Procedural Generation of Tower Defense levels

Johan Öhman

Tutor: Erik Berglund
Examiner: Anders Fröberg
Copyright

The publishers will keep this document online on the Internet – or its possible replacement – for a period of 25 years starting from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© Johan Öhman
ABSTRACT
This report aims to present a method for generating levels for tower defense games. Tower defense games are digital strategy games played by defending bases against oncoming enemies that travel along pregenerated paths. The intended way of generating levels was to use Wave Function Collapse and Bézier curves to create paths and then generate a terrain mesh to match the paths using Marching Cubes. This approach was abandoned because it was considered unnecessarily complicated and there was a concern that there would not be enough time to complete a level generation system. The abandoned system was replaced by a system where a terrain mesh is generated first and is then used as the base for generating the paths iteratively where the direction of the path is changing based on a number of different rules. The path generation system was completed with a simple algorithm to decide on the number of enemies and defense towers. The result is a simple tower defense game with a complete but unbalanced system for generating levels. Though the system is unbalanced, it is implemented in a way that allows for balancing to be made. While a change in direction from the original idea was considered necessary, it only means that that approach was too time-consuming for this project, not that the techniques used in the original idea are inappropriate for these purposes.

INTRODUCTION
The release of the Atari game Rampart in 1990 marked the beginning of an era: the era of tower defense games [1]. During the 30 years since Rampart, the tower defense genre has maintained popularity, finding a new crowd with the inception of smartphones as a gaming platform.

While the genre has evolved and inspired many varied games over the years, the central theme of a tower defense game is to defend by preparation. The threat consists of a number of opponent objects that approach an object (base) that the player must protect. To achieve variation in gameplay, there are typically different types of opponents that can be combined into many different challenges for the player.

Another opportunity to achieve variation in gameplay is to change the layout of the level. Near infinite variation in gameplay can be achieved with procedurally generated levels, since no manual work is needed to generate them. The challenge is to match the quality of manually created levels, where effects on gameplay can be considered by the level designer. Factors that need to be considered include variation, playability and performance. The evaluation in this report will focus on playability.

Game
To perform the evaluation of the procedural generation techniques, they are implemented in a game that is developed for this purpose. The game shares many mechanics with classic tower defense games: attacking objects move along predefined paths towards the player base, which can be defended by destroying the attackers before they reach it. Attackers are destroyed by towers that can be placed alongside the paths.

New sections of the level are unlocked as the game progresses. This is a deviation from traditional tower defense mechanics, introduced to provide more progress to keep the player interested.

Research Question
The question this report aims to answer is:

How can procedural generation be used to create tower defense levels with high playability?

This question is focused on the implementation of a system for generating tower defense levels in the context of one particular game.
Delimitations
The intention is not to compare procedural generation algorithms but to attempt to achieve a desired result specifically in the context of this game project.

While the evaluation of the procedural generation techniques is done in the context of developing a game, the report will not cover the entirety of the development process. Only aspects that are relevant to generation will be explained in technical detail. For example, an explanation of the method used to create a mesh from the generated data is considered relevant context, while the code behind the game mechanics is not.

THEORY
Procedural content generation (PCG) is sometimes explained as the automatic generation of game content using algorithms. The word automatic is important in that it excludes content generated by designers and by players in-game, but it also poses a problem. Often, a certain amount of control over the generation is desired. Instead of calling procedural content generation automatic, it may be more useful to refer to it as having limited or indirect consideration for user input. Using this definition, we include content generation algorithms where it is possible to change parameters to tweak the result. [2] This report is only concerned with the generation of levels, and in this context there are two main reasons to tweak the generation: playability and variation. Tweaking the generation process may be focused on improving the playing experience for individual levels or to gain more control over the variation between different levels.

