Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Fakultät für Mathematik und Informatik
Institut für Informatik
Lehrstuhl für Informatik IX
Games Engineering

---

**Bachelorarbeit**

zur Erlangung des Grades
Bachelor of Science (B.Sc.)
im Studiengang Games Engineering
an der Julius-Maximilians-Universität Würzburg

# Self-Organised Construction of Particle-Based 3D Artifacts

---

vorgelegt von
Lenny Siol
Matrikelnummer: 2334375

am 27.03.2023

*Prüfer:* Prof. Dr. Sebastian von Mammen
*Betreuer:* Prof. Dr. Sebastian von Mammen
Lehrstuhl für Informatik IX
Julius-Maximilians-Universität Würzburg

## Zusammenfassung

In dem virtual reality Spiel "Swarm Tower Defense" muss der Spieler mithilfe von Schwarmagenten seine Basis verteidigen. Er tut dies, indem er die Agenten Ressourcen sammeln lässt, mit denen sie dann Verteidigungstürme bauen. Das Ziel des Spiels ist es, die Gegner davon abzuhalten, die eigene Basis zu erreichen. Die Verteidigungstürme müssen strategisch platziert werden, um die Gegner, die jedes Level mehr und stärker werden, abzuwehren. Um den Bau eines Turms zu initialisieren, platziert der Spieler zunächst eine Markierung auf der Karte. Wenn ausreichend Ressourcen zur Markierung geliefert wurden, ist der Bau des Turms abgeschlossen. Das Ziel dieser Arbeit ist es, das bereits existierende, binäre Turmsystem durch organisch wachsende Türme zu ersetzen. Um das organische Wachstum der Türme zu simulieren, werden diese während des Spiels prozedural generiert. Wo und wie die Türme wachsen, wird dadurch beeinflusst, an welcher Stelle die Schwarmagenten ihre Ressourcen abwerfen. Das Ergebnis ist ein Tower Defense Spiel, in dem der Spieler Schwarmagenten beeinflussen kann, organische Turmstrukturen zu bauen. Diese Art des Turmbauens bietet eine neue Spielmechanik für das Tower Defense Genre, ohne zu weit von etablierten Mechaniken abzuweichen.

## Abstract

In the game "Swarm Tower Defense", the player has to defend his/her base by controlling a swarm of agents in virtual reality. The player commands the agents to gather resources and build defending towers. The goal is to prevent waves of enemies from reaching the player's base. The player has to strategically place defensive structures in order to withstand increasingly more and stronger enemies with each level. To build a tower, the player places building markers on the map. When enough resources are delivered to a marker, the tower construction is finished, and the tower is fully built. The goal of this work is to replace these already existing binary towers with organically growing towers. This is done by procedurally generating the towers during runtime, using the coordinates where the swarm agents deposit their resources.

The result is a tower defense game in which the player can influence the agents to build organic tower structures. The way the towers are built presents novel gameplay challenges for the player while also not deviating too much from established mechanics used in tower defense games.

# Contents

# 1 Introduction

Simulating a swarm is a complex topic that can yield interesting and unpredictable results. Defining the behavior of a single swarm agent and observing how complex behaviors emerge when a group of them interact with each other has a unique appeal. There are many different applications, like simulating the behavior of an ant colony in order to try to solve complex problems like the "traveling salesman problem" (Chen & Chien, 2011) or more abstracted behaviors like controlling the movement and coordination of enemies in video games like "Project Zomboid" (2011).

## 1.1 Background

One of these applications is the virtual reality (VR) game "Swarm Tower Defense," developed as part of a research lab at the University of Würzburg. As the name may suggest, the game combines swarm mechanics with the tower defense genre. In the game, the player takes control of a swarm of agents modeled to look like mechanical dragonflies. The player's task is to defend his/her base from oncoming waves of enemies. This is done by influencing the swarm to gather resources from defeated enemies. The resources are then used to build defensive structures which destroy the enemies or slow them down in their advance.

The player advances through different stages, facing increasingly more and stronger enemies. This forces the player to build more and more defensive structures as the game progresses. Should the player survive each and every stage through the strategic placement of the defensive structures, he/she wins the game.

## 1.2 Goals

In this thesis, the building of the defenses is expanded upon. In the existing system, the player can place a construction marker on the map using their hands in VR. The

player can choose from three different types of construction of markers, resulting in one of three towers. While under construction, the swarm agents can be guided to deposit resources to the marker. Once a certain resource threshold is passed, the tower's construction is finished. The construction marker disappears, and the tower becomes active.

When active, the towers function depending on their type. The game has single-attack towers that damage the nearest enemy at regular intervals. The shock tower has less reach and damage but damages all nearby enemies, and the shield tower generates a roadblock that slows down the enemies' advance. While keeping the spirit of this three-typed system, the interaction of swarm agent behavior and how the towers are built is changed.

Since the agents are already represented as small robotic insects, the idea is to organically grow the towers like an insect hive would. To achieve this, the tower meshes are procedurally generated during runtime using the agents' positions when they drop off a carried resource. This system replaces the old towers and, therefore, the simple two-step building process. The new system allows the swarm to create novel, organic tower shapes while retaining most of the control the player has over the tower placement. The goal is to create towers unique to the tower defense genre and integrate them into the existing game in a visually appealing way while also impacting the gameplay meaningfully.

## 1.3 Methodology

To achieve this, information about different procedural content generation (PCG) methods and how PCG is used in combination with swarms and tower defense games is collected. The desired algorithm is then narrowed down to Marching Cubes. First, a single Marching Cube is implemented. The cube is then tested in a testing scene separate from the main game. Then edge weights are added, and a simple grid of cubes is created. The grid is then tested by assigning random values to each point. Afterward, a simple mock agent is implemented. The agent moves around and deposits resources randomly. After ensuring the cubes behave correctly, simulated gravity is introduced to let the resources fall to the ground. Next, the system is switched from a single resource type to multiple resource types. Then the system is tested again with multiple simple agents that drop different types of resources.

With this in place, a surface shader is created, and the placement of the predefined elements is implemented too. Then the functionalities for the different tower types are added. Lastly, the new system is ported to the existing game scene. After ensuring the system works with the actual agents, the game is then tested to adjust damage numbers and movement speed modifiers. To ensure that the new system works, the game is played multiple times, trying different settings. The performance is measured, and the towers are graded according to predefined criteria.

## 1.4 Detailed Gameplay Loop

To give context, this section briefly describes how the game is played, what options the player has, and how the player wins the game.

The game begins with a tutorial. The map is empty except for the agents idling and a dead enemy on the ground. A tutorial text instructs the player to commandeer the agents to gather resources from the dead enemy. When the player does this, he/she is prompted to construct his/her first tower (1.1).


**Figure 1.1.** The beginning of the game. The player is instructed to build his/her first tower.

The player does this by first placing a marker. The player places a marker by grabbing it from a selection table next to the game map and then dropping it at

the desired position. This is done using VR controllers (1.2). In the beginning, the player only has one type of marker, which builds laser towers. The player first unlocks the shield and then the shock tower as the game progresses.


**Figure 1.2.** The player is placing a marker with the controller. The marker is snapped to the ground plane when the trigger is released.

Now that the marker is placed and the agents have gathered resources, the player can create a path using a spray can. The player can shape the path by moving the spray can with the VR controller while holding down the trigger. When agents come near the path, they follow it according to their swarm logic (1.3).

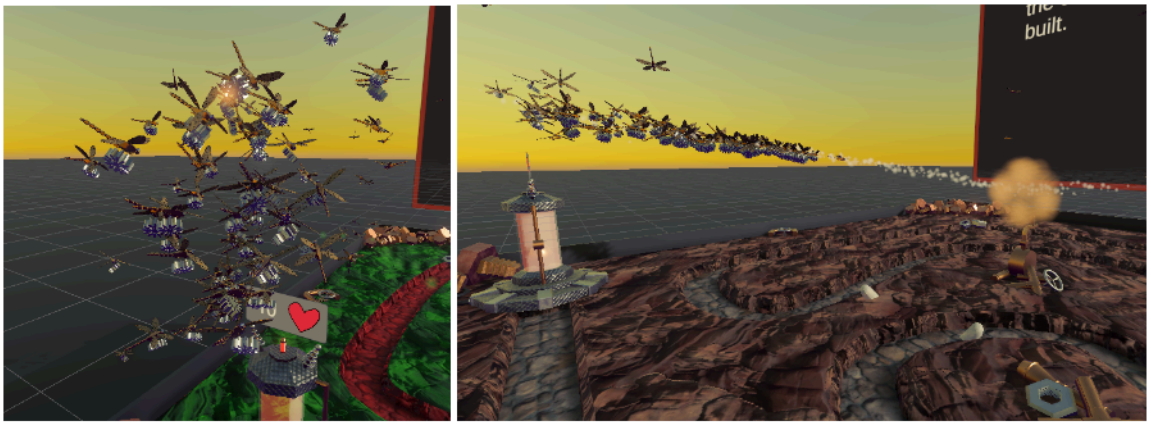


**Figure 1.3.** The agents are carrying resources. When they encounter a path set by the player, they follow it.

If following this path results in agents colliding with a marker, they drop their resource if they are carrying one. The resource falls to the ground. On impact,

the Marching Cube mesh is modified, creating the tower mesh by deforming the ground plane. The ground is increasingly deformed with each dropped resource, slowly building up the tower structure (1.4).

