our system, evolutionary search algorithms could be used to great effect. Each module that is placed has data that can be modified, including the amount each one is stretched or bent. There is also a large variety of different modules. Giving the content generator a way to

intelligently decide which module is placed (a corridor, room or junction) could be decided based on an evolutionary search algorithm. The evaluation function would determine quality based off of how much space the module leaves for the map to grow, the direction its exits are aiming, and/or how much space it takes up. Once a good module is identified, it can further improved by tweaking the bend and stretch to find the best value. This would ensure that later modules would be of higher quality than purely random selection might produce.

A downside of using evolutionary search algorithms in our system is the runtime investment. Because our system treats modules like characters that need time to change their animation, the evaluation algorithm is less efficient than conventional content generators. Every module being tested, together with each iteration of bending and squashing, requires an animation update, resulting in hundreds or thousands of updates over the creation of a map.

Nevertheless, an evolutionary search algorithms provides an excellent way of ensuring that high-quality content is regularly achieved. In some cases, information on what does and does not produce quality content can be stored, effectively reducing the likelihood of poor generation, and providing a stronger start for the evolutionary search. Unfortunately, such techniques are often slow to execute (Abramson) even with such benefits, potentially resulting in long load times that would annoy players.

3.1.3. Fractal Terrain Generation

Another important PCG technique is fractal terrain generation. While most of our system is designed for use with custom models hand-made by an artist, there is room for adding terrain generation in larger, more open maps. Ideally, this would enable seamless transitions between indoor and outdoor environments, allowing players to experience environments not restricted to enclosed caverns or pre-built structures.

In simple fractal terrain generation, a set of random values is generated for various points on the map, with remaining values interpolated to create smooth, believable environments. This strategy is not ideal, because it is not easily controlled to ensure optimum gameplay – it is, in essence, random. One solution to this problem is agent-based landscape creation, in which algorithms are run on generated terrain, modifying it as needed to more closely fit desired qualities.

This methods might be used solely for background and non-gameplay content, like outdoor locations, but this is not feasible with our module-based method. For example, if an outdoor Mayan ruin level was generated, the player might find themselves placed on a high-altitude location, such as the peak of a ziggurat, with nothing else to see.

Ideally we would like to merge terrain generation with our hand-crafted module system, and do so on the fly. This would require a reliable algorithm for generating content without forcing the player to wait too long. It would likely start with a database of information about an area to be generated, including how much space is to be allotted, the number of items and pickups available, how many enemies are in residence, etc. A genetic algorithm might be used to place goals, items and creatures intelligently, and design the terrain around those objects, using calculations to ensure that points of interest are accessible to each other. Creature spawn locations can similarly be guaranteed to have attack access to the player. This ensures a balance between believability and playability, offering a welcome change from confined maps created by many PCG systems.

3.1.4. Analysis of AMMG

The algorithm behind the Automated Module Map Generation system has a large number of steps. Currently the generation of a map is controlled by having a large set of modules, each one with a developer set chance of appearing. This could also be changed so that a more complex system is used to determine which modules are placed, but that is heavily dependent on the game that is being created using the system as well as the actual modules being used. Because the system is so heavily dependent on its intended use, and the evaluation functions or methods of choosing modules, this paper will explore the structure of the algorithm and general performance that can be expected.

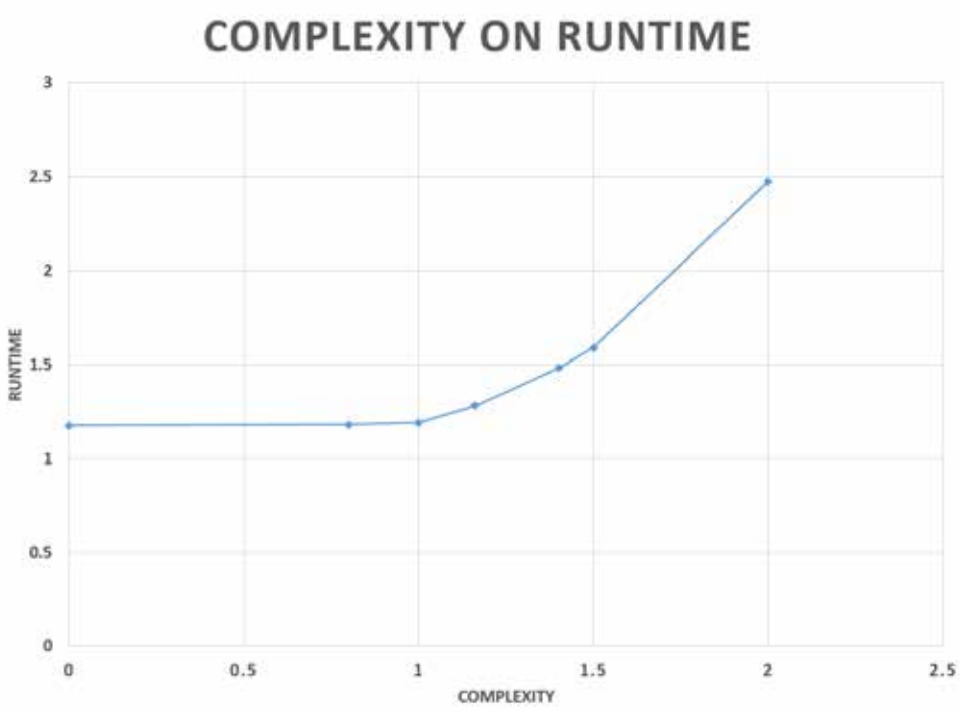
The AMMG system is set up to place a specific start module into the map, and develop outwards from that point. The total depth of the map, or how many modules can be linked off of each other from that original module, it controlled by the number of iterations set by the developer. In each iteration, the system attempts to place a new module on every open connection found in the map. It does this in a batch, allowing all modules to be baked and animated in the same frame rather than one by one. If a module cannot be placed on a connection, due to collisions or other reasons, it is left unconnected for future iterations.

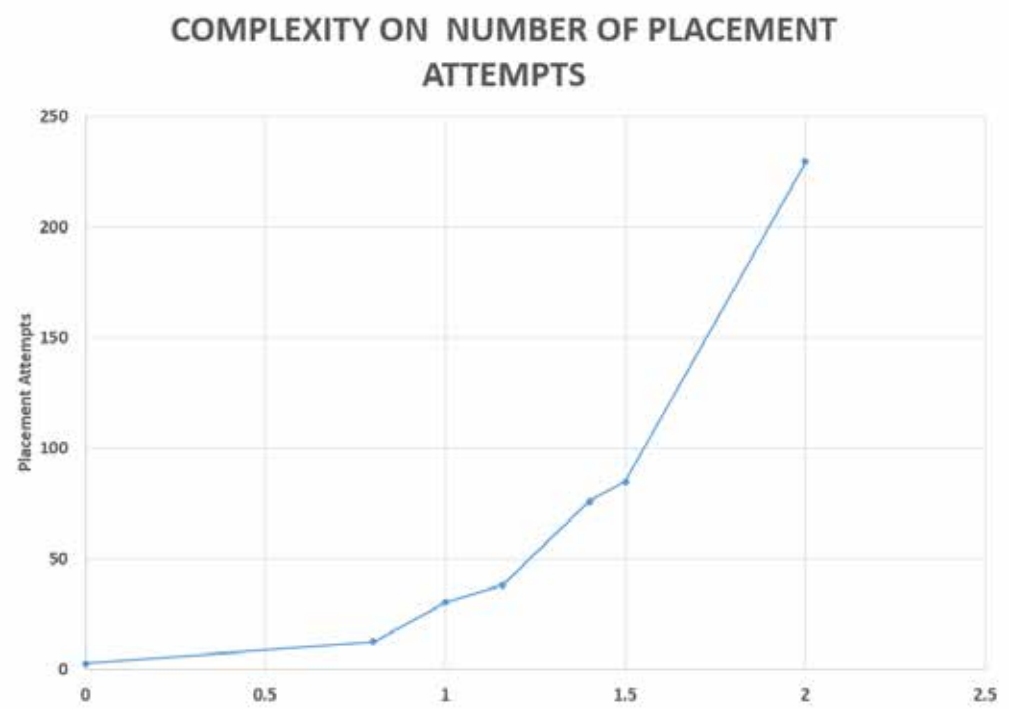
Optionally, multiple attempts can be made to place a module, potentially resulting in many more attempts but a higher average depth.

Regardless of exactly what modules or style of generation is used for a game, one thing will heavily decide how long the map generation takes - module complexity. Module complexity is a nebulous term we used to describe the likelihood of a module causing future collisions or module placement attempts during the map generation. This is most heavily decided by how many open connections a module creates, but other factors like shape can have an effect.

Several tests were made on the effect of module complexity on runtime, and the number of module placement attempts that were made to create a map. In these tests, we defined module complexity as the average number of open connections that were left after a module was placed. For example, a module cap - one that uses up a connection and has nowhere else to link future modules, has a complexity of 0. A hallway has a complexity of 1, a ‘T’ shaped room has a complexity of 2, and so on.

We ran the module generation with an iteration depth of 10, with these likelihood of each module varying with each test. The most important data collected was the average runtime, number of modules in the final map, and the number of attempts made. Each test was made over 400 maps generated, which were averaged to get their data points.





