FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Name Surname

Thesis title

Name of the department

Supervisor of the bachelor thesis: Supername Supersurname

Study programme: study programme

Prague YEAR

I declare that I carried out this bachelor thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

Dedication.

Title: Thesis title

Author: Name Surname

Department: Name of the department

Supervisor: Supername Supersurname, department

Abstract: Use the most precise, shortest sentences that state what problem the thesis addresses, how it is approached, pinpoint the exact result achieved, and describe the applications and significance of the results. Highlight anything novel that was discovered or improved by the thesis. Maximum length is 200 words, but try to fit into 120. Abstracts are often used for deciding if a reviewer will be suitable for the thesis; a well-written abstract thus increases the probability of getting a reviewer who will like the thesis.

Keywords: keyword, key phrase

Název práce: Název práce česky

Autor: Name Surname

Katedra: Název katedry česky

Vedoucí bakalářské práce: Supername Supersurname, department

Abstrakt: Abstrakt práce přeložte také do češtiny.

Klíčová slova: klíčová slova, klíčové fráze

Contents

1 Introduction **8**
- 1.1 Tower Defense . . . 8
- 1.2 Roguelike . . . 9
- 1.3 Original Vision . . . 11
- 1.4 Current Scope and Goals . . . 12

2 Game Design **13**
- 2.1 Design goals . . . 13
  - 2.1.1 Strategic Depth in Every Battle . . . 13
  - 2.1.2 Strategic Depth in Every Run . . . 14
  - 2.1.3 Make Various Builds Viable . . . 16
  - 2.1.4 Force Exploration . . . 17
  - 2.1.5 Provide a Challenge . . . 18
- 2.2 Procedural Generation . . . 19
- 2.3 Battle . . . 20
  - 2.3.1 Attacker Waves . . . 20
  - 2.3.2 World . . . 23
  - 2.3.3 Attacker Paths . . . 24
  - 2.3.4 Attacker Types . . . 27
  - 2.3.5 Buildings . . . 28
  - 2.3.6 Towers . . . 29
  - 2.3.7 Abilities . . . 30
  - 2.3.8 Materials and energy . . . 30
  - 2.3.9 Fuel . . . 31
  - 2.3.10 Hull . . . 31
  - 2.3.11 Status Effects . . . 31
  - 2.3.12 Time controls . . . 32
- 2.4 Blueprints . . . 33
  - 2.4.1 Design . . . 34
  - 2.4.2 Augments . . . 35
- 2.5 Battle Graphical User Interface . . . 35
  - 2.5.1 Waves and time controls . . . 36
  - 2.5.2 Fuel and Hull . . . 36
  - 2.5.3 Wave Preview . . . 36
  - 2.5.4 Materials and Energy . . . 36
  - 2.5.5 Blueprint Menu . . . 36
  - 2.5.6 Settings Button . . . 37
  - 2.5.7 Info Panel . . . 37
- 2.6 Attacker HP Indicators . . . 38
- 2.7 Selection and Highlighting . . . 39
- 2.8 Battle Camera Controls . . . 41
- 2.9 Future Features . . . 42
  - 2.9.1 Setting and Story . . . 42
  - 2.9.2 Run Structure . . . 42
  - 2.9.3 Saving the Game . . . 42

2.9.4 Money . . . 43
2.9.5 Permanent Unlocks . . . 43

3 Analysis **44**
3.1 Game Engine . . . 44
3.2 Procedural Generation . . . 44
3.3 Path Generation . . . 45
3.3.1 Hub Position and Path Starts . . . 46
3.3.2 Generating the Main Paths . . . 48
3.3.3 Simulated Annealing . . . 48
3.3.4 Generating Paths using Simulated Annealing . . . 49
3.3.5 Simplifying the Relative Improvement Calculation . . . 53
3.3.6 Final Paths . . . 55
3.4 Terrain Generation . . . 59
3.4.1 Wave Function Collapse . . . 59
3.4.2 Advantages and Disadvantages of WFC . . . 62
3.4.3 Using WFC for Terrain Generation . . . 63
3.5 Obstacle Generation . . . 66
3.5.1 Obstacle Placement Parameters . . . 66
3.5.2 Obstacle Placement Algorithm . . . 67
3.5.3 Generating Obstacle Models . . . 68
3.6 Attacker Wave Generation . . . 71
3.6.1 Model 1: Single Attacker . . . 72
3.6.2 Model 2: Infinite Waves . . . 73
3.6.3 Model 3: Finite Waves . . . 74
3.6.4 Model 4: Damage in an Area . . . 76
3.6.5 Model 5: Multiple Batches . . . 78
3.6.6 Model 6: Multiple Paths . . . 79
3.6.7 Model 7: Abilities . . . 79
3.6.8 Wave Generation Overview . . . 80
3.6.9 Generating Sequential Waves . . . 81
3.6.10 Generating Parallel Waves . . . 83
3.7 Random Number Generators . . . 85
3.8 Battle Simulation and Visuals . . . 87
3.9 Targeting Attackers . . . 88
3.10 Range Visualization . . . 89
3.10.1 Determining The Range Shape . . . 89
3.10.2 Representation for The GPU . . . 90
3.10.3 Computing Everything on the GPU . . . 91
3.11 Blueprints and Info Panel Text . . . 91
3.11.1 Blueprint Representation . . . 91
3.11.2 Info Panel Text . . . 92
3.12 Modifiable Commands and Queries . . . 92
3.12.1 Modifiable Commands . . . 93
3.12.2 Modifiable Queries . . . 93
3.12.3 Event Reaction Chain . . . 94

4 Developer Documentation **95**

5 User Documentation — Designer **96**

6 User Documentation — Player **97**

7 Playtesting **98**

Conclusion **99**

Bibliography **100**

List of Figures **103**

List of Tables **105**

List of Abbreviations **106**

A Attachments **107**
A.1 First Attachment 107

1 Introduction

Video games are a popular form of entertainment. There is a plethora of games to choose from, each offering a different experience. Still, it is always possible to create something new that players might enjoy. The author of this thesis enjoys both *tower defense* games and *roguelike* games and there are not many games that combine these two genres. In this thesis, we will design and implement a video game, that uniquely blends them, and discuss the decisions behind it. So, what do we mean by a *roguelike tower defense* game?

1.1 Tower Defense

A game genre can encompass many characteristics, most often its mechanics, but also its theme, art style or the medium it is played on. Genres have no exact definitions or strict boundaries, they are characterized by how people use them to describe games.

Tower defense is often described [1][2] as a subgenre of *real-time strategy*. This means the game focuses on long-term planning, but also quick thinking. In *tower defense* games, the player has to defend against waves of attackers by building defensive towers. As an example we'll look at *Plants vs. Zombies* [3], released in 2009.

In *Plants vs. Zombies*, the player defends their house from zombies. As shown figure 1.1, the zombies come from the right side of the screen and advance left. If any zombie reaches the far left edge of the screen, the player loses the level. The goal of each level is to survive all the incoming waves by placing plants that kill or otherwise impede the zombies. We can also see two *Repeaters* in the upper left part of the image, one of them shooting at a zombie. Further to the left, there are a lot of *Sunflowers*. These are a very important part of the game, because all plants cost *sun*, and *Sunflowers* produce those.

Figure 1.1 A level in *Plants vs. Zombies*.

In our game, the player will also build towers, to defend from waves of attackers, and economic buildings that produce materials. Though, it will differ a lot from *Plants vs. Zombies* in the overall structure of the game. The main game mode of *Plants vs. Zombies* is a campaign consisting of 50 individual levels. If the player loses a level, they can try again and again until they succeed in beating it. After most levels, the player unlocks a new plant, which they can use in upcoming levels, slowly building up their arsenal. In our game, however, once the player loses, they lose all their progress and must start from the very beginning. This and some other mechanics are taken directly from the *roguelike* genre.

1.2 Roguelike

Roguelike is a subgenre of *role-playing games*. In *role-playing games*, the player takes on the role of a character and goes on an adventure. The character can grow stronger by acquiring new abilities, items, or experiences. The player has to make decisions about how to upgrade their characters to overcome the challenges they might face. *Role-playing games* are a very broad genre with a long history, for more information we recommend the book *Game Design Deep Dive: Role Playing Games* [4] by J. Bycer.

The *roguelike* genre is named after the game *Rogue* [5], released in 1980. In this single-player turn-based game, the player explores a grid-based dungeon and fights monsters that inhabit it. Along the way, they collect various weapons, armor and other magical items that improve their abilities. It features a mechanic nicknamed *permadeath*, which means that when the character dies, the player loses all progress and must start from the very beginning. The dungeon is randomized — it is different in every run, so the player can't just memorize the layout. These are the most defining features of *roguelikes*, but games of this genre aren't just clones of the original *Rogue*. The breadth of *roguelike* games is well explored and explained by J. Bycer [6].

A more recent game that's a good example of this genre is *Slay the Spire* [7]. In *Slay the Spire*, the player ascends a spire and fights various enemies. The fights are also turn-based, and when the player's character dies, they have to start from scratch. However, it is not a traditional *roguelike*. The game is not played on a grid, instead the spire the player navigates is a graph of separate rooms, where they move from bottom up. We can see this in figure 1.2. Here, the player has been to the rooms that are circled, and now they have to choose where to go next. The player can come across different kinds of rooms, each represented with an icon. The most important are enemy encounters, where the player fights monsters using a deck of cards.

Figure 1.2 The map screen in *Slay the Spire*.

In figure 1.3, the player character is shown on the left, facing a *Jaw worm* on the right of the screen. On the bottom, there are cards that the player can play to fight the enemy. At the start of each turn, the player draws five cards from the deck. We can see that each card has a name at the top with its corresponding illustration below. Below the illustration is text explaining the effect of the card when played. Most cards deal damage to the enemies or provide *block* to defend from enemy attacks, but some have more unique effects. In the top right corner of a card is displayed its *energy* cost. The player can only spend three *energy* per turn, so they can only play a limited amount of the cards they drew. It is important to play the right cards in order to kill the enemy without taking a lot of damage.

Figure 1.3 A fight in *Slay the Spire*.

Even though the player never knows exactly what cards they'll draw, they can shape the deck they draw from throughout the game. The player starts each run with a predefined deck of starter cards, and as they progress, they add new cards into their deck. For example, after every fight, they get presented with three randomly selected cards, and they can choose one of them. The player can

also get new cards from events or shops and sometimes remove the cards they don't want. Some cards are rarer than others, and they are often more powerful. However, being lucky and getting the most powerful cards is not what the game's about. The player must learn which cards work together well and which don't, and understand the weaknesses of their deck and how to fix them.

Many games take the *roguelike* mechanic of *permadeath* and randomized procedural generation, but fill in different game mechanics. *Slay the Spire* has the player build their own deck of cards to play with, but they still play as a character that fights enemies. Some, however, deviate much more. In our game, the battles will be in the style of *tower defense*, and the player will collect blueprints for defensive towers and other buildings instead of weapons and armor.

Games that deviate more from the *roguelike* formula are sometimes called *roguelite* games. However, there is no agreement on when a game stops being *roguelike* and starts being *roguelite*. We will not make this distinction, since game genres have no precise boundaries and can be freely blended with others.

1.3 Original Vision

Now that we have introduced the concepts of *tower defense* and *roguelike* games, we can use them to create an overview of the game we intend to make. It will be a single-player game. As stated, the moment to moment gameplay will be a *tower defense*, but on a larger scale, the game will be *roguelike*. This means that it will consist of individual procedurally generated runs, where the player will start from scratch every time. During each run, the player will defend against attackers in many battles and improve their arsenal to grow stronger. Their goal is to get as far as possible, trying to reach the final level and beat the game.

Battles
The goal of each battle is to gather enough *fuel* to continue. The faster the player gathers the *fuel*, the sooner they win the battle. The *fuel* is generated passively, but additional buildings can be built to speed up the process. In the meantime, the player has to defend against waves of attackers by building towers and using abilities. Towers persist throughout the battle and shoot at the attackers, whereas abilities will provide single-use effects that can help in a time of need. All of this costs *materials* and *energy* — resources, which are generated by economic buildings.

Procedural generation
Each battle will take place on a unique, procedurally generated terrain. This means that the paths the attackers take will also differ in each battle. Furthermore, there will be various kinds of attackers and the attacker waves will also be procedurally generated.

Blueprints
On their way, the player will choose from randomly selected *blueprints* to add to their collection. These *blueprints* will allow them to use new abilities, or build

new towers and other buildings. The player will have to choose *blueprints* which work together well in order to use their full potential.

Run progression
The player will also encounter various shops and events. These can present additional choices and provide the player with opportunities to gain various rewards or punishments. The path the player takes will not be linear, allowing them to decide which battles to fight and what to interact with from the map screen.

Platform
We will target the game for personal computers only. Unlike mobile phones, PCs usually have a screen large enough to let us clearly convey all the information the player needs. It won't be for game consoles either because we think a mouse will be the best way to control the game. The mouse allows the player to select a precise position in the world quickly. The player can also control certain aspects of the game using the keyboard.

1.4 Current Scope and Goals

The scale of the game as outlined in section 1.3 is quite large. Furthermore, it would need a lot of content and polish before being able to be released as a full game. Instead of creating a full-featured polished game, in this thesis we will focus on making a functional demo version, which can be used to playtest the core gameplay. The demo will contain some base content in order to be playable, and it must be prepared for future development so that more content can be added later.

The demo version will allow the player to progress through battles and collect blueprints. However, there will be no map screen to let them choose their path as described in the paragraph Run progression of the previous section. For now, the progression will be linear and there will be no events or shops, only battles. All the art and sound assets will be placeholders, but care will be taken to make everything as clear as possible to the player.

The main goals of the thesis are:

1. Design the game's mechanics and features.
2. Implement all the systems and mechanics described in paragraphs Battles, Procedural generation and Blueprints.
3. Include a tutorial to explain the game's mechanics to the player.
4. Run a playtest.

2 Game Design

Before we start implementing the game, we should design its individual parts. An overall design was described in section 1.3. In this chapter, we will go into more detail and flesh out the design. We need to decide which mechanics will be in the game and how will the player interact with them. The game needs to react to the player's actions and communicate the information the player should know. This all depends on what exactly are we trying to achieve. Thus, we will start by setting some design goals.

We would also like to emphasize that some features will not be implemented in the demo version of our game. These features will be marked by the following box:

(not implemented in the demo)

A feature that won't be in the demo.

2.1 Design goals

We aim to make the game's mechanics clear, and controls intuitive and responsive. This is a necessity for every game because without this, the players can't even properly play the game we want them to play. This is an important goal that will inform many of our decisions throughout the design.

We have analyzed several games of similar genres to our game, that we find enjoyable, and we tried to identify what makes them fun. We identified five features, which we think make the games very intriguing and replayable, and we think these would work for our game too. Thus, we intend to design the game, so it exhibits these features, making them our game-specific design goals. We will explain each in a separate section, and we will use other games as inspiration for how to reach them. The goals are:

1. Strategic Depth in Every Battle
2. Strategic Depth in Every Run
3. Make Various Builds Viable
4. Force Exploration
5. Provide a Challenge

2.1.1 Strategic Depth in Every Battle

One of the design goals we identified is that the game should let the player make meaningful strategic decisions throughout every battle. Each battle should be different enough to require the player to adapt to the current situation. This is where the action will happen, but we want the player to make tactical decisions, not test their reflexes. With this constraint, battles would be boring if every one played out the same.