Playability can be viewed as two separate problems. A level must be playable in the sense that it doesn’t prevent the player’s progression. It must also provide a fun experience when played through. [3]

Wave Function Collapse
A relatively recent addition to the vast collection of PCG techniques is Max Gumin’s Wave Function Collapse (WFC) algorithm. The idea of WFC is to take a user-created input, for example a texture, and generate output in the same style. Since the output can be larger than the input, a designer can design the input to generate a large amount of content with desirable features. [4] WFC is not limited to generating textures but has for example also successfully been used to generate 3D structures [5]. Sandhu, Chen and McCoy [6] have showed that WFC generation is not limited to the original WFC algorithm, but that the algorithm can be modified to fine-tune the content generation.

Strategy Game Concepts
An interesting summary of desirable qualities in strategy games can be found in an article by Fabian Fischer [7]. A concentrated version of the ones most relevant for tower defense is described below:

Making strategic decisions is the fundament of strategy games. Enough information to base the decision upon is necessary to call it strategic, but making the decision too obvious impedes strategic thinking.

Coherence in a game system means that all the mechanics are deeply interconnected. No mechanic should seem detached from the rest and ideally, the system should be focused around a single mechanic.

Transparency means that the current state of the game is clear and that the consequences of the decisions to be made are foreseeable. Without transparency, it is difficult to make properly strategic decisions.

Elegance can be found in a simple game system in which the gameplay is made interesting through emergent complexity. Such complexity is a result of a few simple mechanics working together.

Efficiency is the frequency with which the player makes decisions. Since making decisions is the main gameplay feature of a strategy game, it is desirable that there is a minimum of wait time and pointless tasks that prevents the player from making decisions.

Low influence of luck is desirable because luck diminishes the importance of decision making and the value of the feedback received for that decision making.

Theme can be used to make the game rules and mechanics more intuitive to the player. By using behaviors that are conventional to a theme or the real world, some game mechanics may be instantly understood and need no explanation. As an example, the player could assume that a character equipped with a bow and arrow has a larger range of attack than one equipped with a sword. In a 2019 Gamasutra article [8], James Margaris elaborates on the negative effects of conflicting game mechanics and theme. The rules of the game may be difficult to learn for the player since they cannot be intuitively assumed, and the theme itself may seem more like an afterthought than a proper packaging of the game idea. An example shows that these negative effects can be so critical that a majority of a game’s reviews brings them up.

Unity
Unity is a popular game engine used for many modern game projects. [1] It supports procedural generation of meshes through scripting, which is an important part of the level generation in this project. Unity also supports compute shaders, which are programs that run on the GPU but can send data back to the CPU, unlike normal rendering shaders. Compute shaders have a speed advantage for programs

---

1 https://unity.com/madewith
2 https://docs.unity3d.com/Manual/GeneratingMeshGeometryProcedurally.html
where the high number of parallel threads of the GPU can be taken advantage of, and in this project compute shaders can be used to speed up the level generation.

**Perlin Noise**
Perlin noise is a type of gradient noise first introduced by Ken Perlin in 1985 [9]. Gradient noise is a type of noise where sampling from positions close in space generates similar values. This creates gradients, as opposed to a true pseudorandom noise where all values are independent. The original intention for Perlin noise was to generate natural looking surfaces for 3D models, but generated images can also be used as the input for other algorithms, for example generating a 3D landscape where the noise values decide the surface height.

**Bézier Curves**
The Bézier curve is a parametric curve first published by Pierre Bézier in 1968 [10]. The cubic Bézier curve is defined by a start point $P_0$, an end point $P_3$ and two control points $P_1$ and $P_2$. Any point on the curve can be calculated by the following formula, where $t$ is a value between 0 and 1 determining how far along the curve the point lies:

$$B(t) = (1-t)^3P_0 + 3(1-t)^2tP_1 + 3(1-t)t^2P_2 + t^3P_3$$

The Bézier formula does not provide an easy way of calculating evenly spaced points on the curve. When sampling points while linearly incrementing the $t$ value, the distance between the sampled points depends on the local curvature of the curve and will only be constant for the special Bézier curve that is a straight line. Sebastian Lague suggests an algorithm [11] that works by incrementing $t$ with small values and only saving sampled points that are far enough from the previous one. The algorithm uses a set spacing for the points, which results in the last point not always reaching $P_3$.