As the above graphs show, there is a definite trend in the complexity of a set of modules and the amount of work that is done to generate a map. Every module takes the same amount of time to animate, but more complex modules create much more complex maps that are more difficult to place new modules in. So, as complexity increases, the amount of time follows an exponential trend in increased processing time. This extra processing time can be limited through proper use of evaluation functions and module design, but is an important detail to pay attention to when designing maps and the games that use them.

4. Gunsketeers Workflow and Design

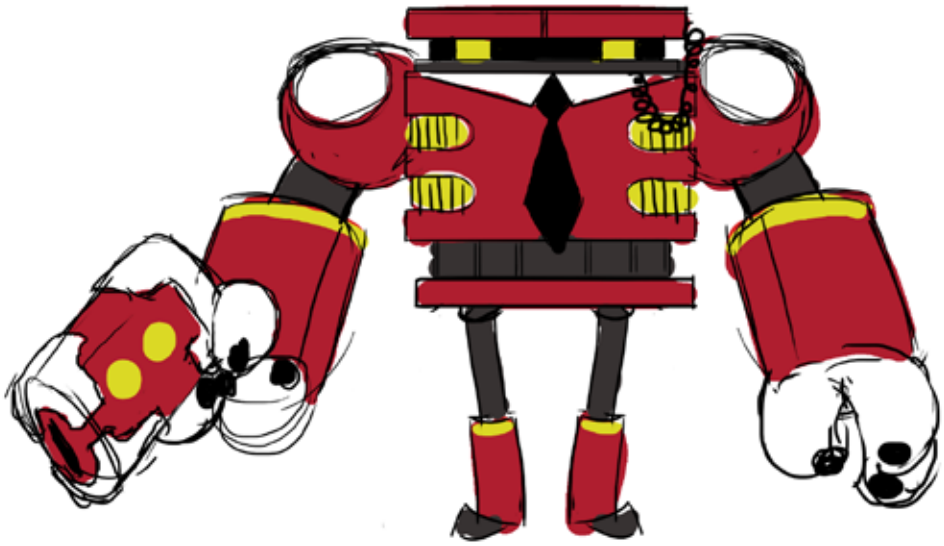
After basic implementation of our AMMG system, we began developing a game to explore the uses and limitations of the technology. Expanding on our previous focus on procedural generation, we decided on a game requiring tight control of the generation system. The gameplay would be inspired by modern roguelikes, with levels easily completed in a short session, emphasizing replayability. A procedural weapon system was created to ensure that each new weapon would be fun and unique.

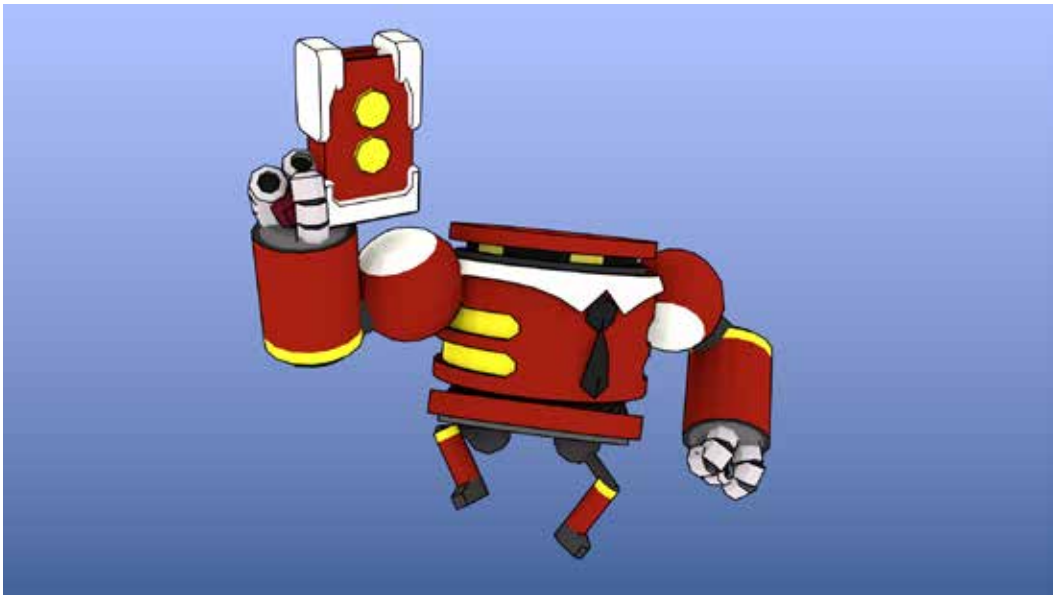
The game's scope was continuously cut back over the course of the MQP. The game as it was originally planned was always somewhat out of scope, but the biggest time loss was to the AMMG system. Once we realized how much potential the system had, we focused on it much more strongly, and the game fell behind a bit. Though we didn't meet our original goals with the game, we're still very happy with how the work we put into it turned out, both regarding the game as a standalone piece of work and especially regarding the AMMG system.

Right from the start, we wanted *Gunsketeers* to be a goofier action game. This is hinted right in the name- *Gunsketeers* is a combination of the words "gun" and "musketeers", as though musketeers don't already have the word musket (a type of gun) right in their name. We wanted it to come across as a witty, tongue-in-cheek game. That sense of humor was the biggest influence on the game's art style.

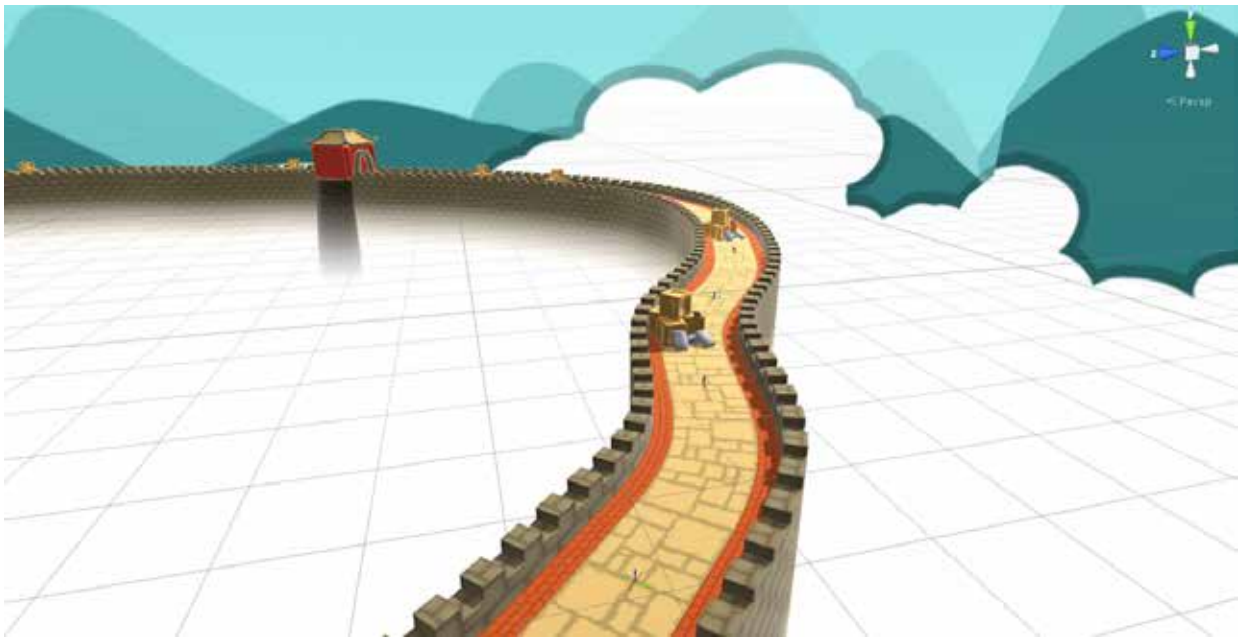
4.1. Art Style

Despite being a shooter, we wanted to avoid Gunsketeers becoming another serious game full of tan/brown color schemes and unnecessary realism. We wanted it to be fun. To avoid that, we made everything, from the enemies the player faces to the level they're running around on itself, colorful and cartoonish.

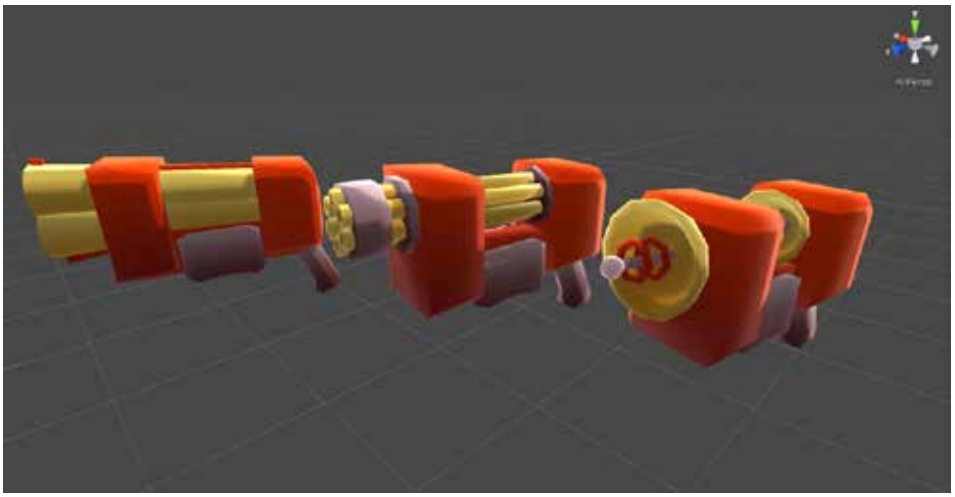




Above: Some examples of enemies, from concept art to final model.