In *Plants vs. Zombies*, the player wants to plant *Sunflowers* or other *sun*-producing plants. The more they build their economy, the more plants they can afford in the future. However, these plants can't kill zombies, so the goal is to spend the bare minimum on defense. This is a hard problem to solve, since when and where zombies will appear is not completely predictable. What makes this even more complicated are cheap single-use plants like the *Potato Mine*. It costs only 25 *sun* and can kill almost any zombie, where, for example, a *Peashooter* costs 100 *sun*, but is permanent and able to kill many zombies over the course of a level. This means the player always has to consider if it's better to place a plant that's the best now or a plant that will be the best in the future.

In *Slay the Spire*, the player has to make a similar decision, but even more often. Almost every enemy grows stronger over time, or makes the player character weaker as they fight. This means that the player always has to consider when it's the best to defend and when it's better to attack. The player can choose to not block some damage now in order to kill the enemy sooner and prevent bigger attacks in the future. The player also has to plan several turns in advance because many cards have longer lasting effects. They often have to decide whether it's better to play a card that makes them stronger in future turns, or a card that helps them now.

Every fight is different because every enemy has distinctive behavior. Some enemies get much more powerful over time, so it is important to kill them quickly. Others punish the player for attacking them, so the player needs to kill them with precision. Fights also vary a lot because the player draws their cards in a different order every time. All this means that the player has something to think about every turn.

Our game will also have economic buildings and instant abilities, so the player has to balance economy and short-term versus long-term defense. The player will have to survive some number of waves, but they will be able to spend extra *materials* to mine *fuel* faster and end the battle sooner. This is similar to being more offensive in *Slay the Spire*, since the waves of attackers should get stronger at a faster pace than the player's defense. Each battle will require a different approach, since the waves will be composed of a different set of attackers every time. We can also vary the nature of a battle by changing up the terrain and making attacker paths different lengths or more numerous. This might seem like too much, but we want to playtest all these options and possibly cut those, which don't work well.

2.1.2 Strategic Depth in Every Run

Another of the design goals is that our game should let the player make meaningful strategic decisions throughout every run and there should be no clear path to victory. In our game, when the player makes a decision when fighting in a battle, its consequences should be contained mostly within the battle. This goal refers to the decisions the player will make outside a battle, which affect all future battles.

In *Slay the Spire*, the player needs to improve many aspects of their deck in tandem. They need to have great defensive cards, cards that can deal with enemies that have a lot of health, cards that can attack multiple enemies at once

and more. The player should also care about the average cost of the cards in their deck. It is bad when the player wants to both defend and attack on a given turn, but they've drawn only an expensive attack and an expensive defensive card. It is also suboptimal when the player plays out all the cards they've drawn, but they have leftover *energy* they didn't spend. Balancing these aspects of the deck leads to some difficult decisions when picking cards to add. For example, should the player pick a good defensive card because they are lacking in defense, or should they pick an attack that's just very strong.

We want to balance the battles in a way, which requires the player to have strong blueprints (see section 2.4) with various qualities. The players should need good economic buildings, *fuel*-producing buildings, abilities and towers good at dealing with various kinds of attackers. They should also have some cheaper towers to build in the first few waves and more expensive towers to build once they produce a lot of *materials*.

In *Slay the Spire*, the player comes across the interesting trade-off between short-term and long-term power even in building their deck. The player wants cards which will have a great potential to be strong in the future, having great synergy with other cards. But these cards aren't strong right now and the player needs to survive the next few fights, making them choose cards that are useful immediately, but might not be as powerful later in the run. As an example we can look at the cards *Iron Wave* and *Double Tap*.

The player starts each run with several copies of cards *Defend* and *Strike* in their deck. Compared to them, *Iron Wave* is a very cost-efficient card. As shown in figure 2.1, it costs 1 *energy* (displayed in the top right corner of the card), the same as *Defend* or *Strike*. However, it does almost the same thing as *Defend* **and** *Strike* combined — it deals damage and gives *block* too. Picking this card can help a lot in the early fights, but it doesn't really grow stronger later in the run. The card *Double Tap*, on the other hand, is not great at the start. In essence, it acts like another *Strike* most of the time, and is useful only when the player draws another attack alongside it. It is however very strong when the deck contains many attacks that cost a lot of *energy* but deal much more damage. Then it allows the player to play a powerful attack twice at the cost of only one more *energy*.

Figure 2.1 *Defend*, *Strike*, *Iron Wave* and *Double Tap* cards from *Slay the Spire*.

We can design the blueprints in our game similarly, making some useful early in the run and some powerful later. This will let the player decide if they need to take a blueprint that will help them now, or a blueprint that can potentially be strong later.

2.1.3 Make Various Builds Viable

One of the goals of our game is that the player should be able to beat the game with a lot of different combinations of blueprints. We will call these combinations *builds*, as is often done [8] for unique combinations of skills, attributes and items a player's character can have in a role-playing game. Builds are distinguished mainly by what they feel like to play with. If two blueprints are used in the same way, then exchanging one for the other doesn't make a new build. To allow the player to choose from various builds, there has to be enough blueprints that feel distinct and better yet, they should interact with other blueprints in unique ways.

In figure 2.2 are shown all the plants from *Plants vs. Zombies*. As we can see, there is a lot of them, and various combinations that work well are possible. The plants usually don't interact with each other strongly, so the player mostly has to combine the plants such that they have no weak spots. For example, longer levels require both cheap and expensive defensive plants. The cheap plants are used at the start of the level, and later they are replaced by the more expensive ones to fit more firepower on the limited lawn. Some plants can struggle against certain zombie types, so the player also wants to choose plants to cover for all their weaknesses.

Figure 2.2 All the plants of *Plants vs. Zombies* in the in-game almanac.

We can also look at a few examples from *Slay the Spire*. Here, builds are often defined by cards that interact in ways that make them stronger. One of the most blatant examples are cards that apply *poison* to the enemy. A poisoned enemy takes damage every turn based on the amount of *poison* they have, and the amount decreases by one every turn. This means an enemy with 2 *poison* takes $2+1=3$ damage in total, whereas an enemy with 4 *poison* takes $4+3+2+1=10$ damage in total. It's easy to see that every card that applies *poison* makes other *poison* cards stronger.

There are also rare cards the player can find, which change how the game works. For example, defensive cards provide *block* only for one turn because the

player character loses all *block* at the start of every turn. However, once the player plays the card *Barricade*, they don't lose block at the start of their turns for the rest of the fight. Cards like this can determine the player's strategy for the rest of the game on their own.

We want the players of our game to try lots of different builds and for that, the builds need to be strong enough to beat the game when the player executes them well. We can tweak the strength of individual blueprints, but we can also design enemies that punish specific builds that would otherwise be too good. For example, in *Slay the Spire*, many enemies shuffle unplayable cards into the players deck for the duration of the fight. This punishes decks with fewer cards way more than decks with many cards, keeping small deck builds from being too powerful.

2.1.4 Force Exploration

We don't want the player to just find a single build that works and never explore anything new. When the player is familiar with a build, it becomes stronger, since they know how to use it effectively. This discourages them from trying other builds, because they can't use them so well, making them weaker. Thus, one of our goals is to force the player to explore and make them learn other strategies.

The main way to get cards in *Slay the Spire* are the rewards after every battle, where the player can choose one of three cards to add to their deck, as shown in figure 2.3. All the ways to acquire cards are randomized, so the player can't just hope to always get the card they want. They have to adapt their build to the cards on offer, so they have to explore different strategies in order to win consistently. In our game, the player will also pick a blueprint to add to their collection from a randomized offer after each battle.

Figure 2.3 Card reward screen in *Slay the Spire*.

In *Plants vs. Zombies*, the player has to adapt to different zombies and level environments. This can be illustrated with figure 2.4, which shows a seed select screen. Here, the player selects which plants they want to use in this level from the selection on the left side. On the right the player can see that this level takes place on the roof and the zombie types that will appear in this level. In rooftop levels, the player has to place a *Flower Pot*, which costs 25 *sun*, on a tile before

they can place a plant there. Furthermore, all plants that shoot in a straight line are of little use here because the roof slopes up, so their projectiles can't travel very far. An experienced player will also notice that *Bungee Zombies* will appear. These zombies swing from above to take the player's plants instead of coming from the right. The player should consider all these factors when choosing the build to play this level with.

Figure 2.4 Seed select screen in a rooftop level in *Plants vs. Zombies*.

In our game, the player could select which blueprints to play with before every battle based on the level's features and attackers. Instead, we chose an approach more similar to *Slay the Spire* — the player will keep the blueprints they collect for the rest of the run, and they won't know the specifics of a battle before selecting it. However, they will be allowed to have only a limited amount of blueprints at once, so they still cannot just keep all the blueprints they encounter.

2.1.5 Provide a Challenge

The player should always have some goal to work towards, just out of their reach. If the game is too easy, the players will have no reason to think strategically or learn. Always having a harder challenge to overcome will motivate the player to improve and keep playing.

Slay the Spire is not easy to beat, but the player can still improve so much even after beating the game. After beating the game, the player unlocks so-called *ascension*. Before embarking on another run, the player can select the *ascension* level they want to play on. Each level introduces a small change that makes the game slightly more difficult. Each *ascension* level is unlocked only after the previous level is beaten, and each difficulty increase is small, so it doesn't discourage the player. These changes are cumulative, so in the end it takes serious effort and luck to beat the game on *ascension* level 20 even for the most skilled players.

(not implemented in the demo)

This system is simple, yet effective, so we might as well use it too. We will also want to balance the base game, so that most players that try are able to beat it, but it still takes some effort and several attempts, so the players feel like they've accomplished something.

2.2 Procedural Generation

Randomized procedural generation is one of the defining features of the roguelike genre. We want to use randomized procedural generation to make each run of the game unique. Since we design the procedural generation algorithms ourselves, we have great control over the results. However, procedurally generated parts of the game can be really hard to balance. We need to make sure the randomized parts of the game feel fair to the player. It doesn't feel good if the player loses the game because they were just unlucky and couldn't have done anything to prevent the loss. Another issue with randomized procedural generation is that things might start to feel very homogenous. For example, hand-crafted levels can have features that really stand out. We need to decide which parts of our game will be procedurally generated and to what degree.

(not implemented in the demo)

The overall structure of each run will be decided by what we call the *map*. As stated before, it will be a graph, and the player will go from node to node along the edges. Each node will be a battle, shop or an event. We really want each run to be different enough, that the player doesn't develop a single strategy to use in every run. Since the player will decide where to go, it's not a problem when some paths through the map are more difficult than others.

Every battle will also be randomized to a large degree. The world a battle takes place on will be procedurally generated, making for a different environment every time. However, the combination of features that can appear in a given world will be decided by the world's *terrain type*.

(not implemented in the demo)

There will be several hand-crafted terrain types to randomly select from, some appearing only early in the run and some only later. This is to create several cohesive styles of the worlds that look and play distinctly from each other. We feel this is better than if we just let the world generator mix all the features every time, because the results would be more homogenous. We will go into more detail about the world generation and these features in section 2.3.2.

In *Slay the Spire*, each encounter is chosen from a pool of hand-picked enemy combinations. Each of these pools contains encounters of similar difficulty. If the authors of *Slay the Spire* decide one of the encounters is too difficult for its pool, they can tweak the encounter to make it less difficult, or move it to a different encounter pool.

In our game, however, the attacker waves in each level will also be procedurally generated. We want this, because each level in our game will consist of many waves, each with many attackers. We could design many sets of waves, but we feel that would make the levels too predictable, once a player learns these sets. So, the waves shouldn't be tied to the previous waves in a level. Each wave in a level will be harder and harder, so this would mean we would have to populate tens of pools with hand-crafted waves, which feels very inefficient.

From the player's perspective, all procedural generation and the rewards they receive will be random and unpredictable. However, each run will have a single seed that deterministically decides all the "random decisions" the game makes. Two runs with the same seed should look identical and if the player makes the same decisions and choices, the outcomes should be the same. This allows the players to share seeds of the runs they found interesting and compare their skill in the same situations. Furthermore, this is helpful for debugging, because it lets us easily reproduce any issue with the generation just by running it with the same seed.

We could also procedurally generate the blueprints and attacker types. However, here we want to have greater control, because that will allow us to create designs that have powerful and unique abilities. Procedurally generating these would be very difficult, and it would often lead to abilities that are either uninteresting, or way too powerful.

2.3 Battle

As stated in section 1.3, in our game, the player will fight in battles throughout each run, and these battles will have tower defense gameplay. In this section, we will describe the battles in more detail and explain our intentions.

2.3.1 Attacker Waves

The attackers in various tower defense games often come in waves. However, in *Plants vs. Zombies*, the zombies also come in continuously throughout a level in addition to the large waves, to keep the pressure up. Even in games where attackers come in distinct waves, the waves are usually on a timer and once the level starts, they keep coming. One example of such a game is *Kingdom Rush* [9]. In figure 2.5 is shown the indicator which shows the time remaining to the next wave. This means the game is also full of action and requires the player to think quickly. Furthermore, this indicator lets the player call the next wave early. If they do, they get some coins as a reward, but this is risky, because the player's defense might get overwhelmed.

Figure 2.5 Next wave indicator from *Kingdom Rush*.

However, we want to emphasize the long-term strategy, so we will give the player plenty of time to plan out their next move. There won't be any timer, instead, they can start the next wave when they are ready. This is also common in tower defense games, used for example by *Bloons TD 6* [10]. This brings our game closer to the turn-based gameplay that is often featured in roguelike games. First it is the player's turn to build towers, and then the attackers' turn.

There are also many ways the attackers can move in different tower defense games. Most often, the attacker paths are predetermined, and the player builds their towers around them. The attackers go from the start of the path and try to reach the end of the path. This is especially great when there is multiple different levels in the game, each featuring different paths, because it makes different towers more useful than others in each level. In figure 2.6 are shown two levels with distinct paths from *Bloons TD 6*. The path in the first level shown has a lot of tight turns, perfect for close-range towers or towers which damage all attackers in an area. In the second level, the path is made up of few long straight segments, where are much more useful towers that pierce through many attackers in a straight line. Since we want to have various procedurally generated levels in our game, we will also have attackers come on predefined paths that will be different in each level.

Figure 2.6 The levels *Park Path* and *Another Brick* from *Bloons TD 6* with the attacker paths highlighted.

There are other options used in other games. In *Desktop Tower Defense* [11], for example, the attackers try to cross a rectangular playing field. It starts out empty, but as the player fills it with towers, the attackers have to adjust their path, because they cannot go through the towers. In figure 2.7, we can see the purple attackers funnel into a narrow passage between the white towers. Since the player decides the path of the attackers, they have to learn what kind of path works well, but then they can build it all the time. This is not ideal for us, because we want the player to adapt to the environment, not the other way around.