**Marching Cubes**
Marching Cubes is an algorithm used to construct a mesh from a set of sampled values. The original implementation by Lorensen and Cline [12] was used to create 3D models from medical images, with the intention of facilitating the interpretation of for example CT scan results. The 3D models generated by the Marching Cubes algorithm are more accessible than scan images to physicians not trained to interpret them, like radiologists are. A CT scan results in a number of 2D images from which the algorithm extracts sample point values and generates appropriate triangles. The points are in a grid pattern with a constant interval, and eight points at a time are considered, which means the algorithm is “marching” over the data and handling one cube at a time. Points are assigned a binary value depending on whether their sampled value is above or below a threshold value. The binary values of the eight points are combined into an integer used to look up an array in a pre-generated table, describing the triangles to be generated. The triangle arrays are composed of integers describing indices corresponding to the sides of the cube. This means that each corner of all mesh triangles is fixed to one line between points, but the exact position along the line is interpolated using the threshold value and the sampled values of the points. Instead of this interpolation, the midpoint between the points can be used, but the interpolation results in a smoother looking mesh.

**Fitness Function for Level Generation**
In a 2016 study by Stephenson and Renz [13], a procedural content generation algorithm is created and examined. The generated content is levels in a style imitating the one in the game Angry Birds. While the physics-based style of levels in Angry Birds is different from the tower defense style of levels covered in this report, the method of evaluation is not limited to a certain style of levels. Stephenson and Renz use a fitness function to determine the quality of a generated level. The fitness function is defined from features of the generated levels that are decided to be relevant in measuring their quality. Apart from providing a method of calculating level quality as a measurable number, the fitness function is used to let the generation algorithm improve itself. Since the levels consist of blocks, the algorithm can be tweaked by adjusting the probability of different blocks being chosen. This works by a training algorithm that generates levels and observes how frequently the different block types were in the levels and what value the fitness function yielded. In an attempt to accept user input, the block probabilities are saved as a table that is readable and editable. [13]

Obviously, when applied to a different level generator, another fitness function must be devised. And while the approach to evaluation used by Stephenson and Renz is applicable for tower defense levels, another difference must be considered: the tower defense levels are not constructed from a predefined set of blocks. Because of this, the method of assigning probabilities cannot be borrowed from Stephenson and Renz [13]. Other input parameters to the generation algorithm must be available for the training algorithm to adjust.

**METHOD**
A PCG implementation approach is selected and evaluated by a fitness function. The fitness function measures desirable properties of generated levels and can thus be used to assess the effect of changes made to the PCG implementation. The content of the fitness function is also iterated upon. In its early stages, the fitness function is based on literature and educated guesses. The final fitness function has been updated with additions and changes reflecting findings on what measurements are most relevant for the playability of a level.

3 https://docs.unity3d.com/Manual/class-ComputeShader.html
Implementation

The game is created using the Unity game engine, and the system for generating levels is integrated into the game. The level generation is not made abstract enough work separately from the rest of the game. Implementing the level generation in a way that lets it be used in different projects would be time consuming and is not prioritized within the limited timeframe of this project.

The level generation and the rest of the game are developed in parallel. A basic prototype for the game is developed before starting on the level generation, so that there is a basic structure in place to attach the level generation code to. For example, there must be a representation of a path before it can be used in the level generation. Letting such structures originate in the game code rather than the level generation is meant to let decisions about code structure be based on game logic, which is an attempt to facilitate the development process. The reason for not finishing work on the game logic completely before starting on level generation is that that would create a risk of spending to much development time on the game logic. It is impossible to know beforehand how much time every part of the project will take, and being able to adapt the scope throughout the process is important. Spending a set amount of time on the game logic before knowing how much time should be assigned to it could be counterproductive to achieving satisfactory results.