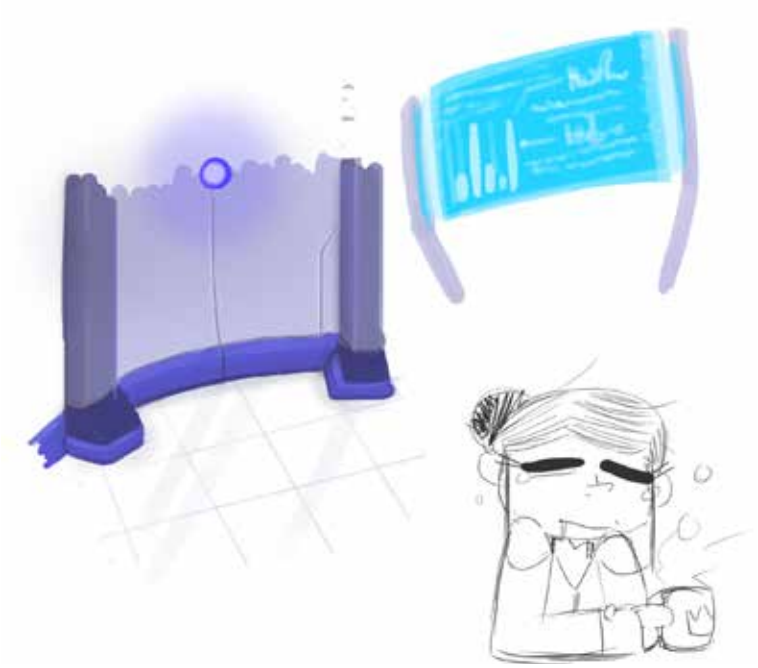
Above: Cartoonishly bright colors were important to *Gunsketeers'* level design.



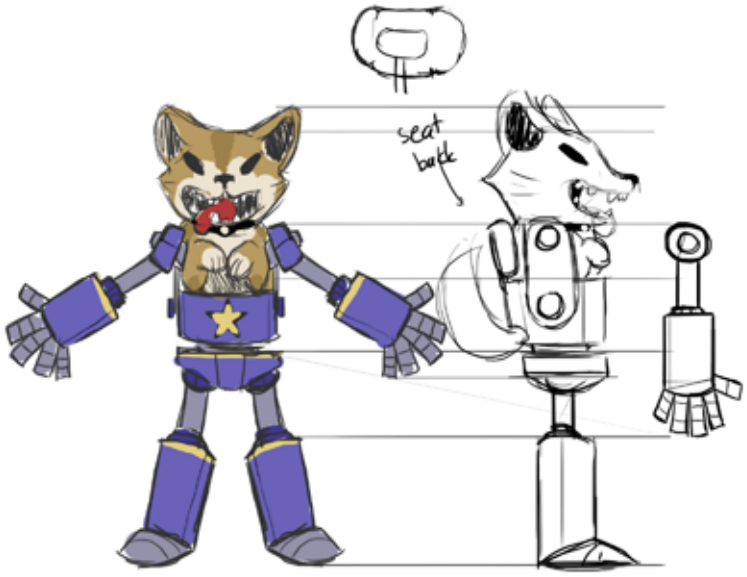
Above: Weapons and character proportions were blockier and more cartoonish to convey the lighter tone.

4.2. Writing

Conveying the sense of humor behind the game was critical to the whole atmosphere, and unfortunately, the writing was by far the weakest link in conveying that. We had fully fleshed out plans for character personalities and the overall story, but none of it ended up directly in the game.



Above: Concept art for the hub world and dispatch character.



Above: Concept art for a character who was cut for time reasons.

The dialogue wasn't intended to be cut. We took on an extra group member just to implement the dialogue system and general UI. Unfortunately, the extra member was delayed thanks to other projects, and fell so far behind that we didn't receive the dialogue system until three months after it was originally due, which at that point was too late to implement.

Above: Concept art for the gun-shaped space station the Gunsketeers were going to be stationed on.

4.3. Contracting Work

By the end of the first few months of the project, we were already behind schedule. Our PCG was capable of creating a fully-enclosed map with logical enemy placement. But the process took far longer than we had expected, and bugs were rampant. Gameplay coding and asset creation was being neglected. To speed things up, we took on two additional team members, a programmer and an artist.

The goal in adding people to a project is to obtain more effort towards completion. However, it is not as simple as twice the hands equaling twice the work, particularly because we were working with an experimental system. New members have to be introduced to systems and workflow, connected to source control, and shown what they need to do. Examples of quick and useful things to hand off to temporary help include:

- Concept art
- Sound effects and music
- Multipurpose textures, like bricks or wood, that can be reused for various purposes
- Small detail objects, like mugs and tables
- New enemies that use the same personality scripts as existing ones

Unfortunately, one of the most important things we needed was not easily handed off to temporary help: map modules. These had to be structured, edited and imported to function well with our new system, all by hand. Specific requirements were not easy to describe or enforce, or to fix when they went wrong. If new modules were needed to show off nice features, we couldn't just draw a picture of the shape and expect to get back a perfectly set-up module. With more time to work on non-critical systems coding, some of this might have been automated and made easier, but we didn't have that luxury.

The remaining examples are almost all art-side. This is no coincidence. A game can still functions with poor art, but with poor code, everything falls apart. Hiring a temporary coder to help with major systems isn't feasible, meaning that only relatively trivial things can be handed off, regardless of the coder's skill.

We did manage to obtain a few more greatly-needed assets, such as our user interface, but progress did not increase as much as we hoped. We had made mistakes in assigning features that were more critical, thinking that we should use the extra help on the most important things. Unfortunately, our new team members had to take time to familiarize themselves with the codebase and the specifics of how we needed things to be built, slowing them down. This left us unable to progress on key features until critical assets could be delivered.

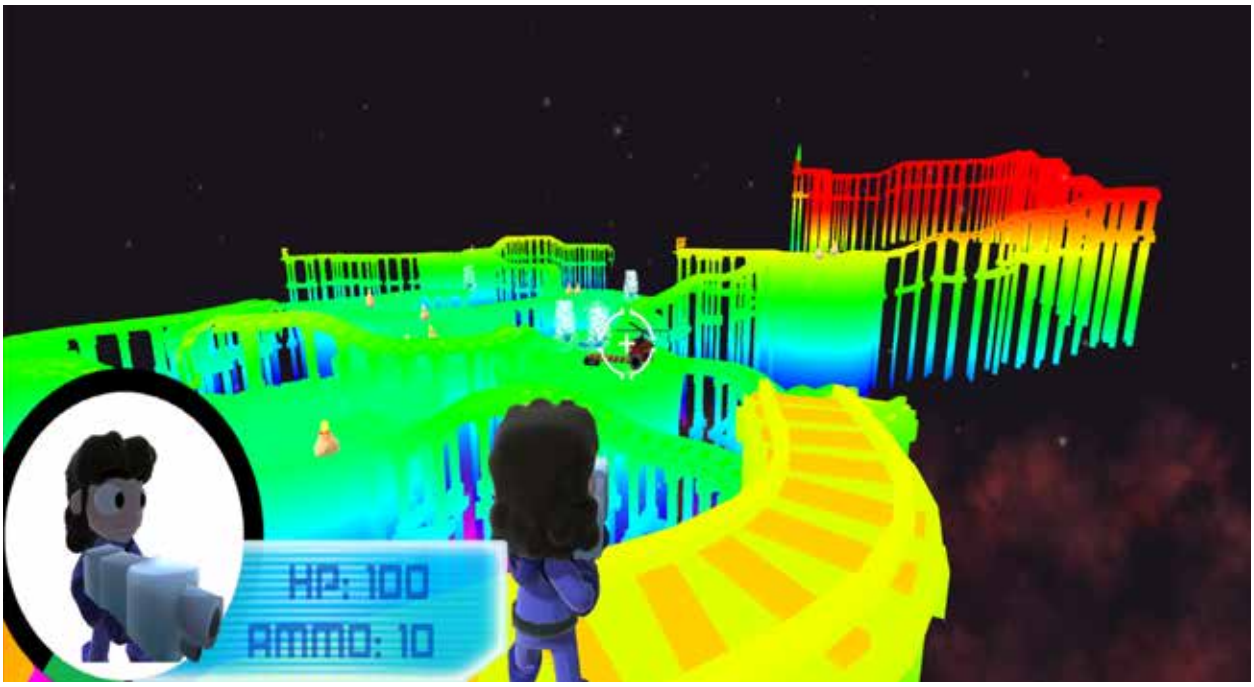
The biggest takeaway for student projects, especially those under similar time constraints, is that you should reduce the scope of your project *before* adding extra team members. If there are any parts that aren't required for the reduced-scope version, but are easily compartmentalized and will not waste the time of core members, it may be okay to hand them off to new people.