Figure 2.7 Attackers being funneled between towers in *Desktop Tower Defense*.

In *Plants vs. Zombies*, the zombies come from the right side of the screen and try to reach the left side, as we already mentioned. The plants are planted directly in the way of the zombies and the zombies have to eat their way through them to reach their goal. This is unique, and it greatly changes the gameplay. However, this is again not great for our game, because we would lose a lot of potential for the levels in our game to be distinct from each other.

In *Bloons TD 6*, the player receives very little information about what the upcoming waves look like. Here, the player selects the level they want to play on, but the same sequence of waves comes every time, so the player is expected to learn at least those waves that give them problems. In our game, however, the waves will be procedurally generated. We want the player to plan around the upcoming waves, so we need to communicate what the upcoming waves are going to be. This means that the waves should be simple enough to communicate effectively. *Desktop Tower Defense* features a wave preview, shown in figure 2.8, that only describes the type of attacker that will come. We want interesting behavior to emerge from the interaction of different attacker types, so we won't limit our waves to one attacker type, but instead three. We feel that any more would make the waves messy and unnecessarily hard to communicate.

Figure 2.8 Wave preview from *Desktop Tower Defense*.

In fact, each wave will be composed of one to three *batches*. Each batch will be composed of a number of attackers of only one type, spaced evenly. But some waves will be just one batch, but this batch will send a different attacker type on each path on levels with multiple paths. Also, some waves won't spawn attackers on all paths. This should provide enough variety without being too hard to communicate to the player and too hard for a skilled player to predict the wave difficulty. To make it simple both for the player and for us when displaying the preview of the wave, the spacing between two batches will always be 1 second.

The waves in a single battle will get progressively harder, forcing the player will to improve their defense. However, the wave difficulty should increase faster than the player's defense is expected to improve. This increase will need to be carefully balanced to allow for some strategies where the player invests more into *fuel* production to end a battle quickly, but also strategies where the player invests heavily into defense to keep up with the later waves.

2.3.2 World

In some tower defense games, for example in *Desktop Tower Defense*, the towers can only be placed in positions on a grid. In other games, for example *Bloons TD 6*, the towers can be positioned freely, as long as they don't collide with each other, the attacker paths, or other obstacles. While the second option might allow for more interesting tower placement, we will go with grid placement, and the grid will be pretty coarse — only 15×15 tiles. In fact, the attacker paths will also be restricted to the grid. They will be formed by segments, each going from the center of one tile to the center of a neighboring tile. This is because we want the experience a player gains in one level to be transferrable to another level. For example, they might learn that "tower A" placed right next to a straight path can handle a wave of five "attackers B" on its own. They will then know this is true in any level whenever there is a sufficiently long straight path. Reducing the number of path shape and tower position combinations will make the player come across a combination they already know more often, letting them predict better if their defense can handle a wave or not. This is a really important skill to learn, because the player will have to decide before every wave, if they need to invest into defense or if they can invest into their economy.

In some tower defense games, for example in *Kingdom Rush*, there are only few places where the player can place a tower in each level. We feel this is too restrictive for our game, and it would take too much freedom away from the player. This option also really works only in hand-crafted levels, because it is important to select the places for the towers in a way that makes for fun and interesting levels.

However, each level being just a big square of tiles with rectilinear paths on top wouldn't be very interesting. That's why some tiles will contain obstacles that block the player from building on these tiles. Some obstacles will be *small* and some will be *large* — they will also block the line-of-sight of towers that require a straight line between them and the attacker they want to shoot. Some small obstacles will make the tile rich in *minerals* (see section 2.3.8) or *fuel* (see 2.3.9). These will be used by some economic buildings (see 2.3.5), and also act as *small* obstacles. There number of tiles with resource obstacles should always be between some minimum and maximum value, so the levels aren't unfair.

The tiles are pretty big, so when a tile has an obstacle, it's usually not just one obstacle, but a whole cluster of them. For example, a tile won't be blocked by one rock, but rather a cluster of rocks. These clusters will also be procedurally generated. If we used a small set of hand-made models, the repetition would be very obvious, and making a lot of these models by hand seems unnecessary.

Another great way to make the levels more interesting, that is also intuitive for the player, is having tiles at different heights. The heights will be in multiples

of 0.5 units, where one unit is the edge length of a tile. Towers that require line of sight won't be able to shoot over higher terrain or down from steep cliffs. We can also make some tower unable to shoot uphill or downhill for more variety. Some tiles will also be slanted, gradually going from one height to another. These tiles will allow the attacker paths to change their height, because it would be weird if the attackers had to jump up a cliff. Some buildings will be possible to build on slanted tiles and some won't, making slants also a kind of obstacle.

As we already mentioned, each level will randomly select one of multiple *terrain types*. A terrain type will dictate how a terrain should look — the colors used, which terrain feature will appear and how often, and which obstacles will appear. Some obstacles will appear in clusters or near another obstacle, others will be spread out or avoid another obstacle. Each terrain type will have its distinct look, keeping the levels from being all the same.

To summarize:

- **RW1** The world each level takes place on will be a grid of 15×15 square tiles.
- **RW2** There will be *small* or *large* obstacles on some tiles, large obstacles blocking certain towers' line of sight. Some *small* obstacles make the tile rich in *minerals* or *fuel*.
- **RW3** Each tile with an obstacle will usually contain a whole procedurally generated cluster of obstacle models.
- **RW4** The tiles will be at different heights in multiples of 0.5 units, and some tiles will be slanted, going between two heights.
- **RW5** The player can build one building per tile, and only if the tile doesn't contain an obstacle or the attacker path.
- **RW6** Each world will be generated according to a randomly selected terrain type, which determines what the world will be like.

2.3.3 Attacker Paths

In section 2.3.1 we decided that the attackers will travel on predefined paths generated with the world. If we designed each level of our game by hand, we could create paths that just feel like they would be fun to play around. Since the paths will also be procedurally generated, we need to describe what qualities should the paths have, so the generation can later be implemented to produce such paths.

In the previous section 2.3.2 we decided that the world will consist of a grid of tiles and the paths will be constrained to straight segments between the centers of the tiles. We can think of the paths segments as one-way passages between neighboring tiles. This means that a path cannot go twice through the same tile or cross itself, because a tile has the same path segments coming from it, no matter if it was visited for the first or second time.

There can be multiple paths in a level, each with a different shape and in some waves a different set of attackers. This will add more variety and depth to tower placement. Any path will also be able to split into more paths, or join together

with another path, creating new path geometry or sections with different attacker density. When a line of attackers comes to a split into multiple paths, they will alternate in which path they continue to, splitting between the paths evenly.

The player will start each level with one building already built — the *Hub*. It is the goal the attackers are trying to reach to destroy it. Hence, all attacker paths will converge to the tile the Hub is on. The attackers will come from outside the world the battle takes place on. In the game's universe, the worlds are bigger, but the playable area is just a small neighborhood around the Hub. It would be weird if the attackers just appeared on the edge tiles, so their paths will start on tiles just outside the playable world, and the first path segment goes from the path start to the nearest tile in the playable world.

In figure 2.9 we can see an example of a valid path network drawn in blue on a world of square tiles. The black point represents the Hub.

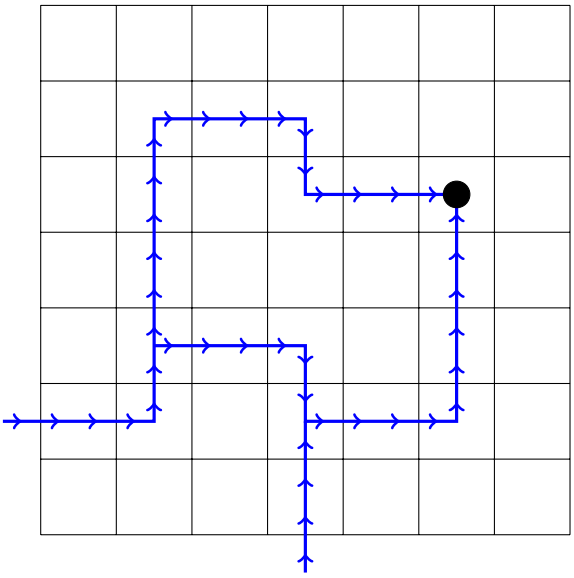


Figure 2.9 An example of a valid path network in a 7×7 game world.

The attackers from single wave batch will start out evenly spaced. If the path they are on splits into two, they will still be evenly spaced, but now the spacing is twice as large. This is illustrated in figure 2.10, where the attackers are represented by black dots on the path. We can also see, that after the paths join back into one, the attacker spacing is no longer even. We don't want this to happen for aesthetic reasons, but also because overlapping attackers could be hard to identify or distinguish by the player. This only happens when the branches of a path are of unequal length and the difference is not a multiple of the spacing between the attackers. We don't want to put more constraints on attacker spacing, so instead, we will constrain path branches to be of equal length. More precisely, each tile on a path has to be the same distance from the Hub, no matter which path an attacker would take.

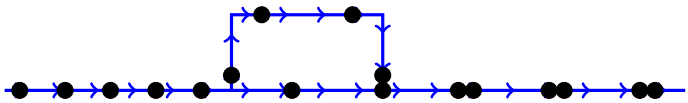


Figure 2.10 Attackers on a path that splits and joins.

Most towers will have limited range, and they will be most effective near the attacker paths. We want the paths to be spread out throughout the game world in order to not have tiles that are just way too far from any paths to be useful. This is illustrated in figure 2.11, where we can see a path network with bad features on the left, and on the right, one with the same path lengths and starting positions, but nicer and more spread out. We have marked tiles whose center is 2 or more tiles from the nearest path with a small cross.

In the right figure, we can also see a red point on one tile, marking a great spot for a tower. This spot allows even a tower with shorter range, illustrated by the red ring around it, to target attackers on a large portion of the path. On the left we have circled another U-turn in the path that is, however, undesirable. This is because there is no empty tile between the paths, so the player can't place a tower there, which feels bad. We don't want many sharp turns like these, but they can occur from time to time for variety. Similarly, paths going right next to each other (marked in red) are bad. The player cannot place towers on paths, so these paths greatly limit the player's access to each other, making for an unpleasant experience. A similar situation occurs when a path goes through tiles at the edge of the world, blocking access to the path from one side, so paths should not go through these tiles very often.

The Hub should be several tiles away from the edges of the playable area, so the player has good access to the path segments near it. It should be very close to the center in levels with many paths, so the paths can come towards it from different sides of the world. It can be more off-center in levels with only one or two long paths, where the path can snake through the world from the side furthest away from the Hub, also covering the world somewhat evenly.

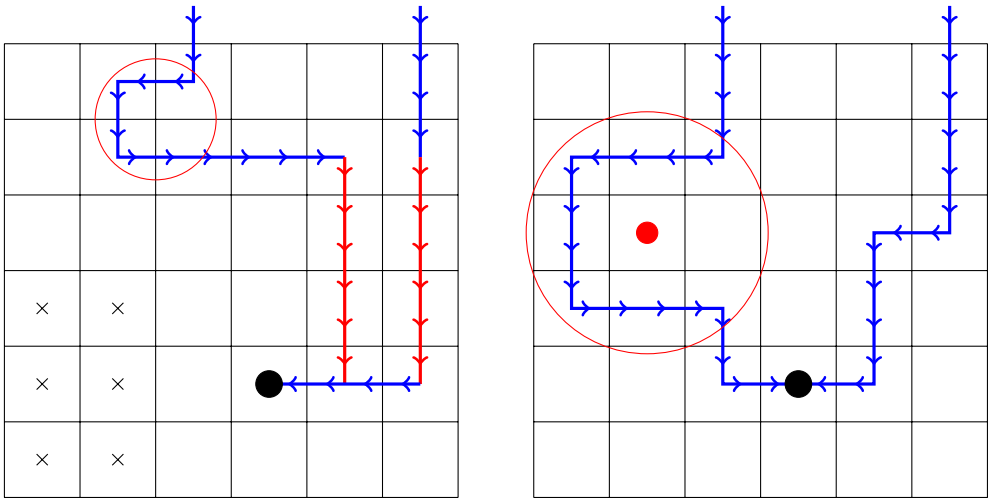


Figure 2.11 A path network with undesirable properties and a path network with great properties.

Similarly, we don't want to produce side branches that just take up more space, but don't deviate meaningfully from the original path, like on figure 2.12. To make this desire into a rule, we can define it in the following way: Every branch must go through at least one tile that is not adjacent (by an edge or by a corner) to any already existing path.

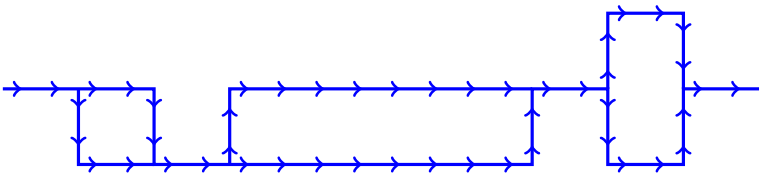


Figure 2.12 A path with undesirable side branches.

To summarize, these are the rules the paths should follow:

- **RP1** The Hub should not be near the edge of the world, and it should be close to the center in levels with multiple paths.
- **RP2** Paths are formed by one-way segments, each from the center of one tile to the center of a neighbor tile.
- **RP3** Paths start on tiles just outside the playable world, and the first path segment goes from the path start to the nearest tile in the playable world.
- **RP4** There can be one or more path starts in each level.
- **RP5** Paths can split or join.
- **RP6** All paths must end on the tile with the Hub, no other dead ends can exist.
- **RP7** Each tile with a path going through it has to be the same distance from the Hub no matter which path an attacker would take.
- **RP8** Paths should be spread throughout the playable world, not bunched up.
- **RP9** Paths right next to each other or the edge of the world, and sharp U-turns (see figure 2.11), should be rare.
- **RP10** Every branch must go through at least one tile that is not adjacent to any already existing path.

2.3.4 Attacker Types

We have mentioned that there will be many attacker types in out game. Each will be designed on its own, but they will be randomly combined to make attacker waves. An attacker type defines the following properties of an attacker:

- **Appearance**. Every attacker will be represented in a battle by its 3D model, corresponding animations and other visual effects. Every attacker type will also have an associated icon to display in the user interface.
- **Hit Points** or **HP** determine how much damage can an attacker take from the towers before it dies.
- **Movement speed** in tiles per second.

- **Size** — either *small*, *large* or *boss* — determines how much *hull* (see section 2.3.10) the player loses when this attacker reaches the Hub. Also defines the height off the ground of the spot defensive towers target. More details below.

- **Abilities**. These can be *passive* (for example "Immune to fire."), *repeating* ("Heals 5 HP every two seconds."), or *reactive* ("Spawns *attacker A* when killed.").

The height of a target a tower shoots at is important — lower targets can easily hide behind a terrain feature or an obstacle. We want some attackers to look bigger than others, and it would be weird if the towers shot at a lower portion of their model. Larger attackers will have their targeting point higher. To make things simple for the player, there are only two heights of the targeting point — *small* at 0.15 units above the ground and *large* at 0.3.