Two different methods for generating levels were used. As will be discussed later, the original idea was not to implement multiple versions, but the need to change direction presented itself about two thirds into the project timeline.

Method 1: The process for creating a level is divided into three parts. First, a basic layout is generated using WFC in the form of a texture. This texture is then used to generate smooth paths using Bezier curves and to create a terrain mesh that can be rendered in the game. The path and mesh generation are necessary steps to create a playable level for the game but are performed using deterministic algorithms. This way, the evolution of the PCG is isolated in the WFC part of the process. Keeping the path and mesh generation algorithms unchanged is meant to simplify the iteration of the generation process.

Method 2: The process for creating a level is divided into two parts: the terrain generation and the path generation. The terrain is generated as a plane where the $y$ value of each vertex corresponds to a sampled Perlin noise value. The path is then created based on this terrain.

Fitness Function

The evaluation of the quality of generated levels is performed using a fitness function. This fitness function is defined to account for important qualities in a finished level. While variation is also desirable, the fitness function is only used on one level at a time, and thus cannot assess variation between levels.

Desired properties in a level are:

Absence of too sharp turns. A path with a very sharp turn looks unnatural given the theme and is therefore undesirable. Identification of such turns is made by looking at each point at a time and calculating the dot product between the vectors between the point and its neighbors. A dot product value over zero means that the angle between the vector is less than 90 degrees, which is used as the definition for too small. Only path points with exactly two neighbors are considered. An end point with only one neighbor is irrelevant, and a point with more than two neighbors is an intersection, in which small angles are allowed.

High variance in path density. Places with high path density are often strategic points for placing towers, since the towers placed there can reach a large total stretch of path and thus be more effective. A high variance in path density results in different parts of the level having different amounts of strategic value.

Moderate number of path branchings. Branchings are strategic positions for placing towers since such towers can reach a large total stretch of path. However, too many such positions would diminish their strategic value.

High variance in height between path points. This is a measure of how much the path varies in height. Height difference has strategic meaning if towers have a bigger range when shooting down on targets below, otherwise it merely has an aesthetic value.

Sawtooth difficulty. Dividing the levels into parts that are unlocked continuously results in a variance in difficulty over time. When a new part is unlocked there is an increase in difficulty since the player must learn how to manage the new area. The difficulty is decreased while the player learns until a new part of the level is unlocked. This results in a sawtooth shaped curve of the difficulty as a function of time. The way a level is divided into parts must ensure that this claim holds true by putting enough strategic points in each part that unlocking a part actually changes the level.

The fitness function also includes a count of bases that are unreachable from paths. Levels with such bases are unlosable since those bases can never be destroyed, and are not considered valid.

The properties are combined into a complete fitness function as a weighted sum. The weights enable tweaking of the relative importance of the different properties. Negative weights are used for properties that have a negative impact on the level quality. Properties that make a level invalid can be given a large negative weight to ensure that the fitness value is low enough that the level is below the threshold and regenerated.
RESULTS

Tower Defense Design
Since the goal of this report is to assess the PCG techniques in the context of a tower defense game, it is of interest that the game is playable and not just a test environment for the PCG. To achieve a more playable game, a pre study was done to identify desirable aspects of tower defense games.

Based on the previously described list of desirable strategy game properties, a number of desirable properties specific for tower defense games can be devised. Based on coherence and elegance, the game should not contain more mechanics than necessary. The central mechanic of the tower defense genre is to place towers that will destroy oncoming enemies. When enemies travel along paths that are known to the player, strategic choices can be made about the placement of towers. The introduction of enemies that do not follow the path would decrease the overall coherence and elegance. An example of this is an enemy that flies towards its target, ignoring the paths. If the enemy does not fly in a predictable path, the property of transparency is also defied.