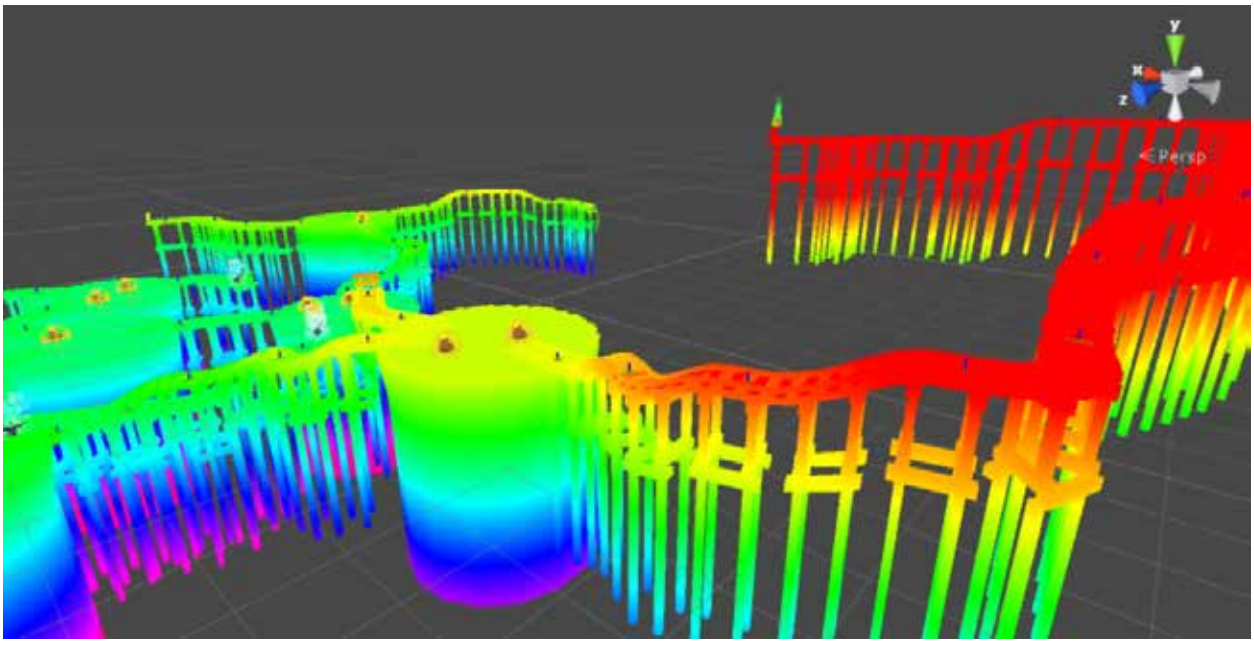
4.4. Image Classification

Another concept explored during this project was the idea of using image recognition and machine learning as an aid, or potential alternative, to simple heuristic algorithms. Because the shapes generated by our system were unpredictable, a heuristic algorithm capable of understanding and rating the complex geometry might be difficult to write. Nevertheless, it was relatively easy for a human eye to look at a generated map and judge if it was poor or well made. We decided to investigate using an image classification algorithm to assess the quality of a generated map, based solely on its visual appearance. This would allow us to cull poor maps efficiently.

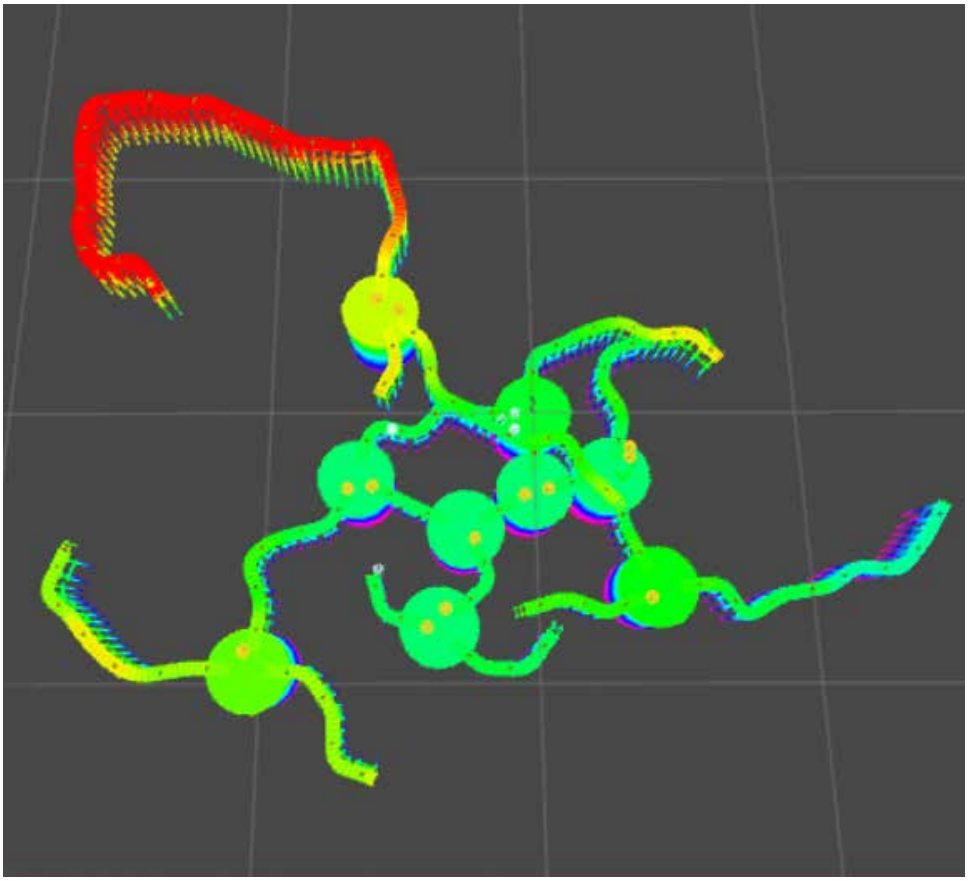
Image classification (IC) is a means of automatically recognizing patterns in a given image. They accomplish this by analyzing the image data in search of particular mathematical properties, such as the prevalence of certain colors and/or shapes. Once data has been gathered, the IC algorithm compares its findings to a library of information that has already been classified and labeled. For example, an image classifier that determines whether an image is of an animal might reference a library filled with animal photos and associated data. System operators can then use machine learning techniques to "train" the algorithm against the library to produce better predictions. In our case, we would need to teach the IC algorithm what a "fun" map looks like. Using Microsoft's InferNet library, we began working on an image classifier to accomplish this task.

An important aspect of the classification process was detecting changes in altitude. Because our system can morph terrain both vertically and horizontally, we needed to know if there were significant height differences between adjacent modules.


HP: 100
Ammo: 10


y
x
z
Persp

A simple rainbow shader was created to color modules based on their height. This allowed us to view modules from the top down, seeing the overall structure of the map without losing information about changes in altitude. The background was kept a solid color, which would allow the image classifier to distinguish between a module and empty space. From there, it was a matter of feeding the view from a camera to the modified InferNet image classifier. By collecting image data from multiple generation instances, we could teach the system over many trials to obtain a “smart” classifier.

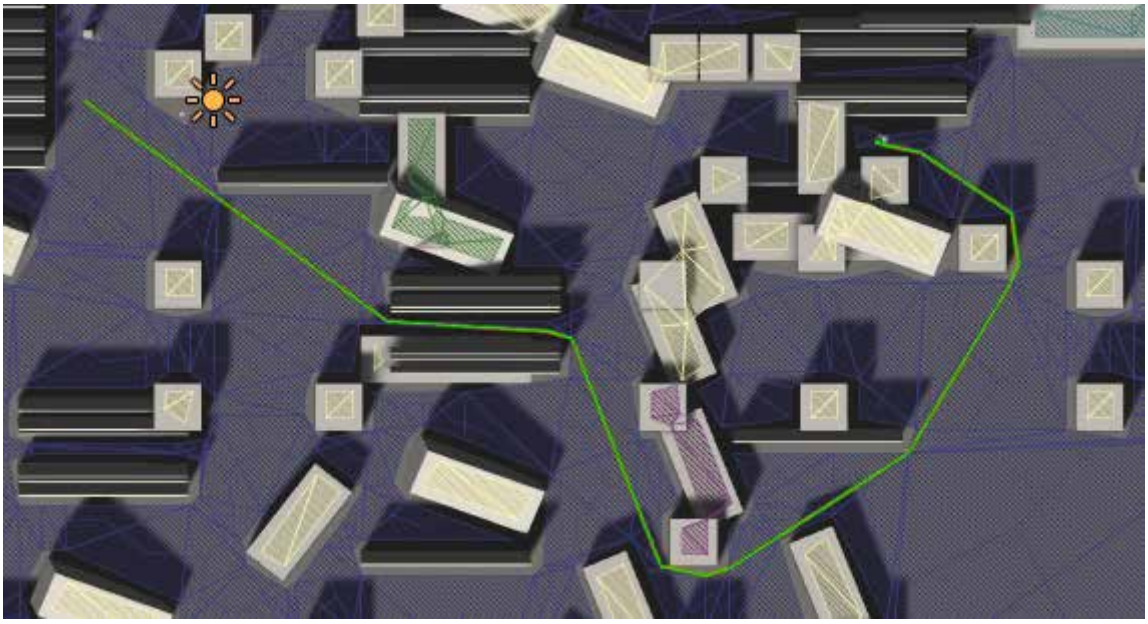


Above: The same map from the top down, as the classifier might see it.