Whenever an attacker reaches the Hub, the player will lose some *hull*. The *small* attackers will come in greater numbers than *large* attackers. To make the stakes more equal, *small* attackers cost the player only 1 *hull*, whereas the *large* attackers cost 3 *hull*.

(not implemented in the demo)

The player will encounter only few *boss* attackers in every run. They will be the main attackers in special boss levels, which are spread throughout the run and cannot be avoided by the player. When a boss reaches the Hub, the player immediately loses the game. Each boss will bend the rules of the game a bit as one of their abilities, but most of them will have the same target height as *large* attackers.

2.3.5 Buildings

The player will be able to build buildings, but only between waves of attackers. They will be able to build one building per tile, if the tile has no obstacles and an attacker path is not going through it. Each building costs some amount of *materials* to build. The player will be able to delete a building at any time, mainly to make way for other buildings.

There are three building types defined by their primary function: *towers*, *economic* and *special*. Towers deal damage and kill attackers, and they are described in more detail in the next section. Economic buildings produce resources, often at the end of every wave, just in time for the player to use them to build more buildings. Some economic buildings produce resources at other times, often as a reaction to some other event, for example an attacker dying.

Special buildings are the buildings that don't fit in either category. They have a unique ability that usually increases the effectiveness of other buildings. One special building could make economic buildings produce more, another could increase the range of towers, yet another might slow attackers down.

One notable special building is the Hub, since the player starts each level with one for free, and they cannot build more. The goal of the attackers is to reach the Hub, and when they do, the player loses some *hull* (see section 2.3.10). Additionally, the Hub produces some amount of *fuel*, *materials* and *energy* at

the end of each wave. These resources are further described in their respective sections.

2.3.6 Towers

Towers are the buildings which deal damage to attackers in order to kill them. There are many properties that distinguish towers from each other. There is a lot of freedom to allow for many unique designs. Combining towers with different properties is supposed to be a fun and interesting part of the game.

Towers usually shoot once per their *shot interval*, but some towers can shoot multiple projectiles at once, others deal a certain amount of damage per second continuously. They can usually only target attackers in a circular range around them. However, some towers have an unlimited range, or their range is not circular. Most towers instantly aim at their target, some take time to rotate around and others cannot rotate at all. Some towers cannot aim upwards or downwards. Most towers require line of sight to their target, but some don't. Most towers fire projectiles in a straight line, but some don't fire projectiles, others fire projectiles that travel over obstacles along a ballistic arc. Some towers can even miss their target. With tower designs, the sky is the limit.

Whenever a tower has more attackers in its range, it will decide which one to target based on the tower's targeting priority. The player will be able to select one of these priorities on most towers:

- First — the attacker that's closest along its path to the Hub.
- Last — the attacker that's farthest along its path from the Hub.
- Closest — the attacker that's closest to this tower.
- Farthest — the attacker that's farthest from this tower.
- Weakest — the attacker with the least HP.
- Strongest — the attacker with the greatest HP.

Each tower will be set to one of these by default, but the player will be able to change the priority of any tower at any time, even during waves. This will let the player have more control over their towers, allowing them to best use their unique properties.

The damage the towers deal comes in many types. For example *physical*, *explosive*, *energy*. Some towers will deal damage of multiple types at once. This distinction lets us make some towers explicitly weak against some attackers — those that are resistant to the given damage type. Or it lets us restrict some synergies, for example by making a building that makes attackers take more damage from *energy* attacks only.

It is worth mentioning, that in most tower defense games, the player can upgrade any tower during a battle by investing more resources into it. The upgrades often increase a tower's damage or fire rate, however some substantially change the tower's behavior. Some games take this to the extreme, for example in *Bloons TD 6*, each tower has 15 different upgrades available, and each tower can be upgraded to two different upgrades at once. However, in our game, the

player won't be able to upgrade their towers during a battle. Instead, they will have to have some towers that are useful at the start of a level, and others that are more powerful, but more expensive, to be used later.

2.3.7 Abilities

Unlike buildings, abilities will be usable during waves only. They will often have only short-term effects on the attackers, so there is no point to using them outside a wave. The primary use-case is to kill or weaken attackers of a wave that might be too difficult to deal with for the towers alone. Abilities cost *energy* to use, but if the *energy* a player has is insufficient, *materials* can be used to cover the difference. The reasoning behind this is explained in the next section 2.3.8.

Most abilities will only deal damage to the attackers, each in its own unique way. However, similarly to towers, there is no restriction on what an ability can do, as long as it makes gameplay sense and offers something new. One ability could create a temporary defensive tower, another could temporarily improve the towers the player has already built. Another ability might improve the player's blueprints for the rest of the battle, yet another might just give the player some additional resources.

2.3.8 Materials and energy

Materials are the main resource within a battle. The player starts each battle with some materials, and the Hub produces a small amount of materials after every wave. Materials accumulate over the battle and there is no limit as to how many the player can have. The player can spend materials on buildings, notably towers and economic buildings. The more economic buildings that produce materials a player builds, the more materials they will have later in the battle.

Intuitively, a great strategy is to build the towers necessary to survive the next wave and spend the rest on economic buildings. However, this strategy heavily relies on the player being able to estimate which combination of towers is strong enough to beat the wave. To help, the player will have various abilities at their disposal, which can be used to kill of or weaken attackers when the towers are not strong enough by a small margin. The abilities should also cost resources to regulate their usage — stronger abilities will cost more and weaker abilities will cost less. However, if abilities also cost materials, it would be the best to spend everything on permanent defense or economy. Ideally a player would leave no materials for their abilities. That is why abilities have a resource dedicated only to them — **energy**. A steady income of energy lets the player use an ability once in a while, without costing them any long term power.

However, abilities can also be paid for in materials. All this time we've assumed that long-term power is always better than short-term. However, this is not necessarily true. In the last few waves, a powerful one-time effect is way better than a weak long-term one, since the battle is ending soon anyway. Due to this, abilities are perhaps most useful in the last few waves of a battle, and allowing the player to use materials for these lets them use the materials on whatever they think is the best. Paying with materials also helps when the player has a few materials left over, so they can use a slightly more expensive ability than they

could without this.

We don't want the player to hoard all their energy and only use it on the last waves. To encourage using abilities throughout the battle, there will be a limit on the energy a player has in reserve. This way there is much less downside to using an ability in the middle of a battle when the player's energy reserve is full anyway, so they can't get any more.

2.3.9 Fuel

We have already mentioned fuel a few times, for example in section 1.3 and section 2.1.1. But in this section, we will summarize everything about fuel.

The goal of each battle is to gather enough fuel to continue to the next level. The faster the player gathers the fuel, the sooner they win the battle. It is generated passively by the Hub, but additional buildings can be built to speed up the process. This makes the player decide when it's the best to improve their defenses and when to build fuel-producing building instead, to end the battle before the more difficult waves come, hopefully leading to greater strategic depth. The maximum number of waves each battle will take is determined by the amount of fuel the player needs to gather.

(not implemented in the demo)

We want some battles to be significantly harder, but they will provide better rewards. These will be marked on the map, so the player will decide if they want to risk a harder battle. Making these battles require more fuel to complete is a great way to distinguish them from other battles. This also applies to boss battles (see section 2.3.4).

2.3.10 Hull

The player starts each run with a fixed amount of hull. As described in section 2.3.4, whenever an attacker reaches the Hub during a battle, the player loses some hull. Once the player loses all hull, they lose the game. A player's hull is a kind of buffer that allows them to make a few mistakes throughout the whole run before they lose. They will be able to restore their hull only a few times during the run (and no hull can be restored in the demo version). For example, they can intentionally focus more on economy in the early waves of a battle, maybe letting a few attackers through, in order to be stronger in the later waves and prevent possibly greater losses. They can even take a harder battle where they expect to lose hull when they are confident they won't need it before they can restore it back. However, an experienced player can use their hull as a resource. Another option would be to have the player start each battle with full hull, but we feel that this option allows for way less strategic depth.

2.3.11 Status Effects

(not implemented in the demo)

Buildings and attackers will both be able to have status effects applied on them.

These represent temporary effects which modify the behavior of whatever they affect. They will be displayed with an icon above what they are applied to, along with a number representing their duration. The duration can be measured in seconds, but other kinds of duration are possible. The source of these effects can be anything from a tower to an attacker.

Here are few examples of effects that might be applied to attackers, x representing their duration:

- **Burning** deals a small amount of *energy* damage over time for x seconds, possibly applied by some fire-based tower.
- **Freezing** slows down the attacker's movement speed for x seconds, possibly applied by some ice-based ability.
- **Shield** prevents the next x damage an attacker would take, possibly applied by another attacker.
- **Stealth** makes the attacker untargetable for the next x seconds, possibly applied on their own.

And a few examples of effects that could be applied to buildings are:

- An **Overclocked** tower shoots 50% faster for x waves, possibly applied by an ability.
- A **Paralyzed** building is out of order for x seconds, possibly applied by an attacker.
- The next x projectiles an **Electrified** tower shoots deal additional *energy* damage, possibly applied by a support building.

2.3.12 Time controls

(not implemented in the demo)

We want the player to be able to pause the game. This is a quality-of-life feature, common among other real-time single-player games. In our game, the waves are short, and the player can just not start the next wave until ready. However, pausing will be very useful during the waves for lining up ability placement. Some attackers move fast and hitting them can be difficult, and we don't want our game to focus on dexterity or reaction time. What's even more difficult is clicking on a fast moving attacker to inspect its details (described in section 2.5.7).

We will also let the player speed up the game to play at double speed. This is useful for less eventful portions of gameplay, for example when a slow-moving attacker travels along a long empty stretch of path.

It is important that the game plays out the same no matter at which speed it's playing, and that pausing doesn't interfere with the game. For example, it would be bad if the towers sometimes missed their targets when playing at double speed. What happens in the game should also be frame rate independent.

On the other hand, everything should look as smooth as possible given the frame rate at which is the game currently rendered. Some animations will have to speed up when the game speeds up, others will still play at the same speed, even when the game is paused.

The demo version will be developed in a way that allows for this separation of game logic and game visuals. However, the time controls themselves will not be available in the demo.

2.4 Blueprints

A blueprint represents a building the player can build or an ability they can use during battles. Each blueprint will include a description that explains its function, including the exact values of important statistics — for example the amount of damage a tower deals with each hit. The player will start each run with few predefined blueprints, and they will collect more blueprints throughout the run, giving them access to more buildings and abilities.

The player will only be able to have a limited number of blueprints at any given time. Whenever they want to acquire a new blueprint while at the limit, they'll have to give up one of the blueprints they already have. This way it is impossible to make a build that is just good at everything. They will have to consider carefully which blueprints they need to cover their weaknesses and which blueprints are the most synergistic with the rest.

Each blueprint costs some *materials* and/or *energy* (see section 2.3.8) to use, though there could be some blueprints that are free. The player must pay this cost every time they want to build the given building or use the given ability.

Most blueprints will have no cooldown, so the player will be able to use them as of ten as they want. Some will have a cooldown given as a number of waves. This means some will be usable only once per wave, some only once per two waves, etc.

As we already mentioned a few times, the player will acquire new blueprints mainly after every battle. The player will be presented with a selection of three different blueprints they have not picked in this run yet. They can choose one of these to add to their collection, acquiring it for the rest of the run.

(not implemented in the demo)

Similar blueprint rewards will appear during some events the player might encounter. Some events will offer blueprints randomly selected from the same pool as battle card rewards, other events will offer predefined blueprints, which don't appear in the regular blueprint rewards. Some shops will also sell blueprints chosen randomly from the reward pool. A blueprint might even be permanently altered in an event or a shop.

In the regular blueprint rewards, some blueprints will be more rare than others. Each blueprint will have one *rarity*, which determines how often it will appear. The rarities are

- **common**,

- **rare**,
- and **legendary**.
- Additionally, there is **starter** — the player starts each run with these. Thus, they do not appear as rewards.
- And also **special** — these blueprints also do not appear in the regular blueprint rewards, but they can be obtained on other ways, usually in events.

As their name suggests, *common* blueprints will be more common than *rare* blueprints, and *legendary* blueprints will be even more rare. At first, the player will encounter a *rare* blueprint about once per a few rewards with the rest being *common*. However, towards the end of the run, most blueprints on offer will be *rare*. Additionally, some rewards, for example after harder battles or boss battles (see section 2.3.4), will contain more rare blueprints more often. The exact proportions are yet to be determined based on playtesting.

2.4.1 Design

The blueprints should be designed in such a way, that it is not great to always take the blueprint with the highest rarity. Of course, rarer blueprints will usually be stronger, but they will usually be more specific in their use. *Rare* blueprints should be similarly good to *common* blueprints in most cases, but potentially much stronger when used in the right way or in combination with the right blueprints. *Legendary* blueprints should have the greatest potential power, but in even more specific circumstances. This power should however be so great, it is worth it to sacrifice some of the build's other aspects in order to get the most out of this blueprint. The overall strategy of a given build could be defined by a few *legendary* blueprints, with the rest built to support them. Of course, some *legendary* blueprints will be useful in more builds than others. It should also be possible to make a strong build just with a good set of *common* and *rare* blueprints.

As we already mentioned in sections 2.3.5, 2.3.6 and 2.3.7, each blueprint should be unique in its own way. The most exciting designs are often those that somehow break the rules. For example, we could design a building that costs *energy*, or a tower that has to be placed on a path, or even an ability that manipulates the player's blueprints or the waves of attackers that are yet to come. However, most of these are hard to balance properly, and most important is whether they lead to fun gameplay or not.

It is also important to design a good set of starter blueprints. It should be as small as possible, but somewhat balanced in most aspects. Specifically, it should provide a way to gather *materials* and *fuel*. There should be at least two towers that can deal with the early levels, and they are distinct from each other, so there are still decisions to make even in the first levels. There should also be a starter ability, since abilities are a core of the design (see section 2.3.7). The individual blueprints should be as simple to use and understand as possible. Their power should be sufficient for the early levels, but they should be worse than other, even *common* blueprints.

2.4.2 Augments

(not implemented in the demo)

The player will also be able to upgrade their blueprints with *augments*. Each blueprint will be able to take up to 2 augments. Each augment will be a slight improvement, applicable to many blueprints. For example, one augment might increase damage, another might change the damage type, yet another might make the blueprint cheaper.

The augments will also have different rarities, determining how often they appear. Similarly to blueprints, they will usually appear as a reward in battles or events and the player will choose one of three. It will also be possible to buy them or get them in events. Sometimes, blueprints will appear in rewards with an augment already applied.

2.5 Battle Graphical User Interface

In this section, we will describe the graphical user interface that will be overlaid over the game world during a battle. The goal of the GUI is to display all the information the player might need, that is not present within the world itself, and to let the player access all the game controls without the need for a keyboard. However, for each of the controls accessible through the GUI, there will also be a hotkey the power users can use.

In figure 2.13 is a mockup of the GUI with red numbers in circles, marking each of the components. We will describe each component in its separate subsection with the last number corresponding to the number in the figure.