To increase elegance, emergent complexity should be extracted from the core mechanic of tower placement. A fundamental criterion for making placement a strategic decision is that different positions have different strategic value. The most obvious example is that a position where a tower can reach a large stretch of path has a higher strategic value, since a tower placed there has a larger capacity to damage enemies. To maximize strategic decisions, there should be a high variance of strategic value over the level, without having too many good positions that make the placement strategy obvious. To increase complexity, the strategic value of positions can be made to vary over time. One way to achieve this is through varying the level layout. In the interest of minimizing the transparency penalty, this is limited to unlocking new areas of the level. Unlocking a new area potentially unlocks more strategic points than was available previously, but also forces the player to rearrange towers to cover newly unlocked areas.

One way of simplifying the implementation of the level generation is to restrict level design to uniform tiles. While it is possible that this would increase transparency by making it easier for the player to assess the game state and predict consequences, it would limit the variance among strategic points, thereby decreasing the emergent complexity. Theme can also be a reason to avoid the tile-based approach in favor of more natural-looking smooth paths.

Based on efficiency, forcing the player to move the camera should be avoided, since it cannot be considered a strategic decision. A fixed camera position introduces restrictions on the level design, given that paths should not be occluded from view according to the concept of transparency. If the game benefits enough from levels with more complex layout, efficiency or transparency must be sacrificed rather than putting the restriction on level design. Theme could have an impact on the choice if it requires certain properties in level design that increase the risk of path parts being occluded from the camera, such as large height variation.

Another motivation for favoring height variation is to increase emergent complexity by introducing a strategic meaning to altitude. This can be achieved by giving tower projectiles a physics behavior that is affected by gravity, increasing the range of shots fired from higher altitude.

In a Gamasutra article on tower defense game design, Lars Doucet mentions the advantages of giving the player full control of time [14]. This means that the player is free to speed up, slow down, or stop time altogether while keeping the ability to make actions. The effect of this is an increased focus on strategic decisions since the player can avoid stressful situations. Doucet suggests that the player is given the ability to pause time or set the speed factor to $\frac{1}{4}$, $\frac{1}{2}$, 1, 2 or 4.

Paths
To facilitate the process of implementing the level generation, the paths are abstracted into collections of points independent of the generation method.

A path point keeps track of three things: its world position, a list of its neighbors and a boolean denoting whether the point is locked. The ability to lock points is used to prevent enemies to travel on paths that are in a locked section of the level.

Following the path works by moving the path follower towards the next path point and choosing a new point upon arrival. The new point is chosen among the neighbors of the point arrived at. Parameters can be passed to the function to control the choice, and by extension, the movement pattern of a path follower.

Iterated PCG Technique
The original idea for generating levels was a three-part process where points are generated, Bézier curve paths are generated from those points, and a terrain mesh is generated to match the paths.

The final implementation is different, both the details of the parts and the overall idea. The replacement is a two-part process: a terrain is generated based on Perlin noise and paths are then generated based on that terrain, without using the Bézier formula.

Terrain Generation
The terrain is generated by creating a square of triangles but calculating the y coordinate of each vertex from a Perlin noise function. The result is a one-sided open mesh with a terrain-like appearance. The generation function has a boolean parameter that decides whether the vertices should be reused for all triangles that has a corner in the same position. When reusing vertices, the vertex normal is calculated as an iterative mean of the surface normal of all
triangles that use that vertex. When using unique vertices, the normal of each vertex is set to the surface normal of the triangle they are part of. The normal used by the shader for calculating the light received by each pixel is interpolated from the normal of the three vertices that make up the triangle. This means that reusing vertices generates a smooth surface, whereas using unique vertices creates visible triangles where all pixels of the triangle use the same normal. The two different visual styles can be desirable for different art styles.

Figure 2: Comparison of terrain generation methods, with vertex normals displayed as red lines. The upper example is generated with shared vertices and normals, giving it a smooth appearance. The lower example is generated using unique vertices and normals, resulting in visible triangles.