**Figure 1.4.** When agents carrying a resource collide with a marker, they deposit their resource. This, in turn, grows the tower. After a set amount of resources is deposited, the marker disappears.

When enough resources have been dropped, the marker disappears. The player has grown their first small tower, and the tutorial is finished (1.4).

Now the game begins. The first enemies spawn and follow a path to the player's base. Should they reach the player's base, the player loses a health point. The player loses the game and has to start over should his/her health points be reduced to zero. While the enemies move to the player's base, they are attacked by the laser tower, reducing their health. The laser tower attacks the nearest enemy every 6 seconds. When an enemy's health is reduced to zero, they die and cease moving. Now the agents can collect new resources from their bodies. The player can repeat the process from the tutorial to build new independent towers or place markers near existing towers to grow them. Once all enemies are defeated, the player progresses to the next stage. With each stage, the number of enemies grows. Also, enemies with more health are spawned. After a couple of stages, the player unlocks the shield towers. All towers are built with the same mechanics. The only difference is the type of marker the player selects. The shield tower can be grown in the path of

the enemies, slowing them down. After more stages, the player unlocks the shock tower. The shock tower throws an orb at the nearest enemy, damaging all enemies in the area around the hit target. Increasing the size of a tower increases its strength. Laser towers, for example, deal more damage with a single attack the bigger they are. The player can also combine different tower types. Each tower can have a main type and a sub-type. Depending on the sub-type, the base tower behavior changes. When the player manages to finish all waves without having their health reduced to zero, they win the game.

## 1.5 Tower Grading

For the towers to be valid, they must fulfill several aesthetic and performance criteria. Another goal is to provide a novel gameplay challenge. Since the tower placement is an essential mechanic and requires intelligent use of resources and strategy (Kraner et al., 2021), having the towers organically grow and fuse with each other will impact the player's marker placement. It will also be expanded on the tower typing, replacing the mono-typed towers with towers that have a main type and a sub-type, further incentivizing the player to grow the towers smartly.

To construct the tower meshes, the "Marching Cubes" algorithm is used. This allows the creation of hive-like shapes from low-level data and easily ties in existing assets into the towers' visuals. To add more visual fidelity to the tower surfaces, a shader is created using Unity's shader graph. The shader is designed in a way that it can overlay different effects depending on the tower's type.

The result is towers that are organically grown by the agents. The player retains a good amount of control over where he/she wants to grow the towers and how they are shaped. The tower functionality differs enough from the old system to create a novel game experience. The towers fulfill all predefined criteria. Namely: Towers do not appear to be flying, always consist of complete meshes with no holes, can be constructed in parallel, and fuse when they grow into each other. Their construction can be interrupted and resumed at any point. They can be shaped by the player and modified by the developer. The predefined parts are correctly placed on top of the tower surface. All while not impacting the performance in a disruptive way.

# 2 Related Work

This chapter will show what procedural content generation is and how it has been applied in tower defense games. It can also be seen which methods can be used to generate content procedurally and how they have been used in combination with swarms.

## 2.1 What is Procedural Content Generation?

"Procedural content generation in games refers to the creation of game content automatically using algorithms" (Togelius et al., 2011). This could be anything from textures to dialogues up to whole games themselves (Mark et al., 2015). In this case, it would be the generation of the defending towers. There is already a broad application of PCG in the game context. Having the content created automatically has many benefits. It can replace or support designers in cutting down on cost and time, which is necessary due to a growing population of players and an increasing demand for content. This is further amplified by an increasing need for quality of said content (Hendrikx et al., 2013). As a survey by Freiknecht and Effelsberg (2017) shows, PCG has been widely applied not only in digital but also analog games such as "The Settlers of Catan" (Teuber, 1995) adding a high degree of replayability. There are also already examples dealing with tower defense games, such as Du et al. (2019) and Öhman (2020), whose algorithms automatically generate new levels for tower defense games.

Du et al. (2019) build on an already existing game by analyzing paths, tower locations, and enemy sequences and then generating new "building blocks" from the data. These building blocks are then used to generate new levels using varying PCG methods for each type of building block.

Öhman (2020), on the other hand, uses PCG to generate the whole level from scratch using the values of noise textures as geometry data for the generated ground mesh and paths.

## 2.2 What is a Swarm?

Swarms can be seen in many forms in nature, like in flocks of birds, schools of fish, or colonies of ants. A swarm is made of many independent actors that navigate based on their local perception. Many stimuli like their environment, simulated physics, or pheromone tracks can be used to influence their behavior. The combined motion of the swarm results from the relatively simple behavior of each and every agent (Reynolds, 1987). Swarms have a long history in computer science, and various algorithms for modeling swarm behaviors have been iterated (Karaboga & Akay, 2009). A widely used model is the bird-like "boids" first defined by Reynolds (1987). The basic behavior of a boid is defined by three simple rules:

- Collision Avoidance: The boid avoids collision with other boids.
- Velocity Matching: Boids try to match the velocity of other nearby boids.
- Flock Centering: Boids try to stay close to each other.

The behavior that emerges from these relatively simple rules has found a wide array of applications.

## 2.3 Using Swarm Agents for PCG

To tie in the swarm mechanics of the tower defense game, how swarm agents are used in the PCG context will be looked at. Cabezas and Thompson (2013) use a swarm algorithm for PCG. The swarm is given a starting point and a deformation pattern by the user. Then the agents traverse the terrain modifying the height of vertices nearby. The height modification is done with respect to the neighboring vertices. Similarly, de Andrade et al. (2020) use the swarm agent's behavior to generate aesthetically engaging 3D animations. The swarm agents traverse a landscape, thereby adjusting the height of the traversed topology. Both approaches show

how the swarm's social behavior can shape meshes in novel ways, providing unique results.

## 2.4 How is PCG Applied in Tower Defense Games?

A tower defense game is a digital strategy game in which the player has to defend a base from oncoming enemies. The enemies follow a predetermined path, and the player has to stop them. This is achieved by building towers that either destroy the enemies or stop them from reaching the player's base in other ways (Kraner et al., 2021) (Tan et al., 2013). So far, PCG has mostly been used to generate the terrain that towers are placed on, path generation for the enemies' movement and enemy spawn behavior, as well as generated levels' balancing. Some research also builds on this terrain generation, testing path-finding algorithms trying to efficiently traverse the terrain, like in the research of Liu et al. (2019) where the enemies try to find an efficient path to the base.

## 2.5 Using Marching Cubes for PCG

Marching Cubes is a 3D surface construction algorithm. It is a cell-by-cell method that creates isosurfaces from scalar volumetric data sets (Newman & Yi, 2006). One example of its use in a tower defense game is the implementation of Öhman (2020), where it is used to procedurally generate a terrain mesh as the floor for the level. A big advantage of Marching Cubes is that it can create high-resolution 3D surfaces from a simple 3D array of data (Lorensen & Cline, 1987).

# 3 Methodology

This section explains how the towers are generated from low-level data, how the agents influence this low-level data, and how the visual fidelity of the generated towers is further increased using shaders and handmade 3D assets.

## 3.1 Tower Construction

The tower meshes are created using the Marching Cubes algorithm, which generates a collection of individual triangle meshes. These meshes can then be combined to form the final tower mesh.

Starting out, a single Marching Cube is created. The cube consists of a base coordinate and eight corner points. The corner point positions are initialized using a set of direction vectors scaled to the size of the cube.

The corner points hold the actual information about how the tower should be built. Each corner point holds a value between -32 and 16 for each tower-type resource. If one of these values exceeds zero, the point is considered "inside." If all values are smaller than zero, the point is considered "outside." Since each cube has eight corners and each corner can either be inside or outside, the number of possible cube configurations is $2^8 = 256$. Many of these configurations are symmetries. A lookup table is used to determine which corner configuration produces which set of triangles. The lookup table returns a list of edges for the corner configuration. Each edge is related to two corner points. Then another lookup table is used to retrieve the points related to each edge. Given these sets of points, the midpoint of each edge can be determined. The midpoint positions are saved to a list. This list can be split into sets of three, returning the corner coordinates for each triangle. Then each triangle can be drawn. The resulting cube can be seen in figure 3.1. The corner points can not be seen in the final scene. They are visualized here for clarity. To add more visual interest to the towers, the position of the triangle corners on the