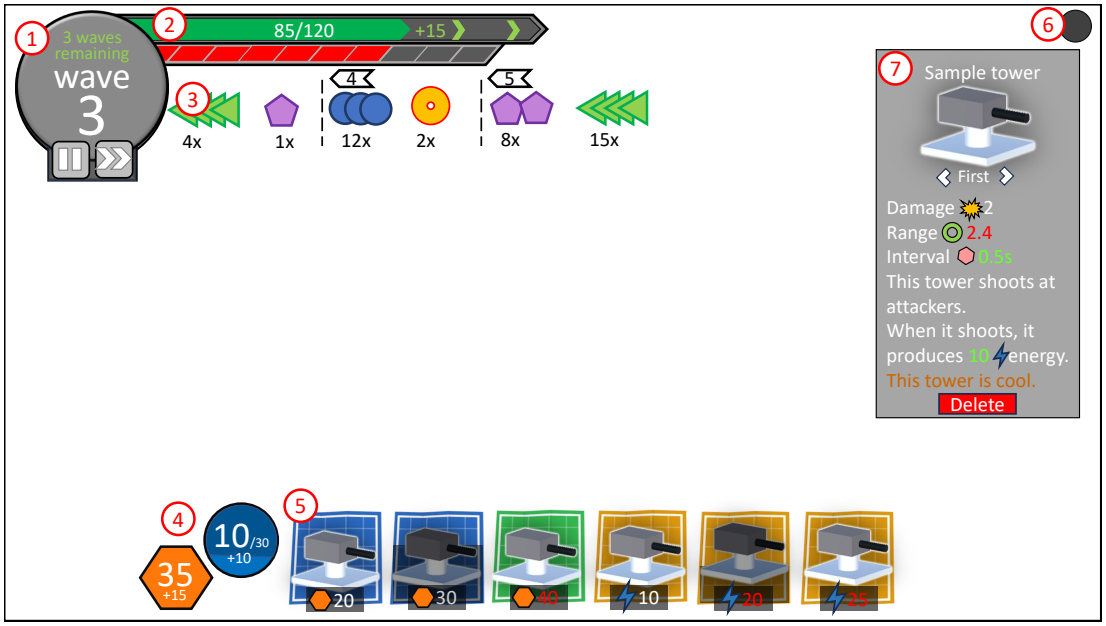


Figure 2.13 A mockup of the GUI during a battle. Red numbers in circles marking the individual components.

2.5.1 Waves and time controls

In the top left corner is displayed the number of the current wave in big white text. Above it is the remaining number of waves in green. This amount is calculated from the current amount of *fuel* and the *fuel* production per wave.

Below this, there is a panel with time controls. Here, the player can start the next wave, or pause or speed up the game. The pause button will change into play when the game is paused or between waves. The speed button will change between one arrow and two arrows, based on the currently selected speed.

2.5.2 Fuel and Hull

Immediately to the right of the wave s display is displayed *fuel* and *hull*. The green progress bar displays how much *fuel* the player has out of the total required to finish the level. It also displays these values in text. Additionally, it shows the amount of *fuel* produced per wave and light green marks previewing the *fuel* values after each of the upcoming waves.

Below is a red progress bar showing the player's *hull* out of the maximum amount.

2.5.3 Wave Preview

Below the *fuel* and *hull* displays it the wave preview. It shows the composition of the current and upcoming waves. Each wave has an arrow with its number and the batches of attackers that will come. Each batch shows one or more icons of the attacker type it's composed of along with a count. The icon spacing represents the spacing of the attackers in the wave itself.

2.5.4 Materials and Energy

At the bottom of the screen is the *materials* and *energy* display. In the orange hexagon is shown the current amount of *materials* the player has, along with a smaller number below it representing the *material* production per wave. Similarly, the blue circle shows the current amount of *energy* and the *energy* limit, with the *energy* production below. The circle is partially filled with a lighter blue to visually show how much *energy* the player has out of the maximum.

2.5.5 Blueprint Menu

To the right of *materials* and *energy* is the blueprint menu. Here the player can select their blueprints to use them. Between waves, only buildings are shown, and during a wave, only abilities are shown.

Each blueprint is shown as a colored square of paper with an icon over it. The color represents the type of the blueprint:

- blue for towers,
- green for economic buildings,
- purple for special buildings,

- and orange for abilities.

Overlaid over the lower portion of the blueprint is its cost. This is what the blueprints look like everywhere in the game, for example in blueprint rewards.

The cost number turns red when the player has insufficient resources. The cooldown is also shown as a partial dark transparent overlay over the blueprint paper. The portion it covers represents the portion of the cooldown that's left until the blueprint can be used again.

By clicking on a blueprint in the blueprint menu, the player will select it. To use it, they will have to click somewhere in the world to specify at which position do they want to use it. The player will also be able to select the blueprints using the number keys **1** to **9** on their keyboard, based on the blueprint's position in the blueprint menu.

2.5.6 Settings Button

In the top right is a button which lets the player access a settings screen. Here the player will be able to access some game settings like sound and music volume. They will also be able to exit back to menu from this screen. When the settings screen is opened during a wave, the game pauses.

2.5.7 Info Panel

At the right edge of the screen is the info panel. It shows up only when the player has selected something, and it displays information about the selected thing. It also shows up when the player has not selected anything, but only hovered over something selectable with the cursor. In this mode, the panel is semi-transparent and no buttons are showing (more information below). This panel will appear whenever the player selects a relevant object, even on other screens, not just in battle. For example, when the player selects a blueprint in a shop.

At the top is the name of the currently selected object. For now, let's assume it's a tower, as shown in the picture. The name is over a large icon of the tower's blueprint.

Over the lower portion of the icon is the targeting priority selector. It is displayed only when there are targeting options to choose from. Usually, only towers can have a targeting priority, and not all of them have it. The targeting priority selector consists of the name of the currently selected priority, and arrows to the left and right of it to switch to the next or previous priority. The priorities are usually a subset of those described in section 2.3.6.

The main portion of the info box is taken up by the description. The contents of the description depend on the selected object and will be specified later. However, there is a few special features the description has that can appear no matter the selected object type.

First of them are icons in the text. These icons are used for important stats or quantities that appear often. For example there is an icon associated with damage, one for range, one for *materials*, etc. The icon usually appears before the mention of the thing it's associated with or before the stat. Examples of this use can be seen in figure 2.13.

Most of the quantities in the description can be modified in various ways. If there is some sort of original version which differs from the current version, we can highlight the changed quantities using colors. Red means the quantity is now worse than before, green means it's better. Note that for some statistics, higher is better, and for others, lower is better. Additionally, it is possible something added a new description that wasn't in the original version. For example, a special building could add new abilities to neighboring towers. This new description is highlighted in orange.

Below the description is a button that allows the player to delete the selected building. Of course, this button only appears when the selected object is a building that it is not permanent, unlike for example, the Hub.

As mentioned, the info panel will display the description of **blueprints**. A blueprint's description contains sentences that explain what the thing, which the blueprint represents, does. Additional quantities that don't fit into these sentences are summarized at the top, as shown in figure 2.13. Also, when a blueprint is selected from the blueprint menu, it will show the remaining cooldown if any.

The descriptions of **buildings** are basically the same as their blueprints. However, buildings can also provide some additional information of their own, for example the total amount of damage a tower has dealt or the amount of *materials* a building has produced.

Similarly, **attacker stats** can be displayed, for example when the player encounters a given attacker type for the first time. The format is the very similar to blueprints with base stats (see section 2.3.4) at the top and additional abilities described in sentences below. Attackers can also be selected during a wave. The changes are the same as with buildings and blueprints, but also the attackers always show their current HP in addition to their max HP.

The player can also inspect an empty **tiles**. Here only little information is provided. Whether the tile has some obstacles, or is rich in *minerals* or *fuel*, whether it is slanted and whether a path runs across it. If the tile has a building on it, the details of the building are displayed instead.

2.6 Attacker HP Indicators

During a wave, we want the attackers in the world to have HP indicators above them to communicate to the player how much more damage they need to deal to a given attacker to kill it. These indicators should be overlaid over the attackers and always face the camera. Their purpose is to let the player know how much HP the attacker has left. There is going to be a lot of attackers on the screen, and thus a lot of HP indicators. They should be simple in order to be legible when viewed when the camera is zoomed out and not too noisy in large quantities.

Thus, we will use the design displayed in figure 2.14. It has a rectangular shape, and it should be one size for all small attackers and one size for all large attackers. The rectangle is filled in with a colored fill on a dark gray background. This fill represents the portion of the attacker's HP that's left out of their maximum HP.

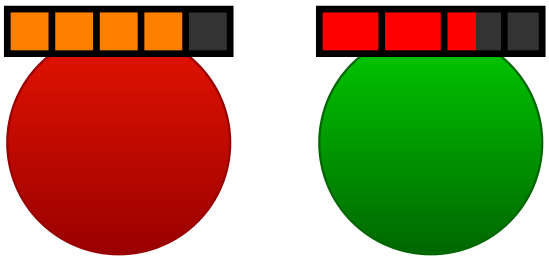


Figure 2.14 Two attackers with HP indicators. The one on the left has 4/5 HP, the one on the right has 25/35 HP.

Over the colored fill, there are black vertical lines dividing the indicator into sections. Each section represents the same amount of HP. In the HP indicator above the attacker on the left, each section represents 1 HP. Thus, we can count that the attacker has 4 HP out of a maximum of 5. However, some attackers will have a lot of HP, and section of just 1 HP would be impractical for those. So each section will represent a power of 10 of HP, and we will distinguish them by the color of the fill. Orange for 1 HP sections, red for 10 HP sections, dark red for 100 HP sections, and potentially also magenta and purple if larger values are needed.

We always choose the largest section size that leads to the indicator having more than one section, so there is always 2 to 10 sections. It is also important to note, that the rightmost section will not necessarily represent the amount of HP it should. For example when an attacker has 35 HP, their indicator will consist of three 10 HP sections and one 5 HP section. This last section will be scaled accordingly, in this case it will be only half as wide as the whole sections, as seen in figure 2.14.

2.7 Selection and Highlighting

In section 2.5.7 we've mentioned that the player will be able to select blueprints, buildings, tiles and attackers. We want to communicate to the player what is currently selected. We can accomplish this by displaying a blue outline around the selected object. Similarly, we can show what's the player's cursor hovering over with a lighter shade of blue.

During a battle, buildings can often affect other objects, for example, towers usually have a big impact on the HP of surrounding attackers. We can use similar highlights to show these relations. All attackers within the range of a tower should be highlighted in yellow when that tower is selected. Similarly, when placing an ability that immediately affects attackers, we should highlight the attackers that would be affected. As another example, while the player is placing a building that affects other buildings, or when the player selects this building, all the affected buildings should be highlighted. For some objects, for example most attackers, we don't even want to highlight anything else. However, what to highlight will have to be decided on a case by case basis, because the objects in our game can have all sorts of unique behaviors. Additionally, when the player hasn't selected anything, but has hovered over some object, we can also display its relevant highlights, since we're not displaying the highlights of any selected object.

Relevant objects won't be highlighted only based on what's selected. As we indicated, when the player has selected a blueprint in order to use it, we can preview what would be affected if the player used it right now. We should also highlight as hovered only the object relevant to the blueprint's placement. When placing a building, we don't want to highlight the attacker the player's cursor is over, because building are not used on attackers. They are built on tiles, so we should highlight the tile the attacker is on. Similarly, when the player is about to use an ability that can be placed at any point on the surface of the world, we shouldn't highlight tiles or attackers, but only highlight the exact point the ability is about to be placed at. This placement highlight can also indicate when a placement is invalid by turning red instead of being blue.

When placing something that can only be placed on specific tiles, we can also highlight these tiles. For example, most towers cannot be placed on tiles with obstacles or a path. There is usually going to be a lot of valid tiles, so this highlight should be a bit more subtle. We think the best way to show this is to tint the terrain of the valid tiles in blue.

We also want to show the range of the ability or tower the player is currently placing (or a tower that's selected). The player needs to see the range in the context of the world, just a number is not enough. The most clear way to do this is to also draw this range visualization directly on the terrain. This is because the situation could be viewed from different angles. In most cases, this would be just a simple circle, however, some towers or abilities can have an unusual range shape, for example a straight line. Most importantly, a lot of towers will only be able to shoot at attackers in line of sight, which will create all sorts of unusual range shapes. We definitely need to communicate to the player where the tower can hit attackers and where it cannot.

In figure 2.15 is an example range visualization for a tower that requires line of sight. The tower in question is represented by the blue square. This situation is showed from a top-down view, however in game, this would take place on a 3D terrain and this range visualization would have to be drawn correctly even from a different angle. In the figure, the tower is represented by the blue square in the center. The black lines separate different levels of the terrain, the lowest level is colored with the darkest shade of gray. In the top part of the figure, we can see the tower's line of sight is being blocked by a higher terrain. The visualization also shows that this tower cannot shoot uphill, since there is nothing drawn on the higher level of the terrain. In the top left quadrant we can see the tower's line of sight by some large obstacles. The tile immediately to the right of the tower is slanted, transitioning to a lower level. On the lower level, we can see some unusual shapes emerge. This gap in the range appears, because the terrain level the tower is on is in the way when the tower wants to shoot at attackers close to the base of the overhang.

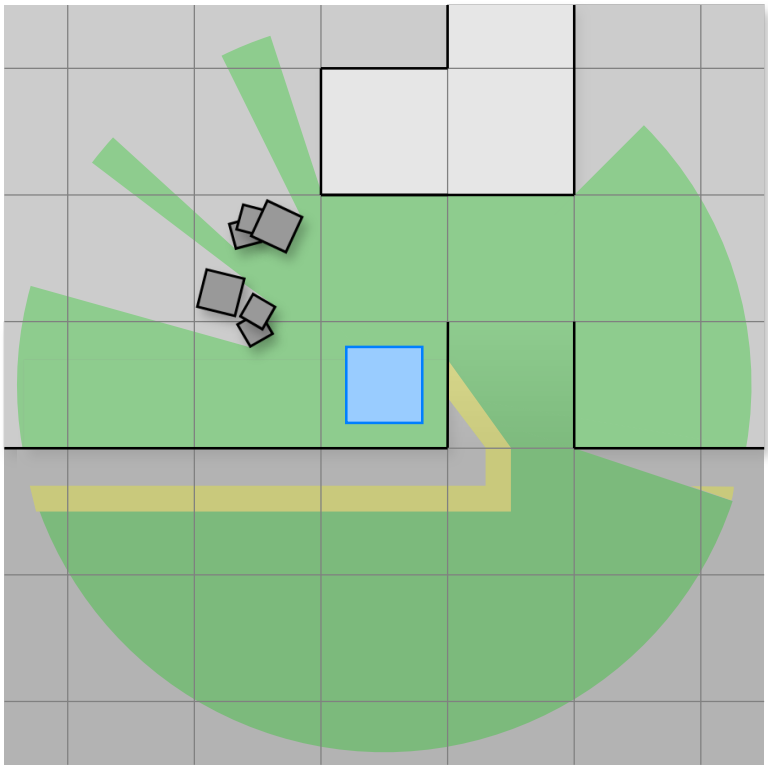


Figure 2.15 Example visualization of a tower's range.