Path Generation
The terrain is already created before the path generation step and will not be affected by the path. The paths are generated by extending a path from a random point on the terrain. Each extension step means a new path point is generated on a set distance from the previous end of the path. While the distance is constant, the direction from the previous point is calculated based on a number of rules that are designed to make the completed path meet some basic requirements for a level, but also yield a high value from the fitness function. This approach to path generation exposes the input as values used by the rules to decide how the path is generated:

- **maxPathPointCount** The maximum number of path points.
- **maxBranchings** The maximum number of branches created.
- **baseCountGoal** The desired number of bases.
- **spawnPointCountGoal** The desired number of spawn points.
- **randomSeed** The seed used to reset the sequence of random values when a new level is generated. Setting the seed before generating a new level ensures that the same input results in the same level being generated every time. It can also be used to deliberately generate different levels based on the same input.

The functions used to determine the direction of the next path extension are as follows:

- **Keep current direction** moves the path in the direction of the latest extension.
- **Avoid high ground** moves the path towards the lowest point on a set radius. Points are sampled on a circle with a radius of the distance between path points, and the direction towards the lowest point is calculated.
- **Avoid borders** keeps the path within the bounds of the terrain. Vector3.zero is used if the point is within the bounds of the terrain on the x or z axis. If the point is outside the bounds on an axis, the returned vector will have the corresponding coordinate set to point back towards the bounds. An additional float parameter can be passed to shrink the bounds. This is useful because the path may need a few segment iterations to have the chance to smoothly steer away from the border.
- **Avoid path** moves the path away from already created paths that are close enough to be a collision risk. The normalized sum of the directions pointing away from path points in range.
- **Avoid bases** moves the path away from already created bases that are close enough to be a collision risk. The normalized sum of the directions pointing away from bases in range.

The new direction is calculated as a weighted sum of the vectors returned by the rules. The weights are used to adjust the importance of the different rules and thus control the behavior of the path.
New branches are created by chance for every extension step. If the number of branches is not already the maximum number, there is a 5% probability that a new branch will be added. The initial direction of this branch is hard coded to Vector3.right, but this is only used as current direction when calculating the direction towards the first visible point of the branch.

<table>
<thead>
<tr>
<th>Levels</th>
<th>Size</th>
<th>Element 0</th>
<th>Random Seed</th>
<th>Max Path Point Count</th>
<th>Max Branchings</th>
<th>Branching Prob</th>
<th>Base Count Goal</th>
<th>Spawn Point Count Goal</th>
<th>Difficulty Aim</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Table Image]

Figure 4: The levels from Figure 3 as they appear in the form of input data in the editor.

**Level Economy Generation**

The physical appearance is only part of a level. The rest is the numbers deciding how enemies spawn and which towers are available to the player. Enemies are grouped into waves, where each wave has a list of enemies and timing information about how to spawn them. Tower availability is defined as the available number of towers of each type.

The level economy is the last part of a level to be generated since it depends on the paths. The values used from the level when calculating level economy are the density of path points and a difficulty value defined in the level input. Both values are used to calculate the number of enemy waves. The formula can be seen below. The basic formula can be seen below.

\[ \text{waveCount} = k \times \text{pathDensity} \times \text{difficulty} \]

The reasoning behind path density affecting the wave count positively is balancing: the denser the paths are, the more path can be reached by the average tower. \( k \) is a constant used for balancing. The number of enemies in a wave, or wave size, is a random number between 1 and 5, and the type of enemy is random. The wait time between each enemy is a constant divided by the wave size and the wait time after a wave is a constant multiplied by the wave size. All enemies of a wave use the same spawn point, and the spawn point is iterated for each wave.

The number of towers generated is a constant divided by the difficulty, since a level becomes more difficult the less towers are available to the player.

**Fitness**

The fitness function is implemented to be used in two ways: by the developer and in-game. To aid the developer, the results of the fitness function, both the total value and the individual values, are logged in the debug console of the Unity editor. In the game, the fitness function is used to decide whether a level must be regenerated. If the fitness value does not meet a set threshold, the game attempts to regenerate the level ten times with incremented random seed values. An error is raised if the level fails to be generated in all attempts.