Ultimately, we chose not to use this image classification system for its intended purpose. It had difficulty “seeing” overlaps and corner cases. On top of an already time-intensive generation system, the delay required to successfully analyze and improve each level might prove unacceptable to players.

4.5. Pathfinding

If our system was to support the majority of games — those with enemies or other moving non-player characters — it needed a pathfinding system. The Unity game engine we were using offered several options for pathfinding, but our case was complicated by the way game levels were dynamically generated. With a standard module-based system, pathfinding segments can be prebaked onto modules and fit together at runtime to allow seamless pathfinding. With our system, we cannot know the final shape of the module, and therefore cannot prebake any pathfinding.



AStar Pathfinding. Aron Granberg. Web.
<http://arongranberg.com/astar/resources/images/frontpage.png> 11 Apr. 2016

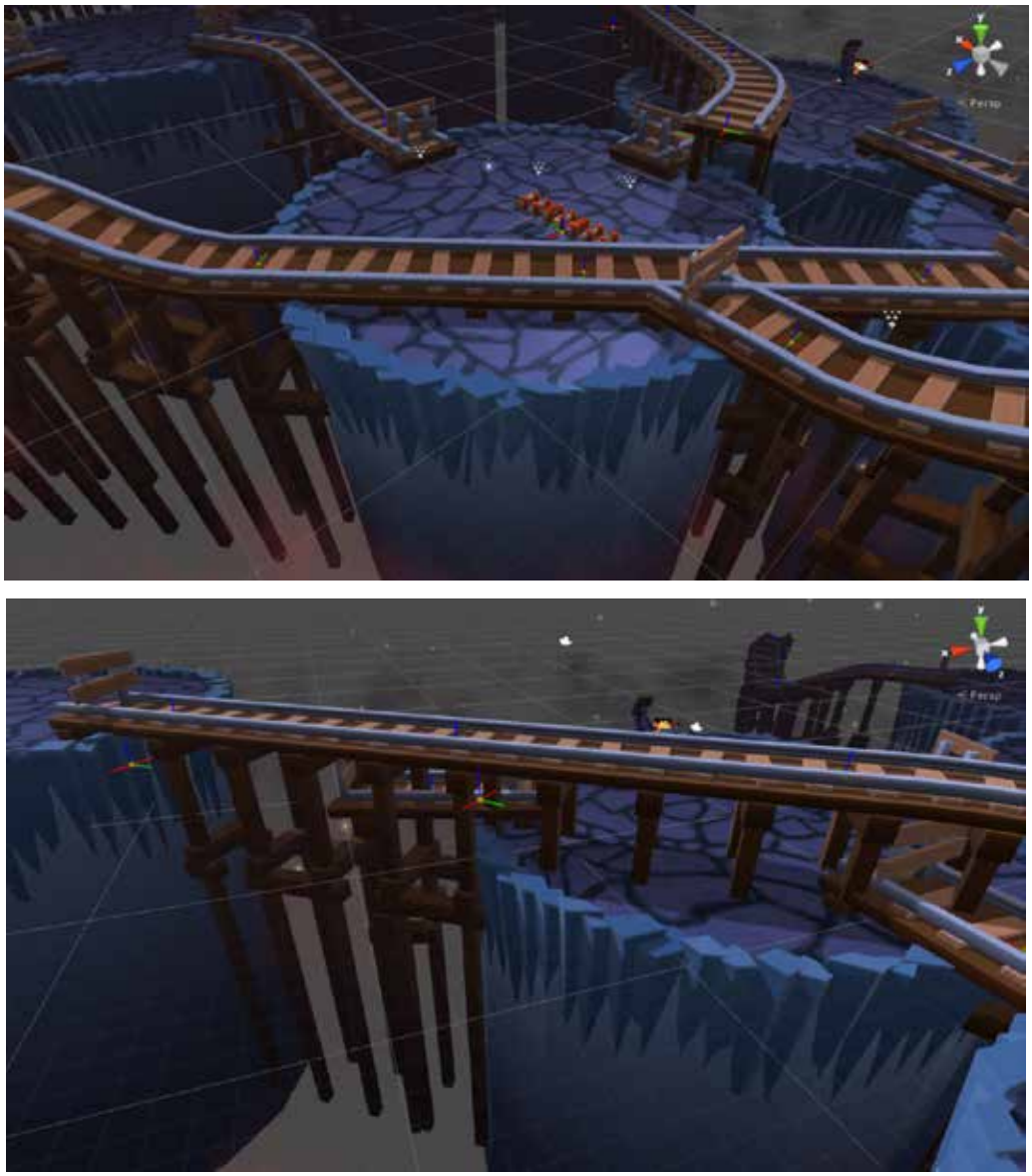
Luckily, we identified a third-party Unity library (A* Pathfinding Project by Aron Granberg) which provided the functionality we needed in the form of recast graphing, using algorithms capable of recalculating specific sub-sections of a graph. Combined with our rigged map system, it was it possible to have map segments move and morph while characters were navigating them. Even decorative objects placed after the initial module morphing could be included in the recast graphing. This ensured that we could decorate our map modules, making them less recognizable as a module the player might have seen before, without losing accuracy in enemy pathfinding AI.

Granberg's library allowed us to quickly calculate pathfinding navmeshes our maps. In initial testing, even very large maps were calculated in a fraction of a second.

4.6. Difficulty working with the system

The AMMG we created is a new technology that has no external documentation or support. Any problems we encountered had to be solved on our own.

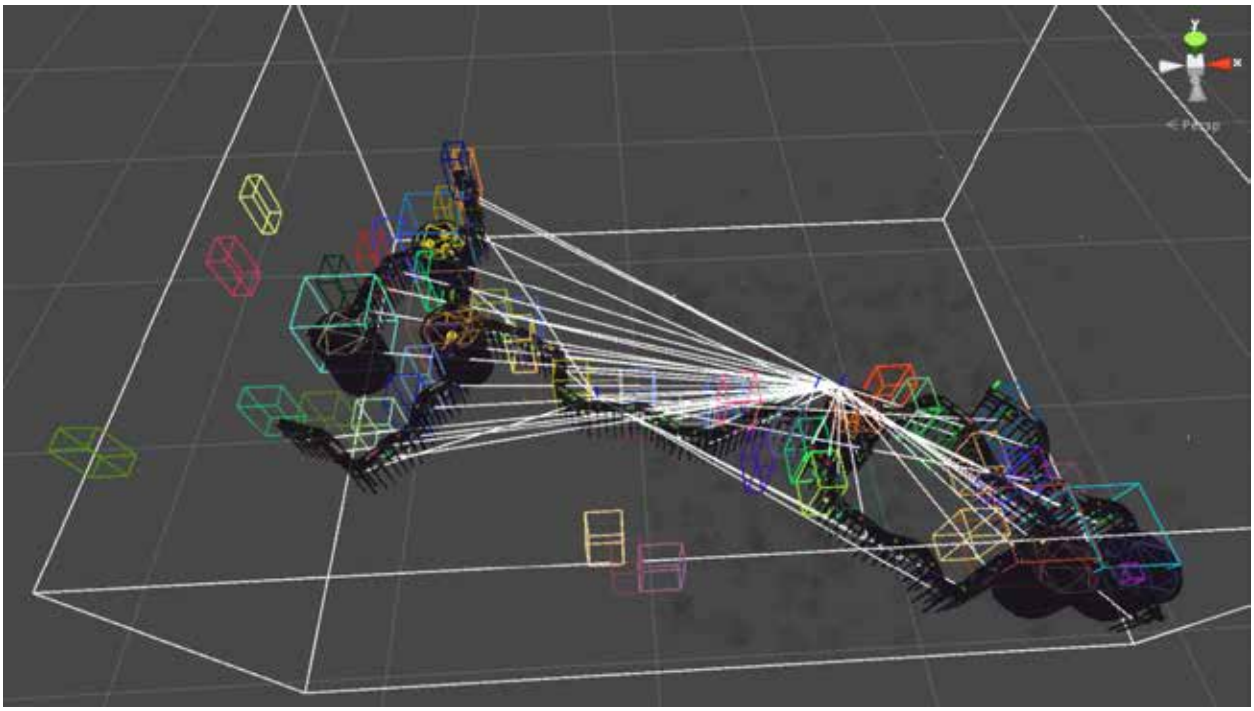
Treating map modules like rigged characters is not something Unity is designed to easily handle. Throughout the project, engine-related problems slowed our progress, especially issues related to collision checking. Early on, it was difficult to bake in the correct collision meshes. We often found ourselves dealing with meshes that were inaccurate, or misplaced very slightly. Modules would be "almost" lined up, with miniscule gaps. While this didn't cause problems with gameplay, it was quite noticeable visually, making maps look unfinished or broken. To prevent this, we had to ensure that the steps of morphing and "freezing" each map module were performed in a very specific order, with careful attention paid to the accuracy and placement of collision meshes. If a call to check the mesh was made at the wrong time, the modules might not line up, or could be completely incorrect.