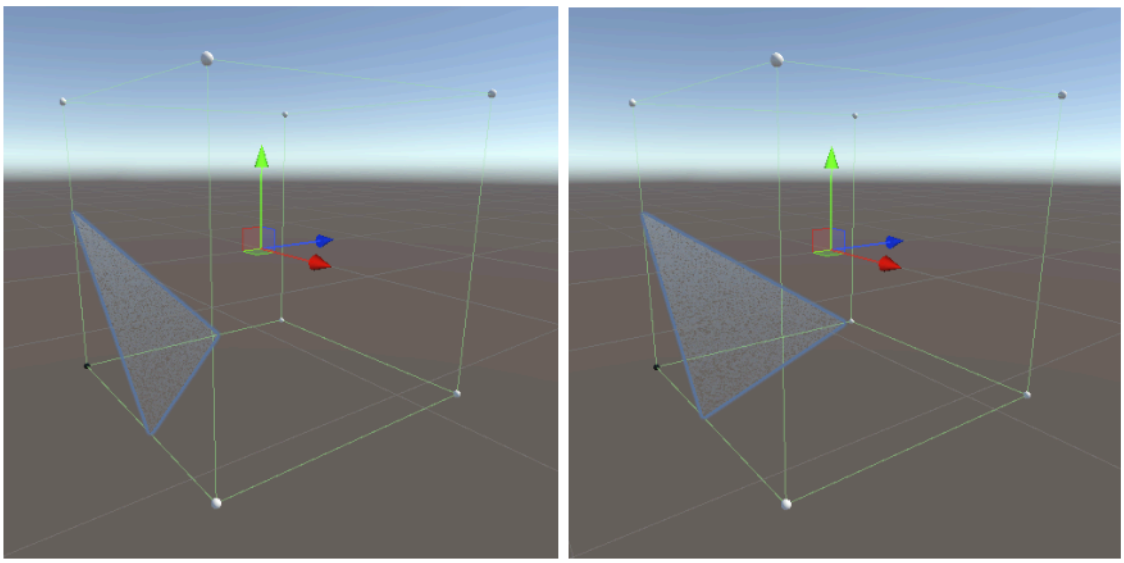


**Figure 3.1.** A single Marching Cube. The "outside" corners have been visualized with white spheres, and the inside corners with black spheres. The generated triangle can be seen in the bottom left corner. The left picture shows a Marching Cube without weighted edges, and the right picture with weighted edges, shifting the triangle.

edge can be adjusted using the resource value of the corners as weights. This also makes the growth of the tower look more fluid since the triangle corners move along the edge instead of snapping between the midpoints of the corners. The influence of the weights can be seen in figure 3.1.

Another thing that can be done is assigning different colors to the triangle vertices depending on which resource type is primarily present in the corner points. This makes it possible to easily blend different colors or materials within a single triangle.

An evenly spaced grid is needed to set up multiple Marching Cubes for the scene. This grid is created using an Octree structure. While using a complex structure like an Octree to generate this grid is unnecessary, the structure has several advantages, as will be discussed later. An immediate benefit is that the resolution, i.e., the number of cubes per volume, can easily be set by adjusting the depth of the Octree. The Octree works recursively, creating eight smaller Octree branches inside of itself. Such a branch is also called a "child." This is done for each depth iteration until the maximum depth is reached. The branch created at the maximum depth has no children and is also referred to as a "leaf branch." At the maximum depth, instead of creating new branches, the leaf branch creates a cube instance. A cube instance is a

3D coordinate with eight corner points. The way the Octree and the cubes are set up, the corner points of neighboring cubes have identical coordinates at the cubes' corners. This ensures that the generated triangles are always properly connected to triangles of neighboring cubes.

When creating the grid, the values are randomly set for each point, outside or inside. As can be seen, the edge coordinates represented by the red spheres in figure 3.2 match the position of their respective neighbor cube's edge coordinates. The result is meshes composed of multiple Marching Cubes.

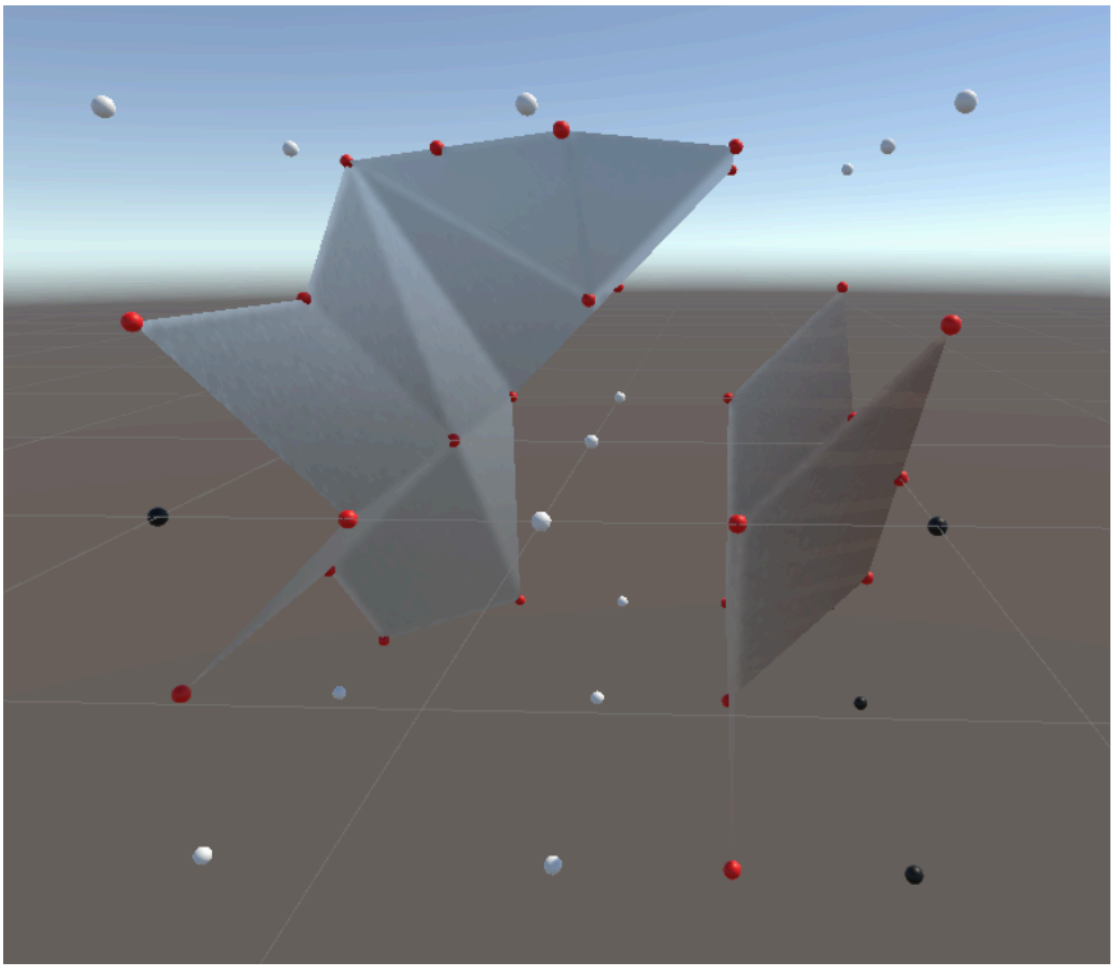


**Figure 3.2.** A grid of unweighted Marching Cubes. The red spheres mark the edge coordinates for the generated triangles.

Now a simple agent can be added to the scene. The agent increases the values of nearby points at random intervals. Doing this yields random floating structures as shown in figure 3.3. The resulting structures are already close to the desired

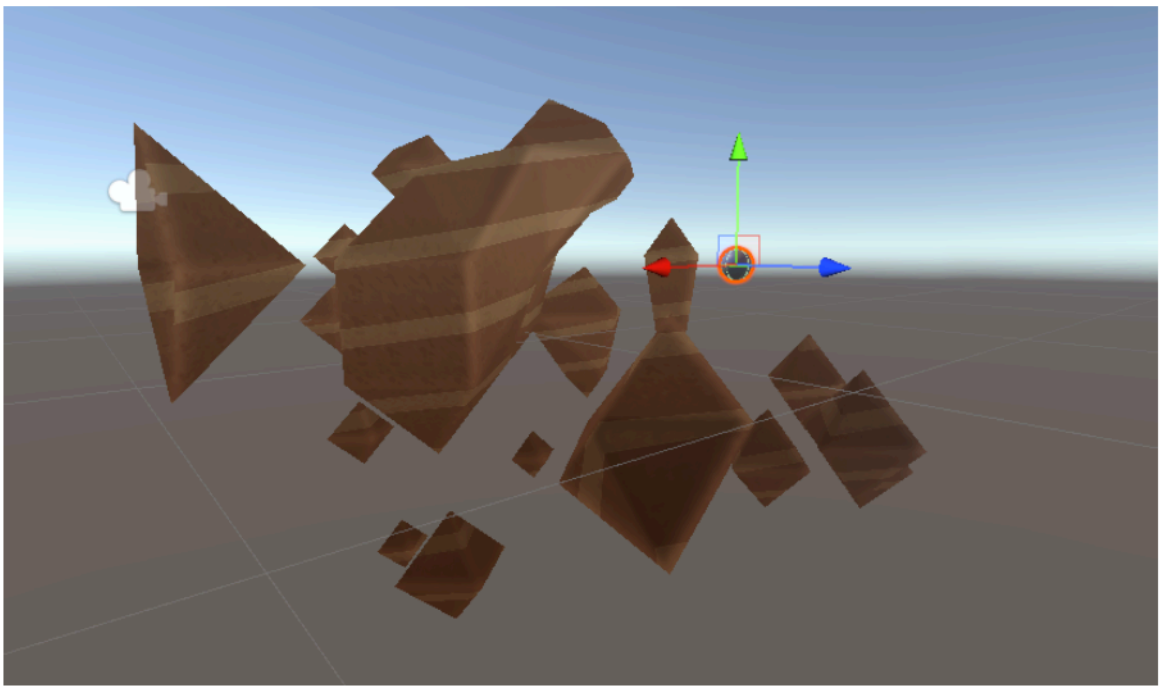


**Figure 3.3.** Random structures generated by a simple agent. The agent can be seen on the right, highlighted in orange.

outcome. However, one of the defined criteria is that all parts of the towers must either be connected to the ground or another tower. To do this, a ground plane is defined by setting all points below a given coordinate to be inside. Now gravity needs to be implemented for the deposited resources to "drop" to the ground.