![Figure 5: Example of debug logging of fitness function values in the Unity editor.](image)

Level economy is not included in the fitness function. The motivation for this is that since it is already based on abstract values that are easy to tweak, it is better to perfect the functions used to generate it rather than going through the fitness function.

![Figure 6: Debug view of part of a level, showing the division used to calculate path density variance and the path density value for each region. Note that only the bounding box of the path is used for this calculation.](image)

**DISCUSSION**

**Method**

The method involved developing the game logic alongside the level generation techniques. A positive effect of this was
that the game logic was always familiar and manageable. A negative effect was that it was sometimes difficult to maintain proper functionality in the generation part when changing the logic part and vice versa. A way of eliminating the negative effect while keeping the positive would have been to finish the game logic before starting on the level generation. That process would, however, have taken a substantial amount of planning to ensure that the game logic would not have had to be changed later, and there was not time for such planning in this project.

The fitness function sometimes seemed like more of a complication than a useful tool. Since the design of the fitness function was complicated in itself, a lot of time was spent tweaking it and there was often simultaneous work on the level generation and the fitness function. A more helpful use of a fitness function would have been to complete the fitness function early and then use it while implementing the path generation to see objective data of how it affected the generated levels.

Another important aspect of this work is the change in direction regarding the method for generating levels. The original plan for generation was partly implemented, but a decision was later taken that that method was not optimal for its intended purpose and, by extension, the possibility of creating a working level generation system. The three most important problems that helped form this decision are listed below.

Point cloud complicates input. The idea behind using Marching Cubes for mesh generation was to keep it general to make it able to handle any input that would be designed later. While it was known that Marching Cubes was originally designed to create a 3D surface at a specific density value in medical scan data [12], examples had also been seen where it was used to generate game worlds. These game worlds were however based on the same principle of generating a surface corresponding to a density value in a three-dimensional space of points (point cloud), whereas in this game, the surface had to be modified more directly. Since the algorithm calls for a point cloud, changes had to be applied to this point cloud to later be considered when creating the mesh in the marching step. The idea was that breaking the pattern of applying all changes via the point cloud would further complicate the process. It is not clear whether this was true since the technique was abandoned before any attempt to apply changes directly to the mesh was made. An example is when setting the height of a specific point of the mesh. First the closest point must be calculated, and the points below must be set to a value corresponding to being inside the mesh, and similarly, the points above be set to a value corresponding to outside the mesh. The closest point should then get a value resulting in the correct height being interpolated by the Marching Cubes algorithm to create its vertex.

Matching the terrain to the paths is complicated. A more complicated example of modifying the mesh is setting the height of the terrain mesh to match the paths running on it. In the reworked generation method, this problem is solved by letting the path height depend on the terrain and not vice versa, even though the absence of Marching Cubes likely would have made it more manageable to match the terrain to the paths. Leaving the terrain as generated according to noise also respects the natural-looking qualities of Perlin noise, since later modifications to the terrain are difficult to make blend in.

Unclear step from WFC to curves. The original plan was to start the generation process by creating a texture using WFC, from which a set of points could be extracted to create Bézier curves. Two problems with this became apparent:

1. Creating the points from a texture is not straightforward but requires an algorithm that outputs ordered points from the unordered points in the texture. Assuming the pixel’s position represents its position in the game world, the only data left to contain information about the order of and relation between points is the color of the pixels. This would be working against the nature of WFC, in which the colors of the input texture are exactly mimicked in the output. With only a limited set of colors representing different types of points or positions without points, the ordering of the points must take place outside of WFC, which limits its usefulness.