Above: Railway map modules clipping through others, ignoring collision checks.

Properly animating and aligning the modules was a major struggle, and likely took up nearly 1/4th of the total time and research spent on the system. Only after the proper sequence of calls and methods was developed could modules be reliably designed and imported.

Our animated map modules are far more complex than conventional PGC mapping techniques. If any part needed is missing or slightly incorrect, the module as a whole is unlikely to work. It is also difficult to pinpoint where an error has occurred, making it vital to implement good error reporting and catching. But often we would only realize the need to error-check something once we had seen the havoc it could cause.



Above: Railway map modules clipping through others, ignoring collision checks.

For example, the above image shows an issue we faced late in the project. It shows a map after generation, with sets of railways and circular cavern platforms. A recurring issue was that modules would occasionally ignore collisions — Unity would report that two objects were not colliding, even when they clearly *were*, visually. The colored cubes are where the objects think the bounds of their colliders are, which in this case is obviously incorrect. Also, due to the way module rigging and placement is handled, all of the white lines should be pointing from where the object is visually to where the map generation began (at coordinates 0,0,0). All objects rely on this arrangement for correct placement, but some of the above objects point to different locations.

It should not even be possible for the generator to finish placing a map containing such errors, as colliders are vital to proper module placement. Strange issues like this had to be dealt with throughout the project, forcing us to be constantly fixing bugs — an expected and necessary step of engineering a complex system — but it took a heavy toll on the quality of the game we were trying to develop. The system remains promising, but developing it into an easily used standalone system would take significantly more effort and polish.

4.7. Unity's Animation System

Ideally, the AMMG system would generate every single module nearly simultaneously, right at the start. The level loads up, the system activates, and it deals with everything as fast as possible. Due to a specific quirk in Unity, though, this proved impossible.

Unity deals with animation of an object at the end of a frame, after other functions have already run. This is presumably so that an object can be spawned in, run whatever code necessary, and then begin its animation uninterrupted by the end of the frame. This guarantees the player never sees, for example, enemies spawning in their default poses. This otherwise useful feature of Unity meant there there was no way to get a module to spawn, animate to a pose, and lock into place within the same frame. We would have to spawn and animate each module on one frame, and place them in the next.

Our maps deal with at least 100 modules at a time, often going as high as 200 or 300. Waiting for one frame per module would slow level generation to a crawl. Our solution was to generate modules in batches. One layer of modules would spawn together, then be placed at all available nodes. The next layer would then be placed during the next frame. Using this system, we brought the frame count down from ~200 to ~14 frames for level generation.

5. Conclusions

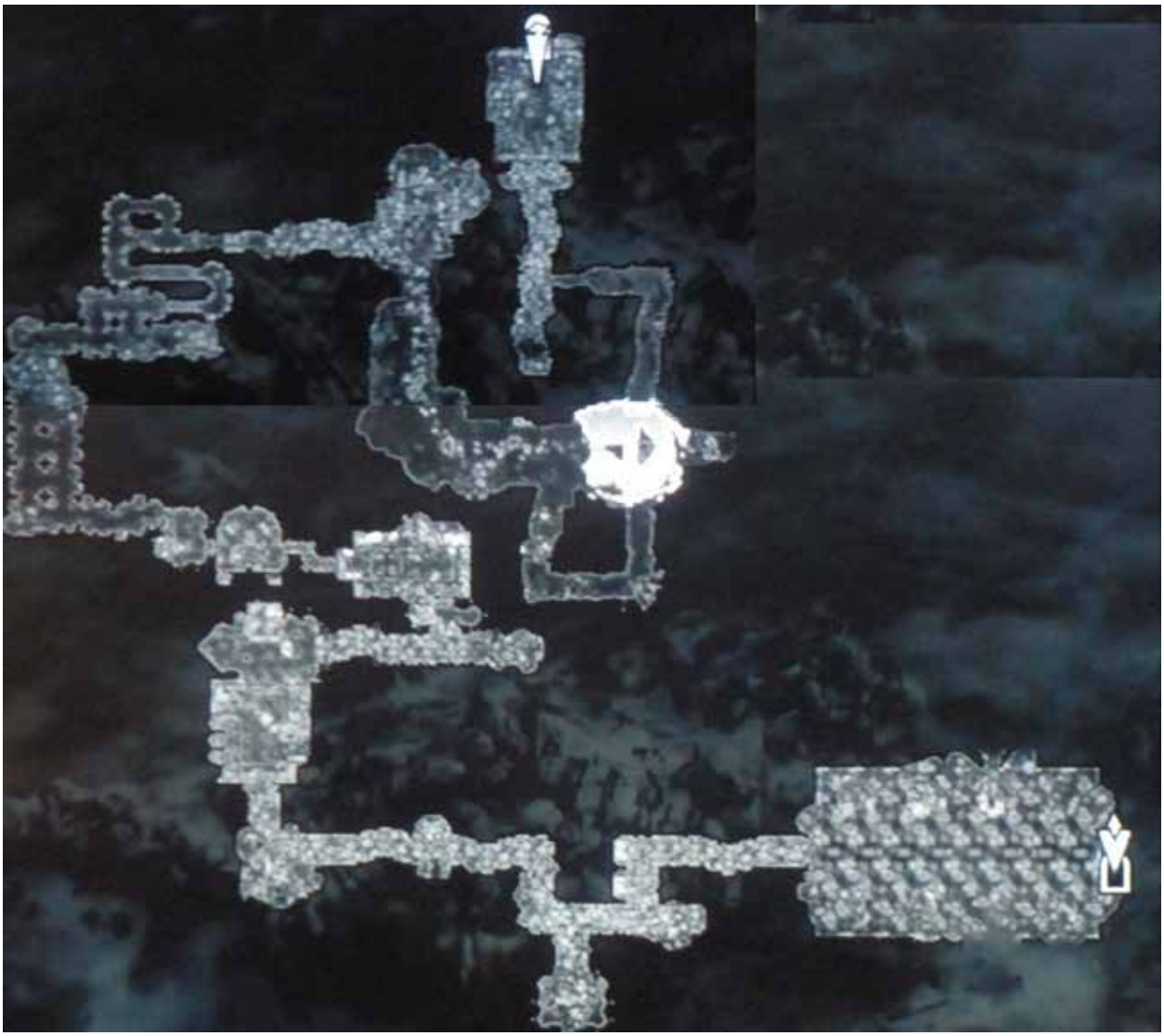
The AMMG system we developed for this MQP is powerful. The roguelike game we made barely scratches the surface of its potential. As we worked with it, we realized there were plenty of genres and art styles that would benefit significantly more from it than the game we were making. That's not to say it's without its drawbacks; aspects of the system make it slower and more unwieldy than conventional procedural generation in some ways, and since it's an original method, there is little in the way of documentation, though perhaps this report will help change that.

5.1. What Would Benefit Most

While we developed a roguelike game to demonstrate the Animated Module Map Generation system, it could actually find use a wide variety of games. Roguelikes, like the one we created, can obviously benefit due to their reliance on content that is generated at runtime. Even games that do not generate at runtime could use the system as an automated development aid. Games employing a surface-normal based gravity could use the system to twist and turn without being limited to downward-oriented gravity. Detailed environments for racing games could be created sacrificing the curving tracks of conventional maps.

5.1.1. Large Games Focused on Replayability

Games such as *Fallout 4* or *Skyrim* feature underground maps and interior structures. These dungeons are hand-created out of many map sections, but often end up with repetitive layouts unless many variations are created, using up valuable development time.



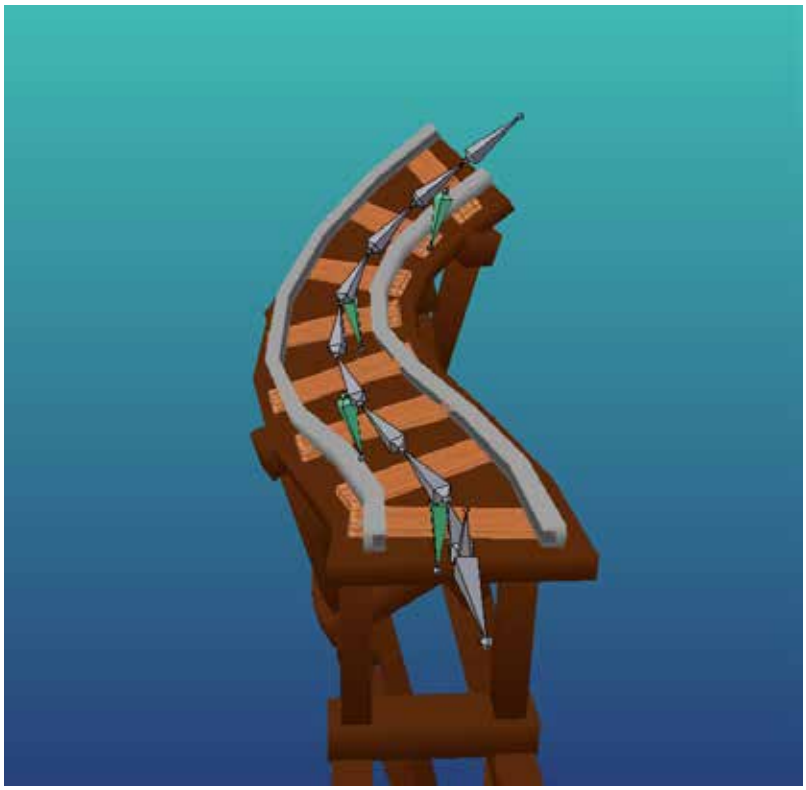
Above: An overview of one of *Skyrim*'s (2011) dungeons, clearly divisible into smaller modules

Skyrim dungeon map. Digital image. Blogspot. Web. <http://1.bp.blogspot.com/-JnrFv09byGA/Tr7txr9OFPI/AAAAAAAABik/ScZ2cw85n2E/s1600/bleak%2Bfalls%2Bbarrow%2Bmap%2Bpng.png>. 11 Apr. 2016

Using the AMMG system, a developer could instead generate a random map using roguelike modules. The map could be saved for further hand-tweaking and detail work. Developers would have to create fewer variations by hand, and could instead focus on creating unique structures or more maps.