### 3.1.1 Resource Dropping and Gravity

Should an agent drop a resource, the resource's position is snapped to the grid of points. This is achieved by rounding the position values to the grid's resolution and then applying the value increase to the point in the grid with the same coordinate. To check if the resource has hit the ground, the state of the point directly beneath the current position is retrieved. If the point beneath is inside, building a tower fragment at the current position yields a tower connected to either the floor or is built on top of an existing tower. The value of a point can further be increased after its state has been set to inside, and this value is used to determine the edge weights. Therefore, it is ensured to increase this value first before starting to build on top of it.

Should none of the point's neighbors be inside or fully saturated, the resource "falls"

by trying to apply the new value to the neighbor point beneath the current position. This process can be repeated until the resource hits the ground or leaves the grid. In the latter case, the dropped resource is discarded. With this in place, valid towers can already be built. However, their structure will always be similar, forming heap-like towers as shown in figure 3.4(left).

To allow more interesting structures to be built, it is checked if the neighbor beneath and any of the other neighbors are inside. Implementing this allows for overhangs and, therefore, more complex structures like arcs to form while still ensuring that no part of the tower appears to be flying. An example of the resulting structures can be seen in figure 3.4(right).

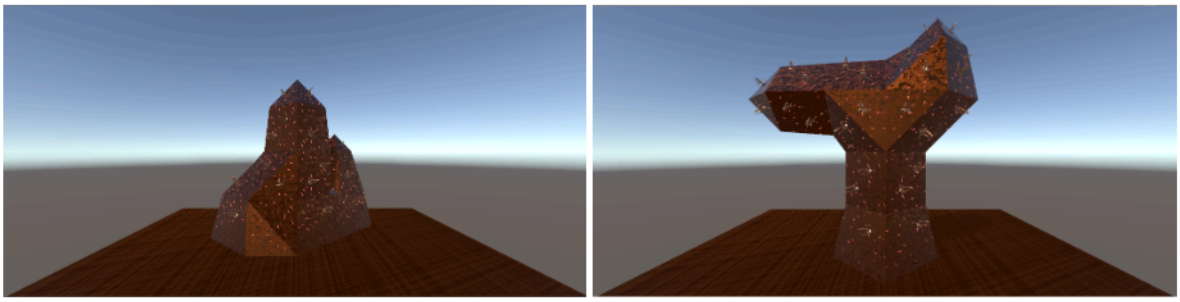


**Figure 3.4.** Towers constructed by a simple agent, dropping resources with simulated gravity. The structure on the left was constructed, only allowing towers to form on top of each other. The structure on the right was constructed, allowing towers to be built as long as any of the surrounding points are already a tower.

### 3.1.2 Tower Placement

Now that the agents can randomly generate valid towers, a way for the player to take control of the tower building is needed. To do this, a marker system is used: To initialize the construction of a tower, the player can place a marker. The marker can be one of the three tower types. It can be placed anywhere on the map. When a marker is placed, a sphere collider is added above the marker. When an agent carrying a resource collides with the marker, the agent drops the resource, and the tower is partially constructed. After a set amount of resources has been dropped, as a result of the agents colliding with the marker, the marker disappears. Should the player want to continue the construction of a tower, they can place another marker near the existing tower. This marker system allows the player to have multiple towers constructed simultaneously. It also prevents the map from being

quickly filled with too many markers resulting in a loss of control for the player. Additionally, it prevents the player from accidentally allocating too many resources to a single tower by accident.

### 3.1.3 Octree

As already described, an Octree is not necessarily needed to create the grid. However, when modifying a point, the eighth surrounding cubes are affected, and their triangles must be redrawn. To find the affected cubes efficiently, the Octree structure can be used. To do this, the algorithm starts at the root position of the Octree and then traverses to each child node that is closer to the modified point than the current position. This process is repeated for each child node selected in the previous step. This is done until the maximum Octree-depth has been reached. At this point, the algorithm has traversed to the leaf nodes containing the cubes. These cubes are the cubes neighboring the point. The complexity of this search is $\mathcal{O}(8 \log n)$, with n = Octree-depth. This search also allows us to easily find the nearest point of any given coordinate. After finding the eight cubes near a coordinate, their shared corner coordinate is returned. Additionally, points at the edge of the grid that have less than eight surrounding cubes are also covered by this approach.

### 3.1.4 Shaders

To increase the visual fidelity of the towers and also for gameplay purposes, a shader is applied to the generated tower surface. The shader is made using Unity's shader graph. The shader has four major components: A base surface that is present no matter what type the tower has and three semi-transparent components that are overlaid depending on the tower's type. The base component is designed to look like a plain copper surface. The overlaid components are animated and are representative of the tower-type functionality. The four components can be seen as separate and combined in figure 3.5.

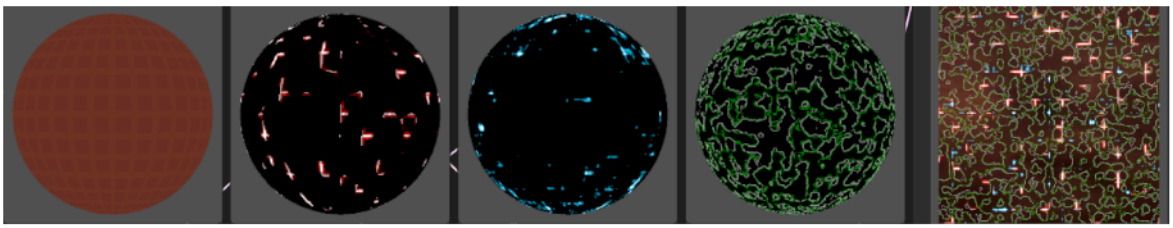


**Figure 3.5.** Surface shader components representing the different tower types. From left to right: Base color, laser, shock, shield, all types combined.

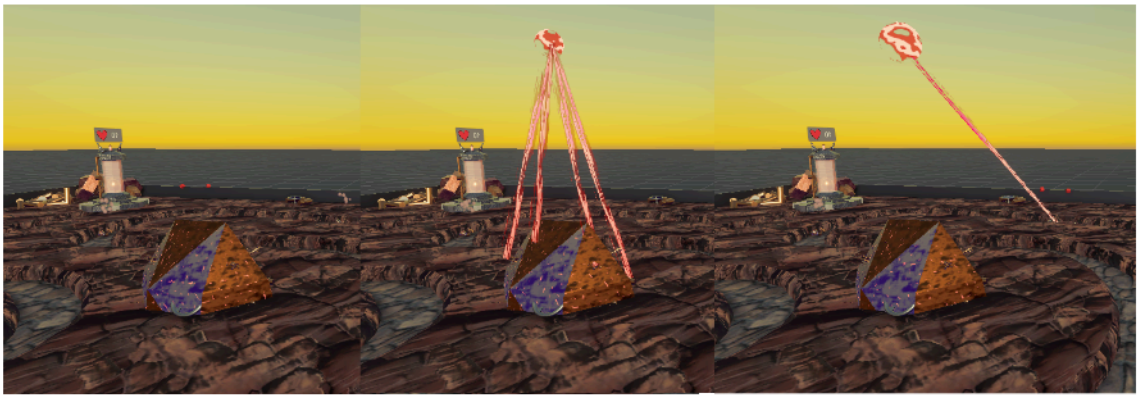


**Figure 3.6.** The process of a laser tower firing at an enemy. First, the tower is idle. Then all tower objects on the surface planes fire a laser at a combined spot. The lasers make a laser orb appear at the target position. The orb grows over time and gets bigger with each surface object contributing. Then the laser orb fires a laser at the nearest enemy, dealing damage depending on its size.

## 3.2 Tower Functionalities

The new tower functionalities are reflecting of the three tower types the game had before the tower generation was implemented. The old tower types were:

- A single attack tower that shoots a fiery projectile at the nearest enemy at regular intervals.
- An area-of-effect attack tower that damages all nearby enemies with lightning at regular intervals.
- A shield tower that generates a wall on a nearby lane. When an enemy encounters a wall they have to spend time destroying it before they can move on.

Respectively the new tower functionalities are:

- A laser tower that charges up and then fires a continuous laser at the nearest enemy.
- A shock tower that throws a projectile at the nearest enemy in an arc. On impact, the projectile damages all enemies in the area around the impact location.
- A shield tower that slows down enemies when they walk through it. While traversing the tower enemies slowly lower the value of the nearest point. This makes the tower slowly disappear as more enemies pass through it, giving the illusion that the enemies "chew" through the tower.

All towers benefit from a bigger size, forcing the player to decide whether to combine towers for these benefits or to have a bigger number of smaller, weaker towers.

When a laser tower is bigger it deals more damage but can only attack one enemy at a time. Multiple smaller laser towers deal less damage but can split their fire on different enemies. One big shock tower covers a large area in one location, while multiple small shock towers cover smaller areas in multiple locations. A shield tower's strength increases inherently with size since a larger covered area means the enemies will be slowed for a longer time. Still, the player can choose to have multiple small shield towers in strategic locations, like near the damaging towers.