Here, we can also see a portion of the range colored in yellow. This means that if an attacker was at this point, the tower would only be able to shoot at it over the ledge if it was a *large* attacker. In section 2.3.4 we have explained that there will be two sizes of attackers. This means that there will be a region where a tower can't see any attackers, a region where the tower can see any attackers (green), a region where the tower can only see *large* attackers (yellow), but also a region where the tower can only see *small* attackers. The last case is, however, very rare, so we can draw this region also in green.

Calculating the exact shape of the region the tower can see can be expensive. We could simplify this by only computing the range visualization where an attacker could appear — on a path. However, we think it is important to show at least an approximation of the whole picture, to make it easier for the player to see what's blocking the line of sight and where.

2.8 Battle Camera Controls

During a battle, the world will be rendered onto the player's screen using an orthographic camera at an angle that mimics isometric graphics [12]. The world is pretty large, so we want to let the player zoom in. They will be able to do so using the mouse scroll wheel. Most of the time, the world will be viewed at an angle in to clearly convey the various heights of the 3D terrain. However, in some cases it would be useful to have a top-down view, for example when inspecting whether a path intersects with a tower's range. We can blend these functions by slowly tilting the camera to look straight down at the maximum zoom level. This is the zoom level at which it's best to inspect details anyway.

Since the camera can be zoomed in, the whole world won't necessarily be on the screen at once. The player will have to be able to move the camera around. Additionally, high terrain can sometimes obstruct what's behind it, so we will allow the player to rotate the camera around the vertical axis, bot only in 90 degree steps to preserve the isometric look.

2.9 Future Features

In this section, we would like to mention other features that will be in the full game, and we've not described them earlier in this chapter. These features will also have to be designed eventually, but they will not be in the demo, so only rough descriptions are provided.

(not implemented in the demo)

2.9.1 Setting and Story

From the names we've given to various game components, it might be clear that we're leaning towards some kind of sci-fi setting. The player would travel across a galaxy using a spaceship, stopping on various planets along the way to refuel for further travels. But what is the main character's story? What is their ultimate goal? Well, the story has not been decided yet. However, the story should be an important part of the game, and it will inform a big portion of the game's further design. Any game can always benefit from an engaging story, especially single-player games.

2.9.2 Run Structure

We want each run of the game to last at most 2 hours, since we want the players to be able to complete each run in on sitting. Each run will probably consist of three acts, each ending in a boss level. This is the same structure as in *Slay the Spire* or *Monster Train* [13], which are both roguelike deck-building games. It seems to work very well for them, so it is where we'll start. However, the final run structure will heavily depend on playtesting.

One thing that is clear even now, is that each battle will take substantially longer than in *Slay the Spire*. *Monster Train* also has longer levels, so each run consists only of 9 fights. It is reassuring to know that a roguelike of this kind can work even with fewer levels. Even though *Monster Train* has fewer levels than *Slay the Spire*, it has a similar amount of rewards and shops throughout the run, letting the players customize their build to a similar degree, if not greater.

2.9.3 Saving the Game

Even though we want the players to be able to play a run in one sitting, saving their progress is still a very useful feature.

Ideally, we would save the game after every choice the player makes and after every tick of simulation during a battle. However, this is technically unfeasible. We could save the game whenever they explicitly choose to save, for example when they exit the game. This would still be pretty difficult, since we would need to serialize the state of all towers, attackers and projectiles, and then we would need to be able to load and initialize everything correctly again.

We could take the approach both *Slay the Spire* and *Monster Train* use and save the game whenever the player makes a decision outside of battle, and during battle only at the start and the end. This would definitely work and be usable. However, we will aim to save the game at the start and end of every wave. In *Bloons TD 6*, the game is saved at the end of every wave only, and the player can build towers even during a wave. This means that a player can exit the game right before they would lose and redo the wave, with a different strategy. In our game this sort of save scumming would be less effective, since during a wave, the player can only use abilities, which are meant to be used reactively anyway.

2.9.4 Money

We that it would be good to add another resource, one that persists throughout the entire run. The main use of this resource, for simplicity called money, would be to pay for items in shops. This would allow the player to make more decisions than for example, if they got one thing in every shop for free and nothing more. This way they can save up their money when a shop doesn't offer anything they'd like to buy, and spend more at shops with things they like. Of course, money would be acquired mainly by winning battles and sometimes in events.

2.9.5 Permanent Unlocks

The most strict definitions of a *rogue-like* mandate that the game doesn't let the player unlock any permanent improvements which persist between runs. The players themselves should improve by playing the game, not their in-game characters. We like this idea and wish to preserve it even within our game. However, it is possible the player will be able to unlock more of our game's content as they play, in a way that is similar to *Slay the Spire*. In *Slay the Spire*, the player starts off as a character called *The Ironclad* and only after playing one game they unlock the second character. Furthermore, for each character, the game offers five sets of new cards and relics to unlock. These unlock serve mainly to not overwhelm a new player with choices or items which are harder to understand.

3 Analysis

Now that we have described the game's design, in this chapter, we will explain the approach we took to implement it from a high-level perspective. We will provide concrete details only for what will be implemented in the playable demo version, but as always, we will make many decisions based on the original vision of our game.

3.1 Game Engine

Game engines provide many important and useful systems for us, so we can focus on implementing the game logic. For our game, we chose Unity because it offers all the features we need, and the author is already familiar with it. There are many game engines we could have used, and the high-level decisions presented in this chapter would be still applicable. However, in some sections we will use nomenclature that is specific to Unity, so we assume the reader is at least familiar with it. More information is available in the official documentation [14].

3.2 Procedural Generation

As explained in the previous chapter, a lot of the game will be procedurally generated, including the map of a run and each battle along the way. In the next few sections, we will decide how to generate the worlds for the battles, and in section 3.6, we will focus on generating the waves of attackers.

We also want to procedurally generate the map of each run, however the run map will not be a part of the demo version of our game, so we won't implement the map generator yet. Many of the parameters for the procedurally generated battles will be decided by the map generator. For example, at the start of each run, the battles should be easy, and they should gradually get more difficult further into the run. Additionally, some battles will be special in some way. For example, some battles will be harder, or use a unique terrain type. We should decide what will be determined by the map generator before the battle generates, and what will be determined only once the battle starts generating.

The map generator will select the terrain type of the world and what attacker types will appear. These parameters define the theme of the battle. To control the difficulty, the map generator dictates how difficult should the waves of attackers be, and how much *fuel* is required to finish the level. This is further explained in section 3.6. The number of attacker path starts and the path lengths also greatly influence the difficulty. More paths are harder to cover with defenses than a single path. Shorter paths give the player's defenses less time to deal with the attackers than longer paths. The map generator will also define the maximum number of path branches, since the paths can split into more. This is mainly to limit the complexity of the path network in some levels. There are many more factors which influence the difficulty of the battle, but they are more difficult to quantify, and we believe their influence is not as great.

The map and the map generator will not be a part of the demo, but we still

want to let the players play more levels and collect blueprints, until they inevitably lose. So, we will just create a simple system that will set up the levels with gradually harder attacker waves, shorter paths and different numbers of paths.

The worlds the battles take place on are composed of three somewhat distinct parts — paths, terrain and obstacles. It would make the most sense to generate each part separately, one after another. We should start with the part that is the most restricted, because each part is additionally restricted by what was generated before it. For this reason, we will start with paths. There are a lot of rules the paths should follow, as described in section 2.3.3. Additionally, the map generator exactly specifies their number, lengths, and maximum number of branches. So, we will generate paths first (section 3.3), then the terrain (3.4), and finally, the obstacles (3.5).

All the randomized algorithms we will use require a source of randomness. For reasons described in section 2.2, we need to choose the right random number generator for our use-case. This is further explained in section 3.7.

3.3 Path Generation

In the previous section, we decided that when generating a world, we will start with the paths. We also mentioned that we will get the number of path starts, their path lengths and the maximum number of branches as an input from the map generator, because these values heavily influence difficulty. In section 2.3.3, we outlined many requirements and suggestions for the paths, in order to make them play well. Generating a path network with good properties is not an easy task. To simplify it, we can split the path generation process into three simpler problems:

1. Select the Hub position and path starts.
2. Generate the main branch from each path start.
3. Refine the paths and make them split and join.

How to accomplish the goal of each of these stages will be described in the following subsections of this section.

Before we continue, we would like to define several terms which we'll use in the rest of this section. These are illustrated in figure 3.1.

As stated in section 2.3.2, the world is formed by a 15×15 grid of square **tiles**. Each tile shares an edge with up to four **neighboring tiles**, or **neighbors**. The outermost tiles of the world which have less than four neighbors are the **edge tiles**. The tile the Hub is on is the **Hub tile**. Some tiles can be marked as **path starts**.

A **path network** consists of *path segments*. Each **path segment** is an oriented straight line from the center of one tile to the center of its neighbor. We can think of them as the edges in an oriented graph, with tiles being the nodes. A tile with at least one segment starting or ending at it is a **path tile**. A **path** or a **branch** is a sequence of consecutive segments. The **number of extra branches** of a path network could be defined as the sum of the number of outgoing segments from each tile beyond the first. For example, the path network in figure 3.1 has

2 path starts, and because there are two tiles with two outgoing segments each, they make for 2 extra branches.

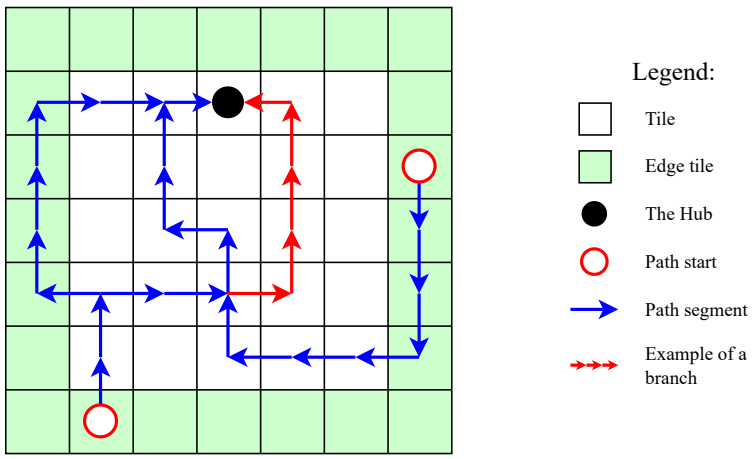


Figure 3.1 A path network in a 7×7 tile world.

If tile u can be reached from tile t by going along d path segments, we say the **path distance** from t to u is d. Two tiles can have multiple different paths between them, however by requirement RP 7, all of these paths must have the same distance. When u is not reachable from t, they do not have a path distance. The **path length** of a start dictates its path distance to the Hub. For example, the path start on the bottom left of figure 3.1 has path length 9.

3.3.1 Hub Position and Path Starts

In the first stage of path generation, we need to select the tile the Hub will be on, and all the path starts. This will be informed by the number of path starts we have to generate and their path lengths.

Hub Position

First, we will select the Hub tile. According to requirement RP1, it "should not be near the edge of the world, and it should be close to the center in levels with multiple paths". There aren't any more requirements, so we will simply select a random tile from tiles that are at most some distance from the center using the euclidean metric. This distance will be the greatest for levels with one path start and decrease with each additional path start.

Path Start Requirements

Now we select the path starts. According to requirement RP3, "paths start on tiles just outside the playable world, and the first path segment goes from the path start to the nearest tile in the playable world." However, other path segments are confined to the actual tiles of the world. For simplicity, and to avoid edge cases when generating paths, we will pretend, that the paths start at an edge tile of the world, where the first path segment will end.

We will add this segment going over the edge only after the paths are generated. This segment is uniquely determined anyway, except in the corners of the world, where we'll always select the one that makes the path go straight, as shown in

figure 3.2. On the right are the pats as we think of them when generating, and on the left are the actual final paths. The red circles represent the path starts, and the arrows represent the first segment of each path that only gets added at the end.

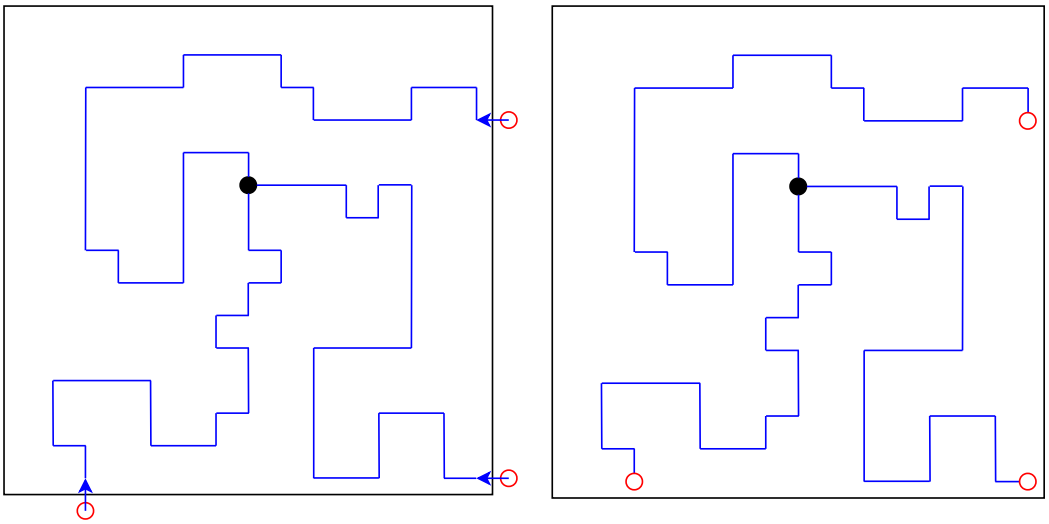


Figure 3.2 Real path starts compared to the ones we work with.

What tiles are valid for a path start with a given path length? Each start must be at an edge tile, such that a path of the given length can go from it to the Hub. It is easy to see that the minimum path distance between two tiles is their Manhattan distance. So, we know that each start cannot be further from the Hub in Manhattan distance that its path length.

We can imagine creating a path backwards from the Hub by appending a segment at every step. After the first step, the path starts on the neighboring tile of the Hub tile. Both Manhattan distance and path distance from the tile to the Hub are 1. After every step, the path distance increases by 1, changing its parity from odd to even or vice versa. The Manhattan distance to the Hub either increases or decreases by 1, also changing its parity. Thus, the parity of the path distance and Manhattan distance always match. This means a path start with an even path length must be a tile with an even Manhattan distance to the Hub, and analogously for odd lengths.

In levels with more paths, we want the path starts to be spread out from each other, in order to cover the world with paths more evenly. We can choose a minimum distance between the paths starts. For levels with just one long path, we can also set a minimum distance from the Hub, so it starts further away from it and has more space to zigzag through the world.

Selecting Path Starts

To actually select the starts, we find all edge tiles, and separate them into two sets — each for one parity. Then to select each start, we can use rejection sampling. This means we take random tiles from the set of with the correct parity until we find one that satisfies all the conditions. As long as the minimum distances between path starts and from the Hub are small enough, this approach always yields a valid list of path starts. However, with stricter parameters it is possible that the first few starts invalidate all other start positions. In that case we can

use rejection sampling again — trying to randomly select lists of starts, until we get one that's valid. If the failure rate was great enough, it would be wise to select a different algorithm, but our requirements are not very strict, so rejection sampling works fine.