5.1.2. Games with Surface Normal-Based Gravity

A significant problem of the AMMG system is that blending between two animations that operate in all three axes of movement can occasionally result in problematic poses. Blending between an animation to make a hallway curve left and another to make it slope down would lead to the hallway bending in an obtuse angle, and even if both original poses ended with both entrances being flush with the ground, the combined one is tilted in a totally new direction. The next module in the sequence will, depending on how the system is set up, do one of two things: either it will start in this strange new position, and the entire map generation will begin to fold over itself like a pretzel; or it will start with the original rotation, flush with the ground, in which case the two modules don't line up.



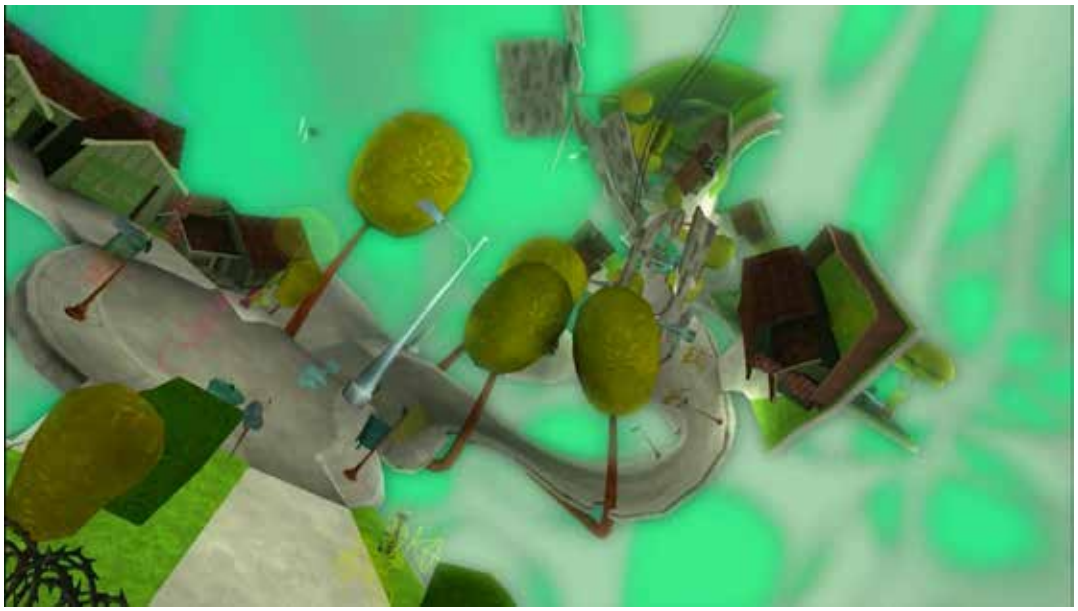
Above: Animation poses like this, where the end bone isn't level with the start, are problematic for most level generation purposes.

This was a hindrance for us. It limited which animations we could blend together, slowing down the entire animation process. There were potential ways to make the system consistently make modules end on a level surface, such as adding scripts that found the last few bones in a chain and rotated them, but we decided that it was far less time-consuming for our project's purposes to just work around it.

Above: *Super Mario Galaxy* (2007)'s winding caterpillar-shaped level could be easily randomly generated in infinite ways using our system

Mario Galaxy. Digital image. Destructoid. Web.<http://www.destructoid.com/elephant//ul/14087-550x-Screenshot%20Super%20Mario%20Galaxy_10%20small.jpg>. 11 Apr. 2016.

If we could somehow exploit this blending behavior to our advantage, it would really harness the potential of the AMMG system. The ideal way to do this is to make a game where gravity isn't always down, but rather locks to the normal of the nearest piece of ground to the player. Games like *Super Mario Galaxy* (2007) have used this mechanic in tandem with their level design to create unique, fantastic worlds.



Above: The “Milkman Conspiracy” level in *Psychonauts* (2005) featured surface normal based gravity.

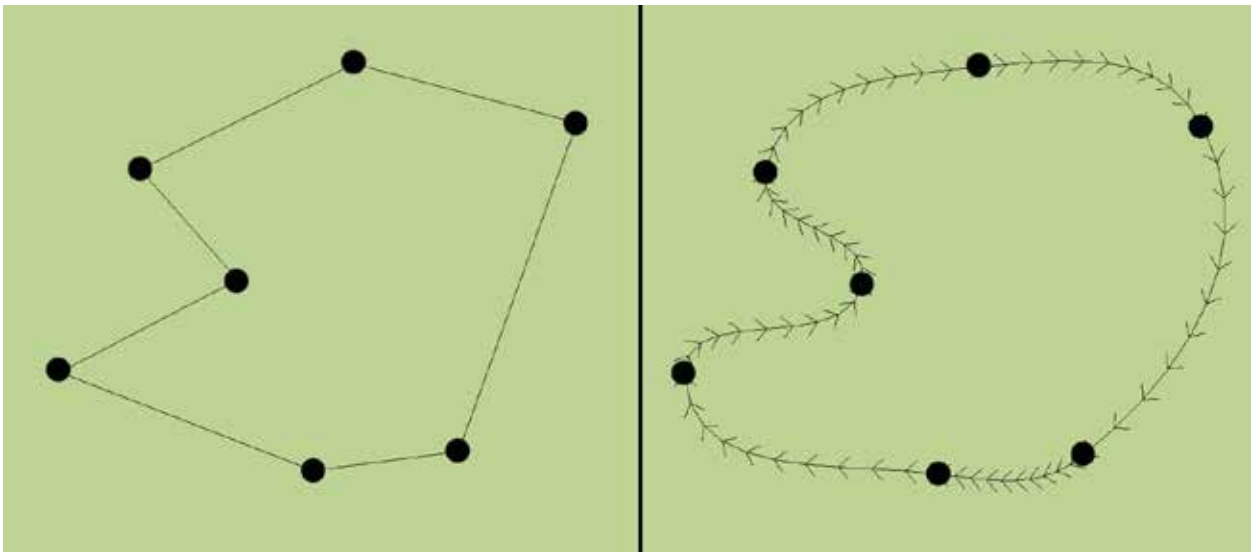
Mario Galaxy. Digital image. <https://i.ytimg.com/vi/zLMtNIVTXk0/maxresdefault.jpg>. 11 Apr. 2016.

Animating modules is only really worth it if it produces dramatic and interesting changes that affect gameplay. If the changes are small, then they may as well be cosmetic variations in a non-animated module generation system. Letting the system go wild and build spiraling non-Euclidean paths on all axes is a promising way to explore the full potential of the system.

5.1.3. Racing Games

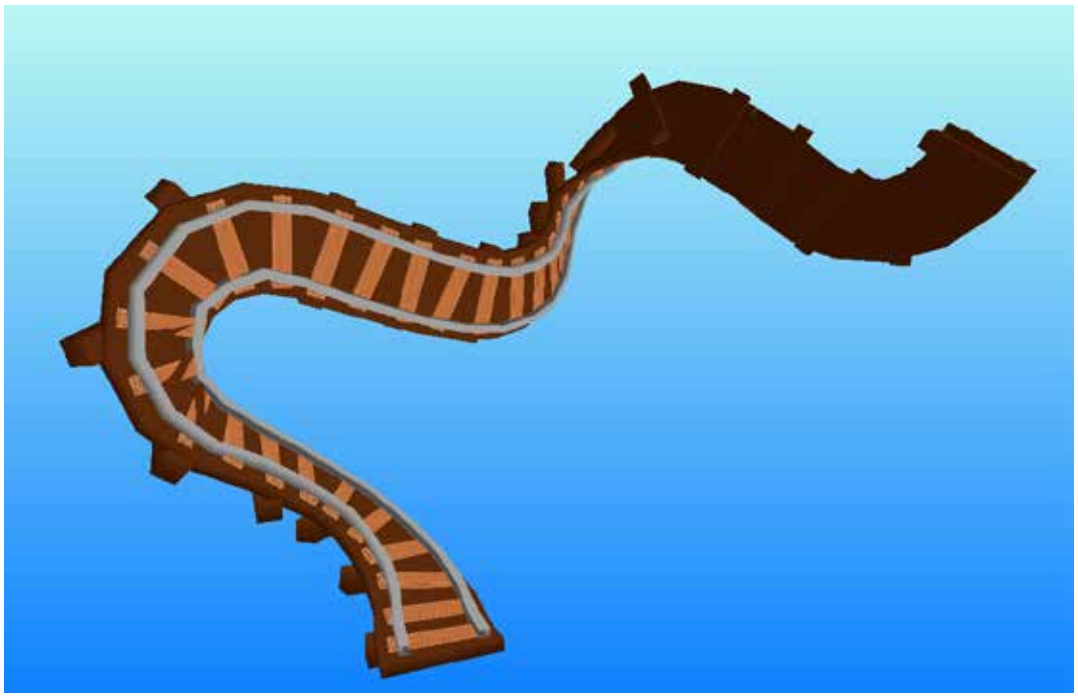
Though they can be deleted for optimization purposes, the AMMG system leaves behind the bones of the animated modules. When not connected to an animator or a skinned mesh, bones are very cheap memory-wise. They are essentially empty objects, containing only a basic transform (position, location and scale). We used them as spawn locations for enemies, which was convenient, but empty points can potentially be used in many other ways.