Figure 3.6 shows how the combined strength is visually communicated to the player.

## 3.3 Combining Types

To add another layer to the tower building complexity the combination of different tower types is implemented. This opens up more different approaches the player can make when playing the game. The combination of tower types is designed in a way that offers the player a clear trade-off. Combining types gives the tower all benefits of its main type and a smaller benefit of the sub-type. While building two separate towers yields the full benefit of all the resources used, combining tower types lets the player benefit from the increased size of the tower as well as fitting more tower functionality in a smaller space.

The benefits of combining tower types can be seen in table 3.1.

**Table 3.1.** Combination of Tower Types

| Type | Laser (MT) | Shock (MT) | Shield (MT) |
|---|---|---|---|
| Laser (ST) | X | Deals more damage to each enemy hit | When consuming the tower, the enemy is damaged significantly |
| Shock (ST) | Damages all enemies in a small area around the target | X | When consuming the tower, all nearby enemies are damaged slightly |
| Shield (ST) | Slow the target enemy significantly | Slightly slows all enemies hit | X |

(MT = Main type, ST = Sub-type)

# 4 Results

## 4.1 Tower Evaluation

To evaluate the success of the algorithm, the aesthetics of the generated towers are quantified as measurable values. The following section describes the criteria that were defined for the towers beforehand. For each of the criteria, the degree to which they are satisfied is evaluated.

**Cohesion:** All parts of the mesh have to be connected to either the rest of the mesh or the ground. No parts of the towers should appear to be flying.

As explained earlier, the towers can only grow in points adjacent to either the ground or another tower's surface. Therefore this constraint is satisfied.

**Completeness:** The towers must not have any visible holes exposing the backside of the generated surfaces.

When the Marching Cube algorithm is applied correctly, it is guaranteed to return complete meshes that do not expose any backsides. The only exception is the edges of the grid where the cubes do not have any neighbors on one or more sides. Setting these points to be inside would expose the backside of the tower mesh at the edges of the grid. To prevent this, the state of all outermost data points is set to be permanently outside. With this, not only do the towers not show any backsides, but they are also complete at the edge of the grid, satisfying this constraint.

**Performance:** The game must maintain a minimum of 60 frames per second during runtime. Since the game is developed as part of a research lab at the University of Würzburg, it must achieve this within the limits of the hardware in their laboratories.

The game's performance is tested on the following system:

- Processor: Intel Core i5-6600K
- Memory: 16 GB
- Operating System: Windows 10
- Graphics Card: NVIDIA GeForce GTX 1060 6GB
- Virtual Reality Device: Oculus Rift and Touch

Over a runtime of 5 minutes, an average amount of 145 frames per second is achieved with a peak of 813 frames and a minimum of 89 frames. No perceivable frame drops were encountered. While the constraint is satisfied, it should be noted that performance improvements could be made nonetheless.

**Adjustability:** The generated mesh should be easily adjustable, allowing users to adjust the aesthetics to their liking.

When it comes to the generated mesh, multiple components can be adjusted. On the developer's side, the resolution of the Marching Cubes mesh can be easily adjusted by changing the Octree's resolution. Additionally, the surface look can be changed by adjusting the shader graph for the surface shader. This allows the developer to achieve a different base look and can also be used to adjust the color and patterns of the different tower-type overlays. This can be useful for achieving different looks or implementing accessibility settings such as a color-blind mode.

On the user's side, the resolution chosen for the Marching Cubes in the game scene allows for a meaningful degree of control. The user is able to create shapes like overhangs and arcs reasonably easily despite the varying amount of control due to the swarm mechanics used for building the towers.

Since the developer and player have multiple easy ways to adjust the generated towers, this constraint is satisfied.

**Parallel Construction:** Building multiple independent towers is strategically important in TDGs. The player has to be able to start the construction of two or more independent towers at the same time.

There is no limit on how many markers that initialize the tower building process can be placed at once. Additionally, towers can be constructed in parallel, and the

building state of a tower has no influence on whether or not other towers can be built. Therefore the constraint is satisfied.

**Coordinated Construction:** In the case that two towers would grow into each other and start intersecting, they have to merge at the point of intersection. Since changing the value of a point causes all neighboring cubes to update, the merging of towers works identically to building a single tower, and the constraint is satisfied.

**Continuous Construction:** If a tower is no longer supplied with resources and stops to grow, it has to be able to resume growth as soon as resources are available again.

Since the resource of a point can be modified at any time independently from any other gameplay system, stopping the supply of resources will result in the tower not growing anymore. Resuming to supply the turret will continue its growth, satisfying the constraint.

**Integration of Predefined Assets:** Lastly, it has to be ensured that predefined assets, such as shield generators or cannons, are correctly placed on the generated surfaces and continue to do so as the tower grows around them. While not defining a strict minimum distance between the assets, they should also not intersect with each other.

The assets are placed on the surface of the triangles. The position is the average vector of the triangle's corner coordinates, giving us roughly the middle of the triangle. The normal of the triangle is used to define the orientation of the assets. All this ensures that the assets are correctly placed on the tower's surface. Since each cube has only a set amount of configurations, the assets are scaled in a way that ensures that they will not intersect with each other, therefore satisfying this constraint.

## 4.2 Gameplay Impact

In addition to the tower visuals, it should also be considered how well the new tower mechanics tie in with the already existing gameplay loop and how the change of mechanics impacts the player's experience. While the tower's functionalities have

mostly been adapted from the earlier system, the new system offers some new gameplay opportunities and challenges.

One new mechanic is that the player has to be careful about combining many small towers into one bigger tower. This forces the player to make strategic decisions regarding tower placement.

Another new mechanic is that the player has to constantly rebuild their shield towers since the towers are partially destroyed when an enemy passes through them.

Lastly, the combination of tower types. Since the functionality of a tower is defined by its main and sub-type, the player has to pay attention to not waste a third type of resource in the construction of a single point. The sub-sub-type would have no benefit to the tower's functionality. The player also has to make a strategic decision about which two tower types to combine and which of these types should be the main type and the sub-type.

Since the game is fully playable and can be finished with the new mechanics, the constraint is satisfied.

## 4.3 Example Towers

In this section, towers grown within the gameplay loop and towers grown decoupled from gameplay restrictions are shown.

### 4.3.1 Towers Grown With Gameplay Restrictions

The shown towers are grown with the goal of winning the game. The growth over a whole game can be seen from a bird's eye view in figure 4.1. In figure 4.2 the process of two independent towers growing together over time can be observed. Figure 4.3 shows structures generated after completing the game.

### 4.3.2 Towers Grown With No Gameplay Restrictions

The following towers were built outside of the regular game loop. To build them the resource costs and the restrictions on marker placement were removed. Doing this allows for different structures to be built. In figure 4.4 a high tower with many overhangs was built. In figure 4.5 a flat sprawling tower structure was created. In figure 4.6 a mixture of the first two approaches was used to create one big structure with smaller structures around it.