3.3.2 Generating the Main Paths

From the previous step, we have the Hub position and all the path starts. Now we want to generate the main paths, which will serve as a template for the next steps. These paths have to have the correct lengths and follow the requirements, which we set in section 2.3.3.

We couldn't find many resources on procedurally generated paths. There is a lot of research on generating road networks, mazes or dungeons. Theoretically, we could use one of the many algorithms for generating mazes [15], and modify it to suit our needs. However, it would be difficult to achieve what we need using an approach that was designed for something else.

We found one algorithm specifically designed for procedurally generating paths, called *path chiseling* by Boris the Brave on his blog [16, 17]. This algorithm creates random paths on a tile grid by randomly blocking off individual tiles until only one path remains. This is more promising, however we were unable to find a good way to modify it to generate paths of specific lengths with the properties we want.

Since many of our requirements are more like suggestions, we always want to fulfill them almost the best we can. This means we can look at the task as an optimization problem: create the *best looking* paths, given the requirements like length, no crossing etc. Since we want the paths to be randomized, we don't need to find the optimum, we only need a random solution that is good enough. Given this, we decided to generate the paths using an optimization technique called *simulated annealing*.

3.3.3 Simulated Annealing

Simulated annealing can be used to find an approximation of the global optimum of an optimization problem, much faster than it would take to find the exact global optimum. A great analysis of this technique can be found in the article *Optimization by Simulated Annealing* [18]. In this section, we will describe the technique, and in the next section (3.3.4), we will use it to generate the paths we want.

The problems simulated annealing can be used for have to be formulated as follows:

> From the set of all states S, find a state s^* that minimizes the cost function $f\colon S \to \mathbb{R}$, given a neighbor function $n\colon S \to \mathcal{P}(S)$ which gives the *neighbor states* of each state.

For example, to use simulated annealing to solve the *travelling salesman problem*, each state is usually defined as a permutation of the cities to be visited. The cost function then gives the length of the salesman's path, and the neighbor function gives all the states that can be acquired by swapping two cities in the original state.

The process of simulated annealing is described in pseudocode as algorithm 1. It starts in an initial state s_0 and runs for max_steps steps. For each step, a temperature t is computed, slowly decreasing from $t_{initial}$ in the first step, to t_{final} at the final step. In each step, a random neighbor s' of the current state s is selected, and an acceptance probability p is computed, based on the values of $f(s)$, $f(s')$ and the current temperature t. The new state s' is then set as the current state with probability p.

This acceptance probability function can be implemented however we see fit, however it should follow these rules: It always accepts a better new state (s' such that $f(s') < f(s)$), but it can also give a non-zero probability when the new state is worse that the current state ($f(s') > f(s)$). The probability to accept a worse new state decreases with decreasing temperature. The acceptance probability of state t cannot be greater than the probability of u when t is worse than u ($f(t) > f(u)$).

Algorithm 1 Simulated annealing

```
1: s ← s_0
2: for k from 0 to max_steps − 1 do
3:     t ← Lerp(t_initial, t_final, k/(max_steps − 1))
4:     s' ← random neighbor from n(s)
5:     p ← AcceptanceProbability(f(s), f(s'), t)
6:     with probability p: s ← s'
7: end for
8: return s
```

It is easy to see that if the algorithm never accepted states which are worse than the current state, it would gradually reach a local optimum. Since it also accepts worse states, it can move away from the local optimum and hopefully end up in a better one. The probability to accept a worse state gradually decreases with the decreasing temperature. So at the start, when the temperature is high, the algorithm explores the search space a lot, but as the temperature decreases, it is less and less likely to escape from the local optimum it finds. Since the exploration prioritizes better states, it is more likely to lead the algorithm to a better optimum than a worse one.

3.3.4 Generating Paths using Simulated Annealing

To use simulated annealing to optimize paths, we need to formulate our task as an optimization problem that simulated annealing can solve. We need to define what is a state, a cost function and a neighbor function.

States

Each state is a path network composed of one path for each path start. We will represent each path by a sequence of tiles the path goes through. Two consecutive tiles in the sequence must be neighbors. Additionally, each path starts at the given path start, has the correct length, and ends at the Hub tile. We chose this representation to make it easy to generate neighbor states. Notice, that we don't

check for any intersections and a path can visit one tile multiple times. This is because the state is going to change only by a small amount at every step. If we banned intersections, we would lose too much freedom during the simulated annealing, and the final state would always end up close to the initial state.

Neighbor States

We decided that two states are neighbors when they differ only by one tile. To generate the set of all neighbors, we have to find all the ways to change one tile in the state to a different tile, such that the result is still a valid state.

For a state to be valid, we require that each path starts with the path start and ends with the Hub tile. This means that the first and last tile of every path can never change. For each other tile, there aren't many options on how to change it. All these options, up to symmetry, are illustrated in figure 3.3. The tiles in the sequence are drawn as circles connected by arrows that show their order in the sequence. The tile we plan to change is drawn as a red circle. Additionally, we draw the tiles which go before and after it in the sequence. These are the only tiles that determine the possible changes.

When the tiles form a straight line, no change is possible. When they form a right angle, one change is possible. And when the same tile comes before and after, 3 changes are possible.

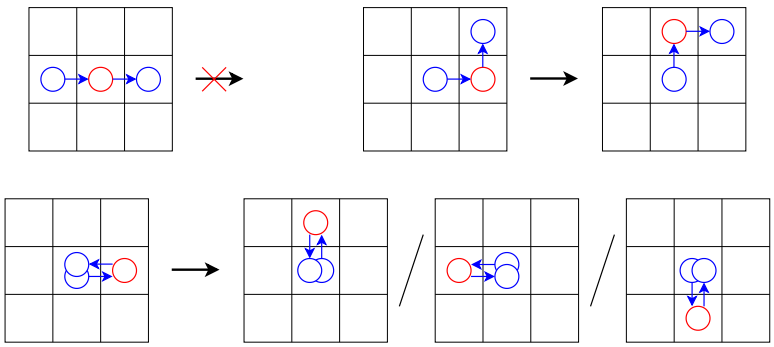


Figure 3.3 All the possible tile changes.

Cost function

Now we select a cost function that gives a better score to paths that are more desirable. Since we want the paths to spread out, we can calculate for each tile of the world what we call a *crowding penalty*. Each time a tile appears in the state, it's crowding penalty increments by 1, and all other tiles get a lower penalty decreasing with euclidean distance. This way, the tiles that get visited the most times get the highest crowding penalty. A tile can be substantially crowded even when no path goes through it just because there is a lot of paths around it.

We will denote the crowding penalty that tile t gets from tile u as $c(t, u)$. The total crowding penalty from tiles in state s for a given tile t is then $c_t(t, s) = \sum_{u \in s} c(t, u)$.

We also add a big crowding penalty to the tiles along the edge of the world, which gradually decreases as we go away from the edge. This is to push the paths away from the edges, as that is undesirable by requirement RP9. We will denote the crowding penalty that tile t gets from its closeness to the edge as $c_e(t)$.

We define the cost function of a state s as the sum of the crowding penalties of each tile in the state:

$$f(s) = \sum_{t \in s} (c_e(t) + c_t(t)) = \sum_{t \in s} \left(c_e(t) + \sum_{u \in s} c(t, u) \right)$$

This is an obvious choice, because we want each tile to be crowded as little as possible. So, a state with less crowded tiles has a lower cost, and a state with more crowded tiles has a higher cost.

However, we can just calculate the relative improvement $r_i = f(s) - f(s')$ between the current state s and the new state s' to save on computation. The relative improvement still gives us enough information to create a good acceptance probability function, but it is simple to compute, because s and s' share most of their tiles.

For s' obtained by changing tile v in s to w, we can calculate r_i as:

$$r_i = c_e(v) - c_e(w) + 2c_t(v, s) - 2c_t(w, s) + 2c(w, v) - 2$$

We prove this in section 3.3.5. This can be computed in $O(1)$ time if we keep the $c_t(t, s)$ of every tile t in the world. We have to keep all c_t updated when we change the current state to the new state. This update is simple for each tile t:

$$c_t(t, s') = c_t(t, s) - c(t, v) + c(t, w)$$

Initial State

Now, with the problem formulated, we still need to fill in a few details to be able to solve it. First, we need to produce an initial state. This is not trivial, because of our constraints on what's considered a valid state. Namely, every path has to be the correct length. However, we can easily produce a valid initial state using a random walk from each path start.

The algorithm starts on the path start tile, and adds it to the state as the first path tile of this path. Then it moves to a random neighbor and appends it to the sequence. This is repeated until it creates a path of the correct length. However, we need to ensure that the path ends at the Hub. To achieve this, we just make the algorithm never select a tile that is further away from the Hub in Manhattan distance than the remaining length of the path.

Acceptance Probability Function

Next, we need to select an acceptance probability function that works well for this problem. We don't compute the cost of the current state $f(s)$ and the cost of the new state $f(s')$ separately, instead we compute only the relative improvement $r_i = f(s) - f(s')$. This means that the function will decide on the probability only based on the relative improvement and the temperature. The function needs to fulfill the requirements outlined in the previous section 3.3.3. The most straight-forward function is simply $r_i - t$. This function can return values greater than 1 and less than 0, this does not matter, because when testing the probability, these values get treated as 1, respectively 0.

Is this the best function for this problem? We don't know, but it performed well in our testing, so we kept it.

Intersection Untwisting

However, when we use simulated annealing with these parameters, it still sometimes produces paths that intersect. This is because it is difficult for the algorithm to fix a loop in the path, as shown on the left in figure 3.4. It would first have to bring many path tiles closer together, in order to let them cross over each other. This is the sort of problem simulated annealing is supposed to be able to overcome. However, we can help it by adding a step that just *untwists* crossings by reversing the section of the path that forms a loop, as shown in the figure. This still leaves two identical tiles, but now, simulated annealing can drive these apart without any issue.

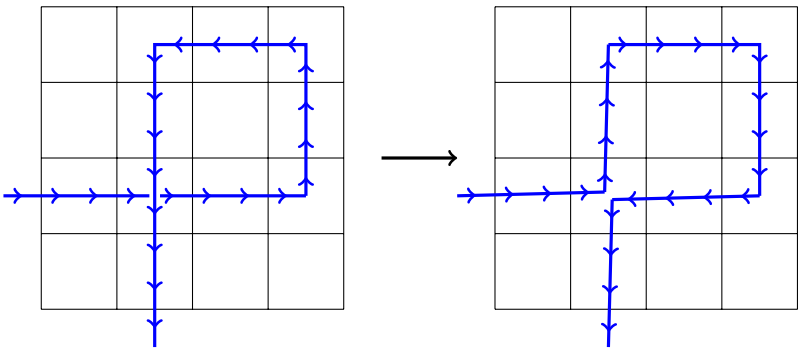
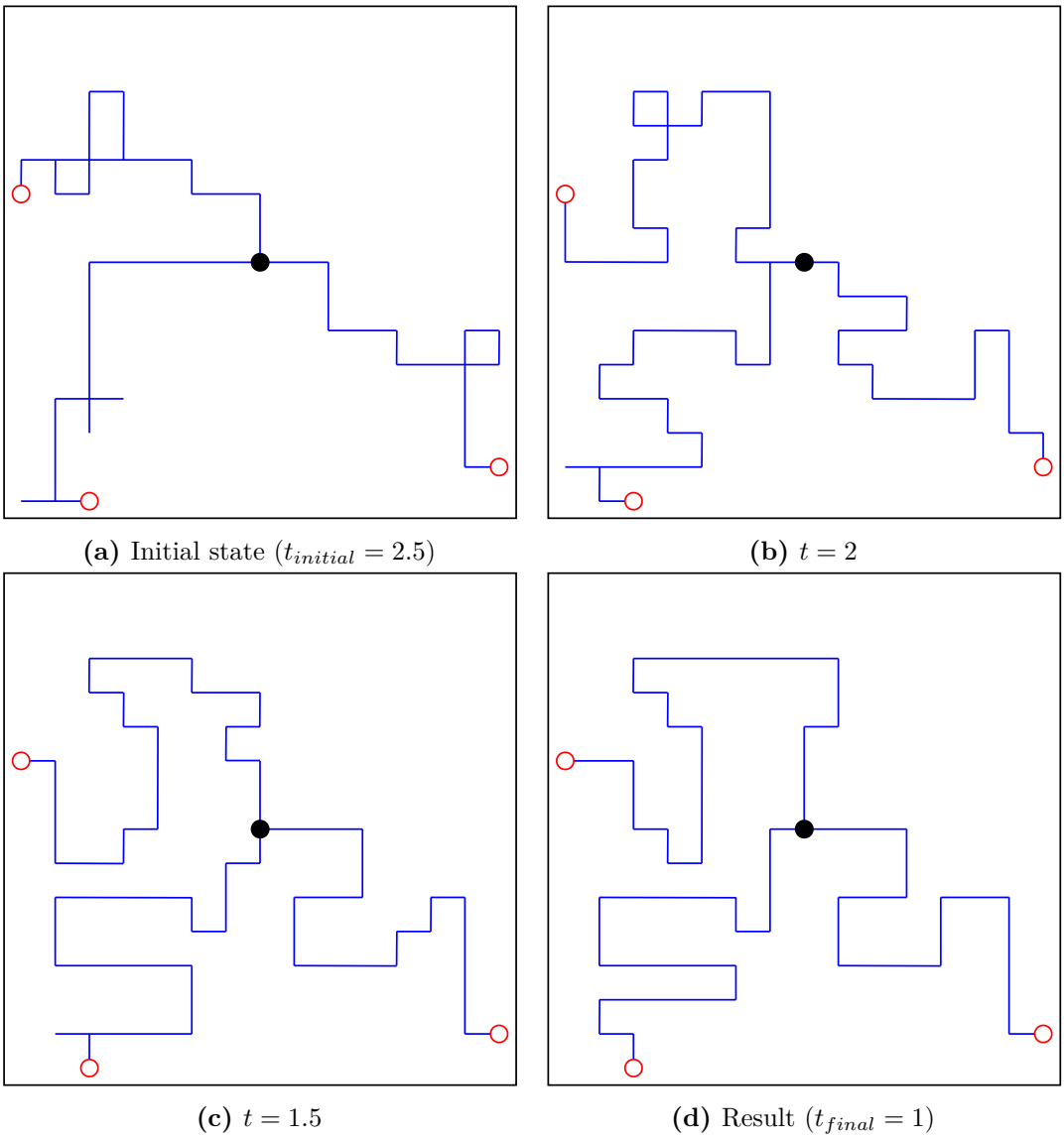


Figure 3.4 Untwisting a self-intersecting path.