Many racing games use a series of waypoints along the track to guide the AI along the intended path. The bones left behind in the AMMG system could be used as such waypoints, allowing a designer to easily create all sorts of unique racetracks.

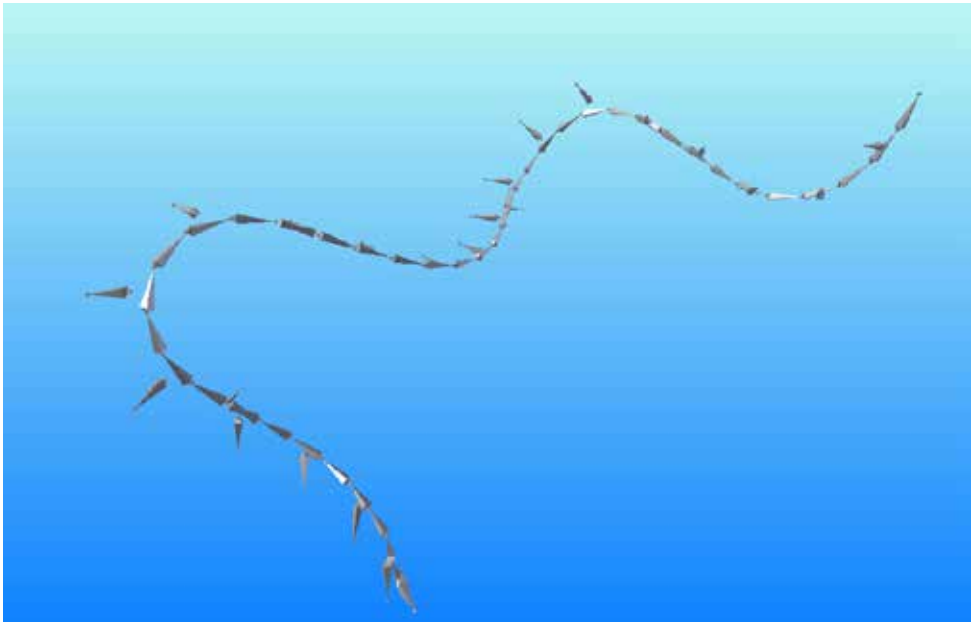


Left: A series of points, connected in an order (such as the order of bones in a rigged module.) **Right:** By automatically calculating waypoints between the main points, a smooth path is created, which can be used as the guiding path for AI in a racing game.

It should be noted that this racing system would work exceptionally well with the surface normal based gravity discussed earlier. Several racing games, including *Mario Kart 8* (2014), utilized a similar gravity system to create interesting and unique levels. The ability to procedurally generate racetracks of this complexity could lead to really interesting games. In fact, taking out the procedural generation and simply letting players design their own courses using these animated modules would be a creative and original approach.



Above: The AMMG system being used to its fullest potential. Our team spent a lot of time trying to avoid situations like this, but with a game built for it, it could have fantastic results.



Above: The skeleton of the previous image

Above: *Mario Kart 8* (2014) used surface normal based gravity as one of its core mechanics to create new twists on classic kart racing gameplay.

Mario Kart 8. Digital image. Hardcore Gamer. N.p., n.d. Web. 11 Apr. 2016. <http://www.hardcoregamer.com/wp-content/uploads/2014/11/mute-city-mario-kart-8-747x309.jpg>.

5.1.4. Abstract Games

One of the slower parts of the AMMG system was ensuring that modules matched up seamlessly. To avoid any seams, each module had to be properly weight painted properly. Allowing the edge vertices be weighted by any bones aside from the nearest node bone would make them shift slightly relative to the node, creating a visible seam between modules once generated. It shouldn't have been a big time loss, but it meant that even the simplest modules had to be manually re-weighted in parts.

Above: *Bastion* (2011) crafted levels out of unstable floating island chunks. The seams between terrain segments helped to enunciate the game's hectic post-apocalyptic narrative.

Bastion. Digital image. Scientific Gamer. N.p., n.d. Web. 11 Apr. 2016. <http://www.scientificgamer.com/blog/wp-content/uploads/2012/01/bastion41.jpg>.

This could be averted by making a game that doesn't risk having seams in the first place. If each module was its own distinct entity — say, a floating island, or an abstract shape not meant to line up with its adjacent objects — then there would be no need to waste time making sure the modules were perfectly rigged.

Above: *Bound* (announced 2015) has intentional seams between its level elements to emphasize its abstract modernist art style.

Bound. Digital image. Gamer Network. N.p., n.d. Web. 11 Apr. 2016. <http://cdn.gamer-network.net/2015/usgamer/Bound-5.jpg>.

5.2. What Would Benefit Least

Not every project would benefit from using the AMMG system. Obviously, the system is already fairly specialized — it can only be possibly relevant to 3D games that require lots of procedurally generated levels. Beyond this, however, there are games that *could* use the system but probably *shouldn't.*

5.2.1. Small Games by Small Studios

The AMMG system takes a lot of effort to implement. Perhaps, in the future, there will be easily referenced documentation and even convenient sample projects that can be used as a base and modified, but right now, any developer interested in using it will have to make it mostly from scratch. In a big company, a few people can be devoted to designing the basic elements of the system while other people work on other aspects of the game. In a small one, most of the team might be swallowed up trying to get the system going. It takes fairly complex coding

knowledge and also many technical art skills to be able to fully understand every part of the system, and some teams may not have anyone with the required skills.

Additionally, if a game is supposed to be a short experience, rather than a sprawling open world RPG or something with lots of replay ability, then it's probably not worth the time to build the system. A sophisticated method of procedural generation is best implemented when it's going to be used many, many times.

5.2.2. Environments That Are Both Indoors and Outdoors

This is a problem that extends to most module-based systems, especially procedural ones. Modules are often designed to only be seen from one angle, such as a cave from just the inside. Making every module work visually from all angles greatly increases the amount of time and effort required per module, as many easily-cut corners (hiding seams behind props, etc.) are impossible when the player has wide freedom of perspective. Our team originally planned to make a level that wove between cramped caves and wide open spaces, but quickly realized that changing between the two environments would be highly inefficient.

Games like *Skyrim* (2011) put a loading screen between the overworld and each individual dungeon. Aside from being an efficient way to load the game, this is used as a chance to separate the dungeons from the heightmap-based terrain of the overworld. We recommend that games with both indoor and outdoor sections segregate the two if either or both are procedurally generated.

Works Cited

Frade, M., de Vega, F., Cotta, C. (2010): *Evolution of artificial terrains for video games based on accessibility.* Applications of Evolutionary Computation pp. 90–99.

Abramson, D., and J. Abela. A Parallel Genetic Algorithm for Solving The School Timetabling Problem CiteSeerX. Australian Computer Science Conference, n.d. Web. 2 Dec. 2015.

Carreker, Dan. The Game Developer's Dictionary: A Multidisciplinary Lexicon for Professionals and Students. Boston, MA: Course Technology, 2012. Print. Pg 338
"Infer.NET." *Infer.NET*. N.p., n.d. Web. 11 Apr. 2016. <http://research.microsoft.com/en-us/um/cambridge/projects/infernet/>.

Moore, Gordon E. "Progress in Digital Integrated Electronics." *IEEE* (1975). Web.
"50 Years of Moore's Law." Intel. Web. 11 Apr. 2016.

Ong, Teong Joo, Ryan Saunders, John Keyser, and John J. Leggett. "Terrain Generation Using Genetic Algorithms." *Proceedings of the 2005 Conference on Genetic and Evolutionary Computation - GECCO '05* (2005). Web.

Togelius, Julian, Georgios N. Yannakakis, Kenneth O. Stanley, and Cameron Browne. "Search-Based Procedural Content Generation: A Taxonomy and Survey." IEEE Trans. Comput. Intell. AI Games IEEE Transactions on Computational Intelligence and AI in Games 3.3 (2011): 172-86. Web.

Shaker, Noor, Togelius, Julian & Nelson, Mark J. (2015). Procedural Content Generation in Games: A Textbook and an Overview of Current Research. Springer pp 20
Silverman, Ken. "Ken Silverman's Voxlap." Ken Silverman's Voxlap Page. Web. 11 Apr. 2016.