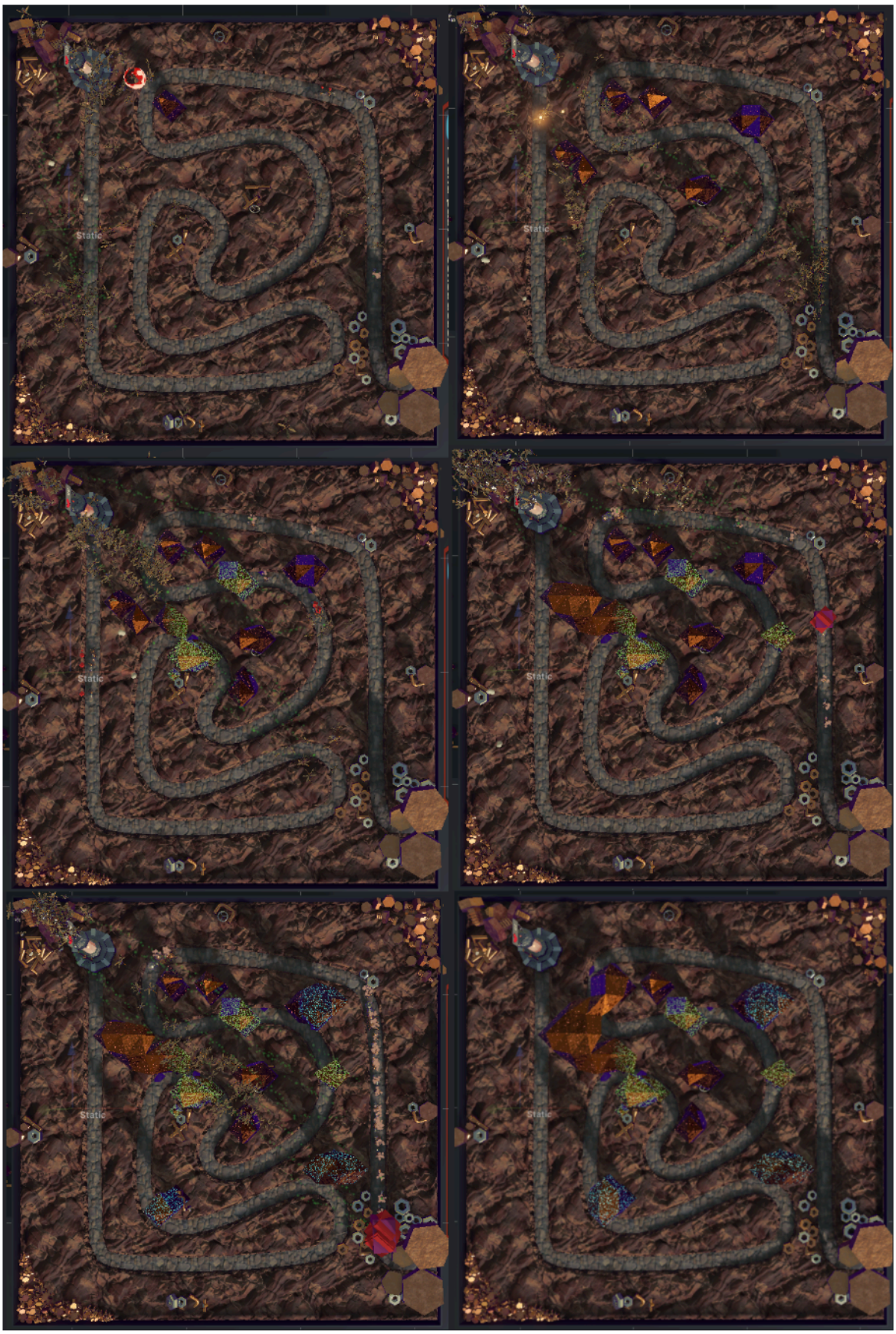


**Figure 4.1.** The tower building progress over the span of a whole game.

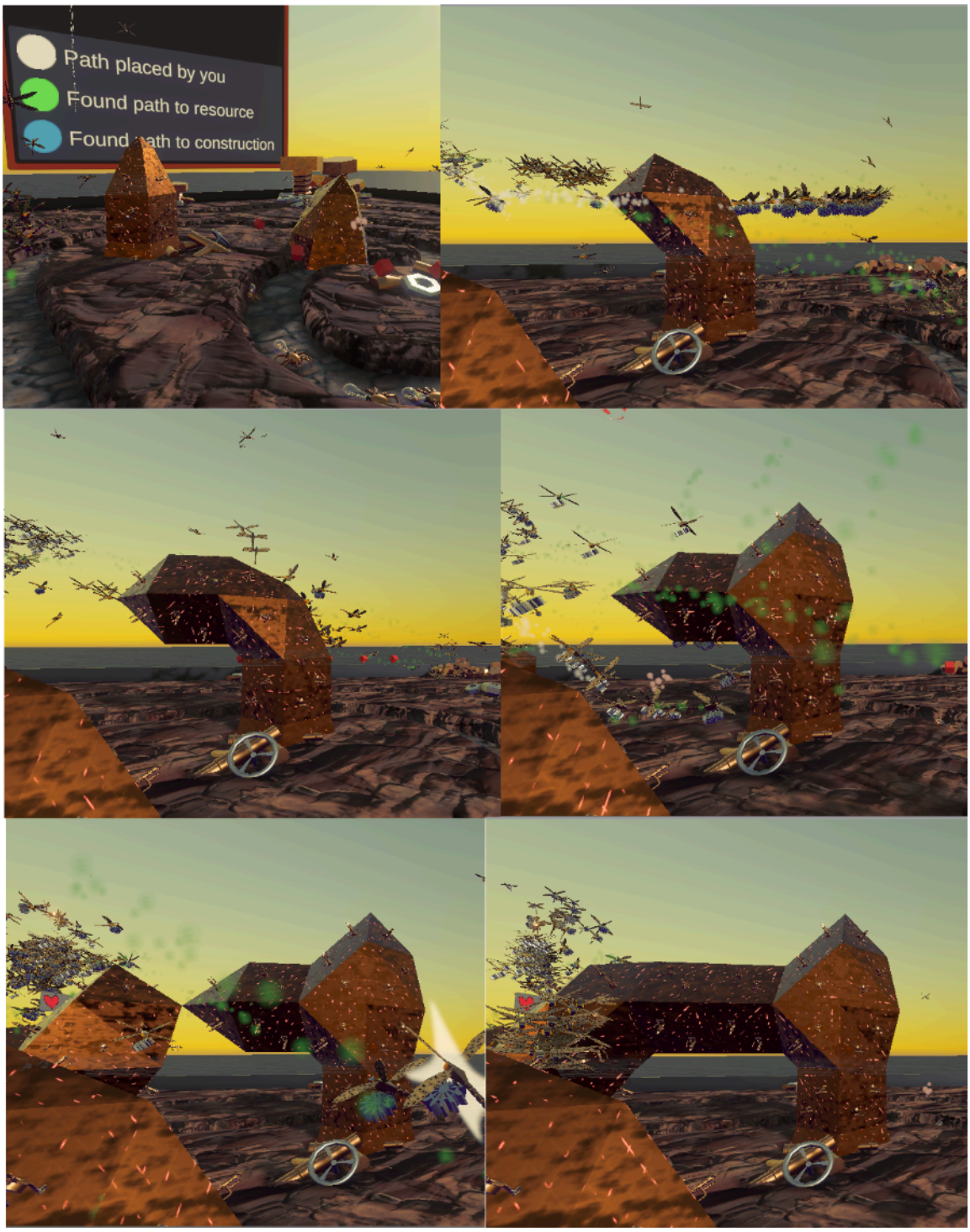



**Figure 4.2.** Two separate towers that grow together as the game progresses.

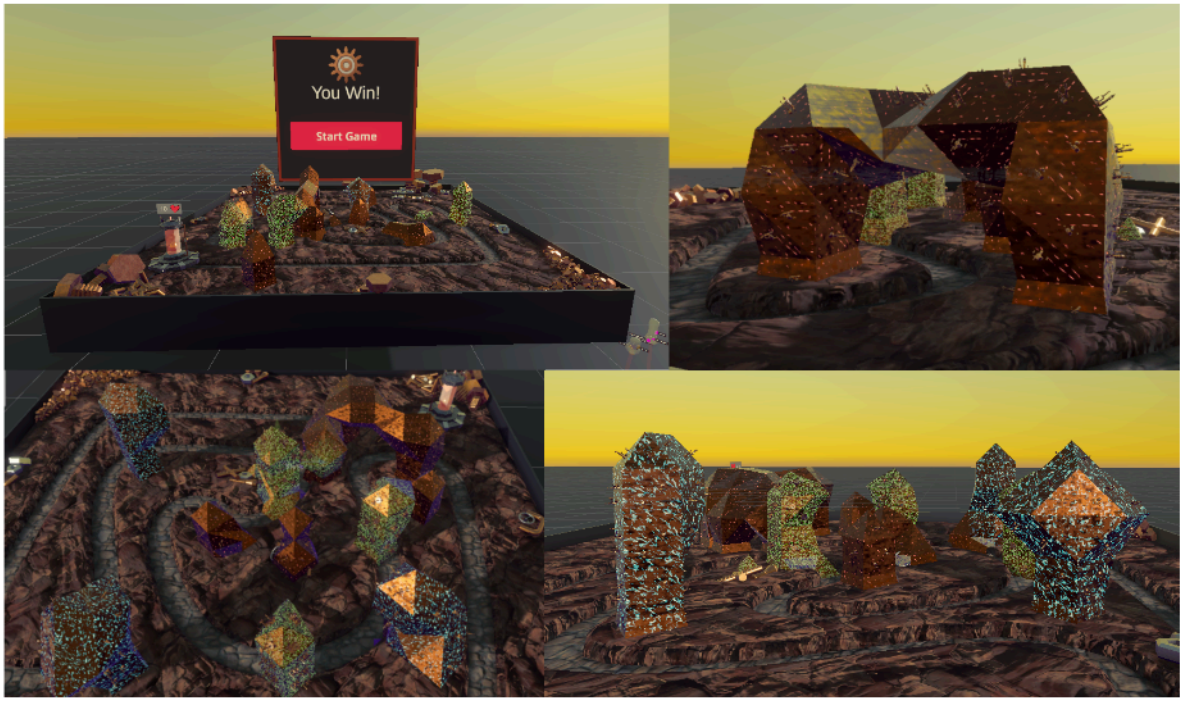



**Figure 4.3.** Tower structures at the end of a successful game.

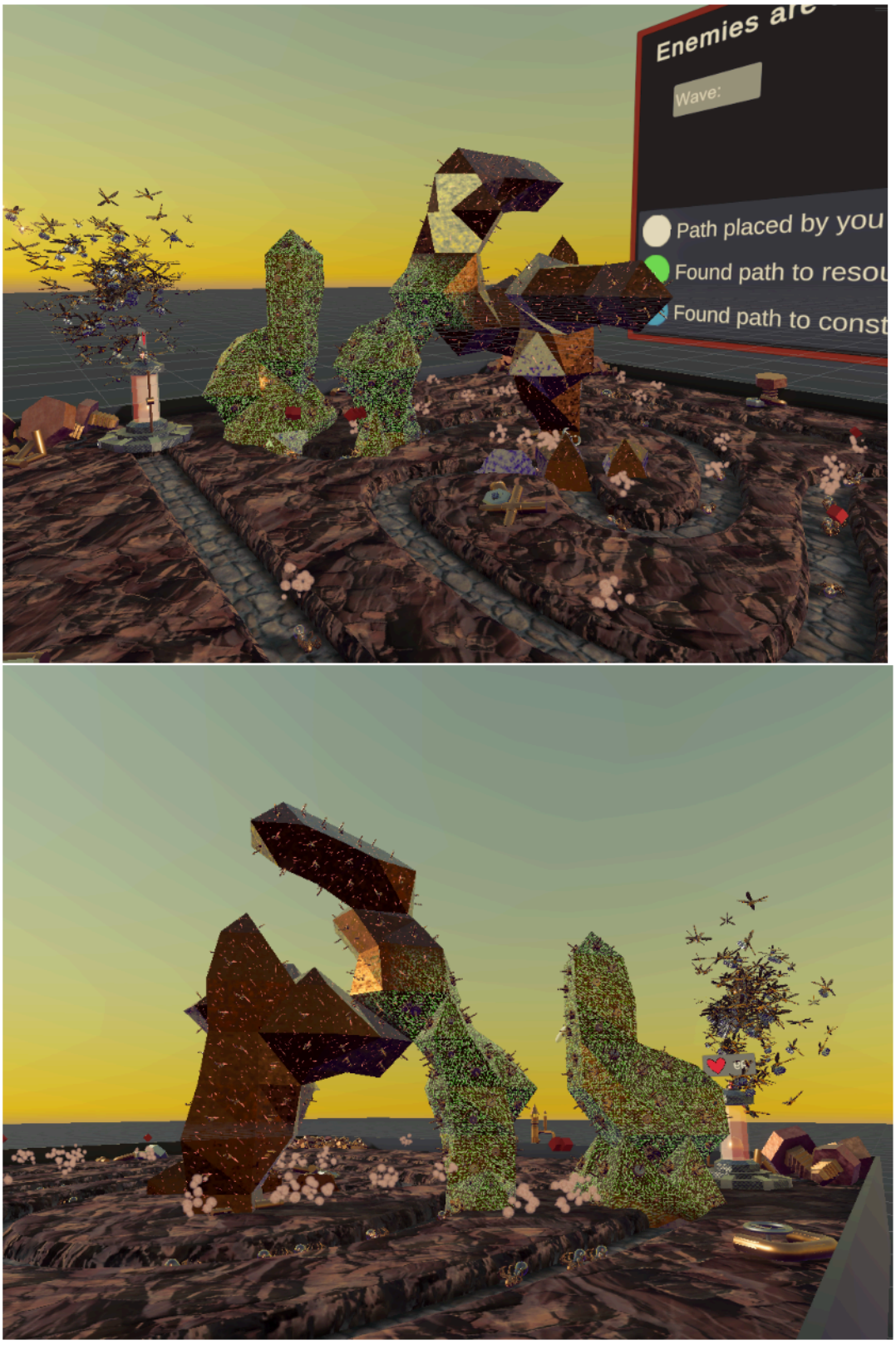



**Figure 4.4.** Towers built to be high and with many overhangs.

**Figure 4.5.** Towers built to be flat and sprawling.

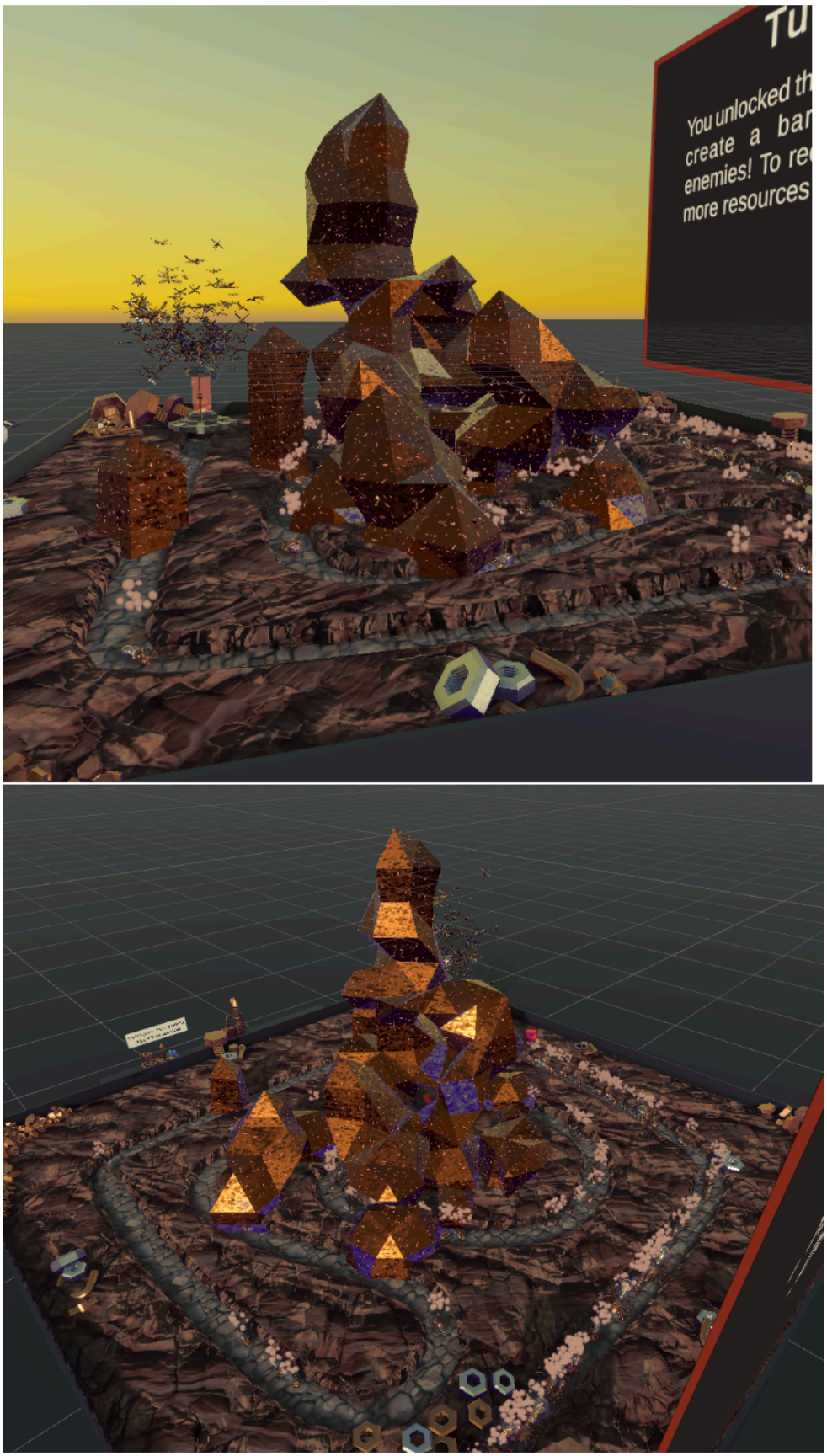



**Figure 4.6.** A big tower structure built by combining tower building techniques used in figure 4.4 and 4.5.

# 5 Discussion

Although the towers meet all necessary constraints, they appear to exhibit a relatively "blocky" appearance, which deviates from the intended organic design aesthetic. While some may argue that the blocky appearance aligns with the mechanical style of other assets, it is clear that the towers were originally envisioned to possess a more organic appearance. A way to achieve this would be to increase the resolution of the grid. The current grid resolution is $32^3 = 32.768$ cubes. While eventually there will be a performance problem, the number of cubes can be increased to $64^3 = 262.144$ without a significant performance decrease. How a tower at that resolution would look can be seen in figure 5.1.

This is not done in the final game for gameplay reasons. By design, the player has limited control over the swarm and therefore, over how the towers are constructed. Increasing the resolution makes it increasingly harder for the player to have any impact on how the towers will look. Keeping the resolution low, on the other hand, allows the player, for example, to intentionally build overhangs by tactically placing the markers.

To summarize: Increasing the resolution reduces the control that the player has. The highest resolution at which the player can still meaningfully shape the towers seems to be $32^3$ cubes. It can be contended that the novel tower system provides a distinct gameplay encounter, effectively integrating tower creation within the pre-existing game framework.