2. Starting with WFC and creating a level using deterministic functions (Bézier curve generation and Marching Cubes) puts all the focus on WFC to generate interesting output. Since this data is far from finished paths, there is a lot of abstraction between the input and the level it generates. This makes it unintuitive how to design the WFC algorithm and its input to tweak the level generation. In addition, problem (1) means that part of the process must be an interpretation step separate from WFC. This step is also abstract and hence suffer from the same difficulty of designing.

To prevent a failure to finish the project in time, a plan was made for assessing the value of the new idea for level generation. It was decided that one workday could be spent implementing a simple version of the new system before deciding whether to fully shift focus to it. This was meant to diminish the risk for unforeseen problems with the system. If there had been problems preventing a simple working version to be completed in one workday, the system could not be considered as intuitive as imagined, and thus shifting all focus to its further development would be a risk. It was estimated that allowing less than one day for the evaluation could have meant that the new system was not given a proper chance.

The impact of the change in approach is not entirely clear. Since the game logic was developed in parallel and there is no record of the amount of time spent on the different tasks, it is impossible to know exactly how many hours were spent on the abandoned code. Furthermore, while time was definitely spent on code that was ultimately deleted from the
project, some of the acquired knowledge could be reused. An example of this is the mesh generation, for which the basic method for generating unique or shared vertices could be implemented in the same way as for Marching Cubes.

Results
The level generation system is adequate for generating playable tower defense levels. While the generated levels are not perfectly balanced, the implemented system allows for balancing to be made. Since the system for generating paths is based on defining a set of rules for the path extension algorithm to follow, it can be easily expanded upon to adjust the levels to meet different requirements. This not only applies to the input variables exposed in the Unity editor, but also the rules defined in code. Since this part of the algorithm is separated into rules, it is easy to add or remove functionality. For example, it may be desirable to remove the rule that keeps the path within the terrain and instead let the paths end at the borders. It would have been preferable to put more functionality into this rule system to make it easily changeable. Some aspects of the level generation, like when and where a base should be created, is integrated into the loop where new path points are created because it was not originally made to be changed. This is due to both lack of foresight and lack of time.

It would also be useful to expose the rules of path generation in the editor, to be able to use different rules for different levels. While the original focus was not to obtain high variation between levels but to obtain high enough playability overall, variation is an important quality even though it is not measurable for individual levels.

An important point is that one big reason to expose as much level input as possible in the Unity editor is that the level input is saved there as a serialized class instance. Since a level is saved as input values that are then processed by the deterministic level generation system to create a level, there is no way to use levels from multiple versions of the code. If the levels were somehow saved as files describing for example lists of path points, levels could be generated by different versions of the code and still coexist in the game. The idea was that that approach would be more complicated to implement, and that changes to the code should not be necessary to tweak the levels anyway.

The level economy generation was not refined as much as the path generation. While the level economy is more abstract and therefore easier to generate and balance than the paths, it too has great importance for the playability of a level. The most interesting results relate to path generation, but the main motivation for also creating a system for generating level economy was to have a complete system for level generation. It can be useful if the system is expanded upon in the future, and if that is the case, it is likely positive that the complete generation system was developed simultaneously. While the path generation and economy generation are largely independent, the path generation design may have benefited from seeing levels in use with enemies.

CONCLUSIONS
The result of this work is a simple tower defense game with a useful system for generating levels procedurally. The system is based on creating a terrain and then creating a path by letting the paths expand based on a number of fixed rules. While the levels do not have as high playability as hoped for, the implemented system can still be considered an answer to the research question. Even though the system is not balanced enough to only generate levels with high playability, the fact that balancing is the only missing piece means that the system itself may be considered adequate.

An important aspect is the change in direction that occurred about two thirds into the project. The original idea for generating levels was conceived while collecting reference material for the work, but was deemed too complicated after starting implementation. It is important to acknowledge that this does not show that the Marching Cubes and Wave Function collapse algorithms are not suitable for level generation in general. With more time, it is possible that the original implementation could have yielded a satisfying result. However, the fact that a change in direction was considered necessary means that at least the original idea was less fitting for the purpose than originally believed.

REFERENCES