## 5.1 Balancing

To balance the laser and the shock towers, the most important parameters are the increase of power they get from a bigger size, their base damage, their fire rate, and for the shock tower, the area affected by the attack. For the shield tower, it is the time they slow the enemies and the number of resources an enemy consumes

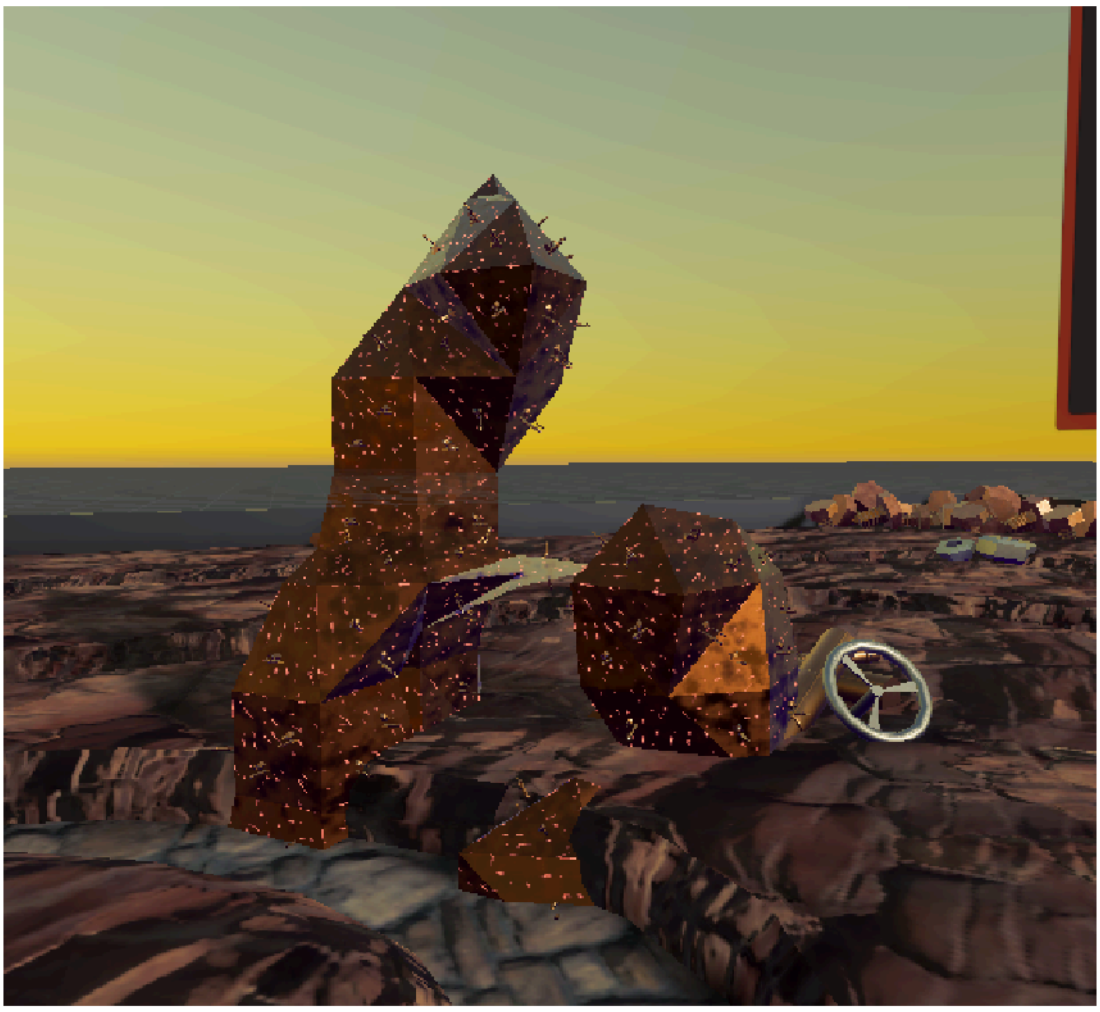


**Figure 5.1.** A tower constructed with a grid resolution of $64^3$ cubes.

when passing through them. Adjusting the number of resources a tower needs to be built is also an important aspect. In the future, these numbers can be tweaked to change how fast the towers grow. A mechanic that was not put in place to simplify the balancing is the decay of towers over time. Implementing this would add another layer of complexity for the player. Since this mechanic would make the game more challenging, it could be an optional difficulty setting. Since shock towers are unlocked at a later stage than the laser towers, they can have better base stats since the enemies are already stronger at this point than they are in the beginning.

## 5.2 Future Work

Since the resolution limitation stems mainly from the game context, it would be interesting to see what towers could be generated in an independent simulation. Swarm agents with different goals could shape interesting structures by building them up. Alternatively, the tower-building process could be inverted, letting agents dig their way through a Marching Cube grid. This could be used to build the underground part of ant hill-like structures. Since the tower-consuming mechanic is already in place for the shield towers, this can easily be done.

Another thing that could be done is to improve the performance of the implemented Marching Cubes. Especially if higher grid resolutions are useful for an independent simulation. When it comes to independent simulations, the resolution should probably be as high as possible. Since such a simulation might not have to be done in real-time or virtual reality, the resolution could be even higher.

When it comes to the game, a couple of aspects could still be improved upon. The marker placement stems largely from the old tower building system and does not allow for a wide range of vertical placement. This could be adjusted in the future, giving the player more freedom. New content could also be added by adding more tower types. So far, the tower types are based on the tower types that the game already had before. Future work could explore which kinds of new towers could be integrated into the game. Since towers can be combined, each new tower type would add an increasing amount of complexity and, therefore, a lot more possibilities for the player to play the game.

# Bibliography


Cabezas, A. F., & Thompson, T. (2013). Real-time procedural terrain generation through swarm behaviours. *FDG*, 421–422 (cit. on p. 8).

Chen, S.-M., & Chien, C.-Y. (2011). Solving the traveling salesman problem based on the genetic simulated annealing ant colony system with particle swarm optimization techniques. *Expert Systems with Applications*, *38*(12), 14439–14450 (cit. on p. 1).

de Andrade, D., Fachada, N., Fernandes, C. M., & Rosa, A. C. (2020). Generative art with swarm landscapes. *Entropy*, *22*(11) (cit. on p. 8).

Du, Y., Li, J., Hou, X., Lu, H., Liu, S. C., Guo, X., Yang, K., & Tang, Q. (2019). Automatic level generation for tower defense games. *2019 IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, 670–676 (cit. on p. 7).

Freiknecht, J., & Effelsberg, W. (2017). A survey on the procedural generation of virtual worlds. *Multimodal Technologies and Interaction*, *1*(4), 27 (cit. on p. 7).

Hendrikx, M., Meijer, S., Van Der Velden, J., & Iosup, A. (2013). Procedural content generation for games: A survey. *ACM Trans. Multimedia Comput. Commun. Appl.*, *9*(1) (cit. on p. 7).

Karaboga, D., & Akay, B. (2009). A survey: Algorithms simulating bee swarm intelligence. *Artificial intelligence review*, *31*, 61–85 (cit. on p. 8).

Kraner, V., Fister jr, I., & Brezočnik, L. (2021). Procedural content generation of custom tower defense game using genetic algorithms. (Cit. on pp. 6, 9).

Liu, S., Chaoran, L., Yue, L., Heng, M., Xiao, H., Yiming, S., Licong, W., Ze, C., Xianghao, G., Hengtong, L., Yu, D., & Qinting, T. (2019). Automatic generation of tower defense levels using pcg. *Proceedings of the 14th International Conference on the Foundations of Digital Games* (cit. on p. 9).

Lorensen, W. E., & Cline, H. E. (1987). Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph.*, *21*(4), 163–169 (cit. on p. 9).

Mark, B., Berechet, T., Mahlmann, T., & Togelius, J. (2015). Procedural generation of 3d caves for games on the gpu. *FDG* (cit. on p. 7).

Newman, T. S., & Yi, H. (2006). A survey of the marching cubes algorithm. *Computers & Graphics*, *30*(5), 854–879 (cit. on p. 9).

Öhman, J. (2020). Procedural generation of tower defense levels. (Cit. on pp. 7 sqq.).

Reynolds, C. W. (1987). Flocks, herds and schools: A distributed behavioral model. *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, 25–34 (cit. on p. 8).

Tan, T. G., Yong, Y. N., Chin, K. O., Teo, J., & Alfred, R. (2013). Automated evaluation for ai controllers in tower defense game using genetic algorithm. *International Multi-Conference on Artificial Intelligence Technology*, 135–146 (cit. on p. 9).

Teuber. (1995). The settlers of catan. (Cit. on p. 7).

Togelius, J., Kastbjerg, E., Schedl, D., & Yannakakis, G. N. (2011). What is procedural content generation? mario on the borderline. *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games* (cit. on p. 7).

## Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Schriften entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Würzburg, March 27, 2023

Lenny Siol

Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

# Versicherung zur Leistungserbringung

Titel der Abschlussarbeit:

Self-Organised Construction of Particle-Based 3D Artifacts

Thema bereitgestellt von (Titel, Vorname, Nachname, Lehrstuhl):

Prof. Dr. Sebastian von Mammen, Chair of Human-Computer Interaction

Eingereicht durch (Vorname, Nachname, Matrikel):

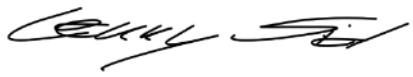
Lenny Siol, 2334375

Ich versichere, dass ich die vorstehende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich keiner anderer als der in den beigefügten Verzeichnissen angegebenen Hilfsmittel bedient habe. Alle Textstellen, die wörtlich oder sinngemäß aus Veröffentlichungen Dritter entnommen wurden, sind als solche kenntlich gemacht. Alle Quellen, die dem World Wide Web entnommen oder in einer digitalen Form verwendet wurden, sind der Arbeit beigefügt.

Weitere Personen waren an der geistigen Leistung der vorliegenden Arbeit nicht beteiligt. Insbesondere habe ich nicht die Hilfe eines Ghostwriters oder einer Ghostwriting-Agentur in Anspruch genommen. Dritte haben von mir weder unmittelbar noch mittelbar Geld oder geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Arbeit stehen.

Der Durchführung einer elektronischen Plagiatsprüfung stimme ich hiermit zu. Die eingereichte elektronische Fassung der Arbeit ist vollständig. Mir ist bewusst, dass nachträgliche Ergänzungen ausgeschlossen sind.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht. Ich bin mir bewusst, dass eine unwahre Erklärung zur Versicherung der selbstständigen Leistungserbringung rechtliche Folgen haben kann.

Würzburg, dem 27.03.2023 

Ort, Datum, Unterschrift