This modification of the state is valid, because the length of the path doesn't change. However, we cannot untwist crossings between two different paths, because that could change their lengths. This means that we should take special care to not produce an initial state where two different paths cross. We can achieve this by calculating crowding penalties while creating the initial state. Then, when the random walk algorithm selects a random neighbor to move to, we make it prefer the neighbors with a lower crowding penalty. Because we don't mind self-intersections, we don't add the crowding penalties from the nodes of the path the algorithm is currently creating. We add them only when the path is complete.

Still, the algorithm sometimes fails in creating a valid path network. In case no valid network is generated, we can restart the path generation algorithm, including picking new starting positions.

In figure 3.5, we can see how the paths evolve over time as the temperature decreases.



(a) Initial state ($t_{initial} = 2.5$)

(b) $t = 2$

(c) $t = 1.5$

(d) Result ($t_{final} = 1$)

Figure 3.5 Evolution of paths during simulated annealing.

3.3.5 Simplifying the Relative Improvement Calculation

In paragraph Cost function of the previous section, we defined the cost function we will use and its components: The cost function for state s is defined as follows:

$$f(s) = \sum_{t \in s} \left(c_e(t) + \sum_{u \in s} c(t, u) \right)$$

Where $c(t, u)$ is the crowding penalty that tile t gets from its distance to tile u, and $c_e(t)$ is the crowding penalty tile t gets from its distance to the edge of the world. The total crowding penalty from tiles in state s for a given tile t is then $c_t(t, s) = \sum_{u \in s} c(t, u)$. We also stated that we will only calculate the relative improvement $r_i = f(s) - f(s')$ between the current state s and the new state s'.

In this section, we show that the relative improvement $r_i = f(s) - f(s')$ for s' obtained by changing tile v in s to w can be calculated as follows:

$$r_i = c_e(v) - c_e(w) + 2c_t(v, s) - 2c_t(w, s) + 2c(w, v) - 2$$

We start by substituting the definition for the cost function f.

$$\begin{aligned} r_i &= f(s) - f(s') \\ &= \sum_{t \in s} \left(c_e(t) + \sum_{u \in s} c(t,u) \right) - \sum_{t \in s'} \left(c_e(t) + \sum_{u \in s'} c(t,u) \right) \end{aligned}$$

Now we separate v from s and w from s'.

$$\begin{aligned} &= \sum_{t \in s} \left(c_e(t) + c(t,v) + \sum_{u \in s-v} c(t,u) \right) - \sum_{t \in s'} \left(c_e(t) + c(t,w) + \sum_{u \in s'-w} c(t,u) \right) \\ &= \left(c_e(v) + c(v,v) + \sum_{u \in s-v} c(v,u) \right) + \sum_{t \in s-v} \left(c_e(t) + c(t,v) + \sum_{u \in s-v} c(t,u) \right) \\ &- \left(c_e(w) + c(w,w) + \sum_{u \in s'-w} c(w,u) + \sum_{t \in s'-w} \left(c_e(t) + c(t,w) + \sum_{u \in s'-w} c(t,u) \right) \right) \end{aligned}$$

We use that $s - v = s' - w$, and we name q the set of tiles both states have in common.

$$\begin{aligned} &= c_e(v) + c(v,v) + \sum_{u \in q} c(v,u) + \sum_{t \in q} \left(c_e(t) + c(t,v) + \sum_{u \in q} c(t,u) \right) \\ &- \left(c_e(w) + c(w,w) + \sum_{u \in q} c(w,u) + \sum_{t \in q} \left(c_e(t) + c(t,w) + \sum_{u \in q} c(t,u) \right) \right) \\ &= c_e(v) + c(v,v) + \sum_{u \in q} c(v,u) + \sum_{t \in q} c_e(t) + \sum_{t \in q} c(t,v) + \sum_{t \in q} \sum_{u \in q} c(t,u) \\ &- \left(c_e(w) + c(w,w) + \sum_{u \in q} c(w,u) + \sum_{t \in q} c_e(t) + \sum_{t \in q} c(t,w) + \sum_{t \in q} \sum_{u \in q} c(t,u) \right) \\ &= c_e(v) + c(v,v) + \sum_{u \in q} c(v,u) + \sum_{t \in q} c(t,v) \\ &- \left(c_e(w) + c(w,w) + \sum_{u \in q} c(w,u) + \sum_{t \in q} c(t,w) \right) \end{aligned}$$

The function c is symmetric, because it depends only on the distance between the tiles, which is symmetric, so we get:

$$= c_e(v) + c(v,v) + 2 \sum_{t \in q} c(v,t) - \left(c_e(w) + c(w,w) + 2 \sum_{t \in q} c(w,t) \right)$$

The crowding a tile inflicts to itself is 1, so $c(v,v) = c(w,w)$.

$$= c_e(v) - c_e(w) + 2 \sum_{t \in q} c(v,t) - 2 \sum_{t \in q} c(w,t)$$

We can add the terms $2c(v,v) - 2c(v,v) + 2c(w,v) - 2c(w,v)$ because they sum to zero:

$$\begin{aligned} &= c_e(v) - c_e(w) + 2 \sum_{t \in q} c(v,t) - 2 \sum_{t \in q} c(w,t) \\ &+ 2c(v,v) - 2c(v,v) + 2c(w,v) - 2c(w,v) \end{aligned}$$

And we collect the sums to sum over the elements of s.

$$
\begin{aligned}
&= c_e(v) - c_e(w) + 2\sum_{t \in s} c(v,t) - 2\sum_{t \in s} c(w,t) - 2c(v,v) + 2c(w,v) \\
&= c_e(v) - c_e(w) + 2c_t(v,s) - 2c_t(w,s) - 2c(v,v) + 2c(w,v) \\
&= c_e(v) - c_e(w) + 2c_t(v,s) - 2c_t(w,s) + 2c(w,v) - 2
\end{aligned}
$$

□

So we see that the equality holds, letting us compute the relative improvement more efficiently.

3.3.6 Final Paths

Now that we have generated the main paths, we just have to generate the side branches. The world generation steps that come after have to make sure to not block the paths we have generated. However, we don't want to constrain them with the whole path network. Since we don't have requirements on the minimum side branch count or their lengths, it is enough to ensure the paths we generated in the previous step get preserved. We can generate the rest of the world first, and only then make extra paths where they fit. We don't want the branching to feel the same in every level. This lets us use the randomness of the world generation instead of needing to introduce more in this stage.

Due to this decision, this step will start with an already generated terrain and obstacles, as described in sections 3.4 and 3.5. These steps respect the original paths, but they will cause other tiles or edges between them to be blocked, as shown in figure 3.6. Here, we can see the original paths on the left. On the right, we can see tiles blocked by obstacles as gray squares with black edges. Additionally, some edges between tiles are also blocked, usually because the two tiles are at different height levels, separated by a cliff. These are the remaining black lines.

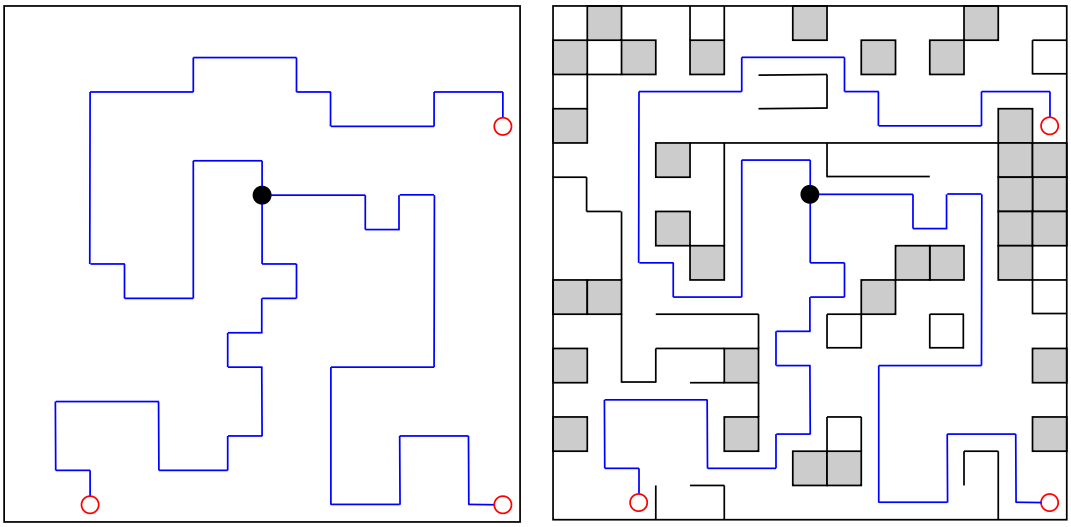


Figure 3.6 Blocked edges and tiles after generating the terrain and obstacles.

Now, how do we actually generate the path network? There are so many blocked edges that we want to add all the valid branches we find, up to the

maximum number of branches decided by the map generator (see section 3.2). To do this, we will use depth first search, but we will add a few heuristics to produce better paths. To decide on these heuristics, we will set a few more requirements: We still want the paths to be spread out, but we would prefer the paths to go straight if possible.

However, we would also like to somehow capture the shape of the paths that were optimized during the previous stage. We could simply make the first path from each start be identical to the original path. We would like multiple paths to sometimes join together, which we cannot do without changing them. To achieve this, we add a rule for every tile with a path: if an original path also went through it, its path distance must be the same as the original path distance. This way, the path segments of any new path will be distributed similarly to the original path, since they have to cross at every choke point. This is illustrated in figure 3.7. On the left are the original paths, and on the right are randomly generated paths which respect the rule. The tiles where the random paths would intersect the original paths are marked with a blue circle.

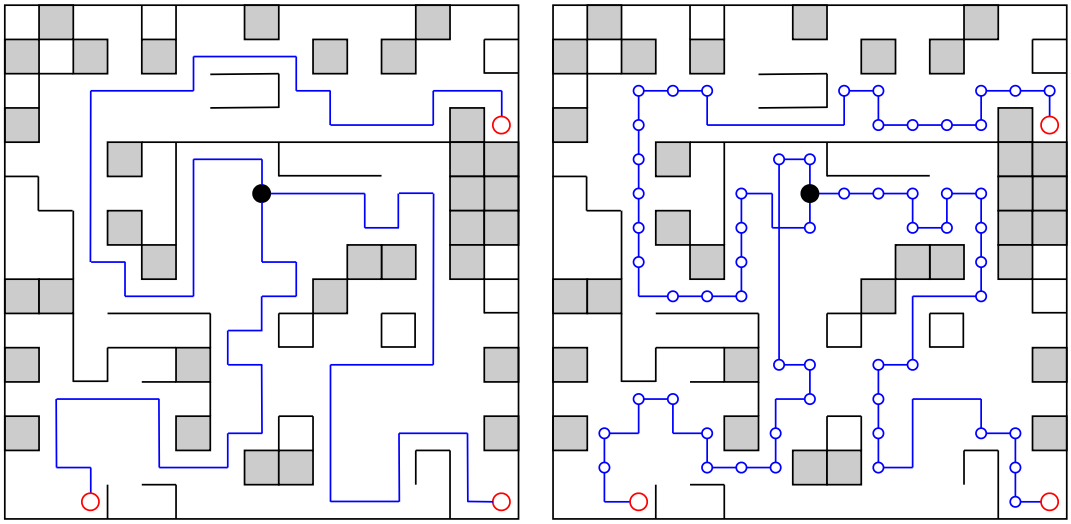


Figure 3.7 Randomly generated paths which respect the original path distances.

We need to make sure the paths we produce are the correct length and respect the required original path distances. We can save a lot of time by precomputing the minimum distance each tile can have. This way the algorithm can avoid tiles from which it's impossible to finish a path of the correct length. For this, we can do a breadth first search from the Hub tile, and we restrict the original path tiles to only "be found" at the right time and not sooner. The result can be seen in figure 3.8. We will call these distances the *minimum distances*.

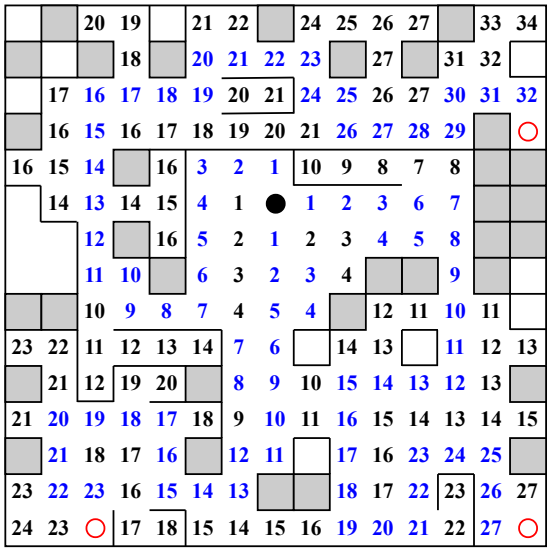


Figure 3.8 Calculated minimum distances.

Now we finally run the depth first search, one path by one. The algorithm keeps a stack of *path prototypes*. These are paths, which start at the start tile, but have not reached the Hub yet. At the start, the stack contains only the path prototype with only the start position as its only element. At every step, the algorithm pops the last prototype from the stack, finds all valid one-step continuations of the current paths, and adds those to the stack.

Valid continuations are the neighbors of the last tile of the current path prototype that aren't blocked. Additionally, they have to have a minimum distance that is less than or equal to the remaining length of this path prototype. The valid continuations are ordered such that the best option will be put onto the stack last, so it gets popped in the next step. The best option is always the one that goes straight, and the rest is ordered by the same crowding penalty as in paragraph Cost function of section 3.3.4.

Once the algorithm reaches the Hub or an already existing path, it checks whether it is valid to finish the current path. First, it must have the correct remaining length to produce a valid path. When it connects to the Hub, the remaining length must be 0, and for connecting to an existing path, the remaining length must match the path distance of that tile. Additionally, as per requirement RP10: "every branch must go through at least one tile that is not adjacent to any already existing path".

If this check fails, the algorithm does not mark this branch, it pops the last item from the stack and continues from there. If the check succeeds, it marks the new path section and updates the crowding penalties to take the new path section into account. Then it continues by making another branch.

However, the last path prototype on the stack will share most of its tiles with the branch we just found. We want the branches be separated from each other as much as possible. To minimize the shared section, we take the path prototype from the opposite end of the stack. So, in reality, we won't use a stack, but a double ended queue. The entire search is described in pseudocode as algorithm 2.

Algorithm 2 Finalizing paths

```
1:  for each path start start do
2:      stack.PushLast(path prototype containing only start)
3:      success ← false

4:      while stack is not empty do
5:          if success then
6:              p ← stack.PopFirst
7:          else
8:              p ← stack.PopLast
9:          end if

10:         if last tile in p contains a path or the Hub then
11:             success ← TryFinishPath(p)
12:         else
13:             success ← false
14:             for each tile c from GetValidContinuations(p) do
15:                 stack.PushLast(p extended by c)
16:             end for
17:         end if
18:     end while
19: end for
```

With this ordering of valid continuations, the algorithm usually reaches the Hub too soon, and then it has to backtrack many times before producing a path that is the right length. To fix this, we prioritize above all the tiles with minimum distance exactly equal to the remaining length. The results of the algorithm are displayed on the right in figure 3.9, compared to the initially generated paths on the left. Segments that have changed are highlighted in red.

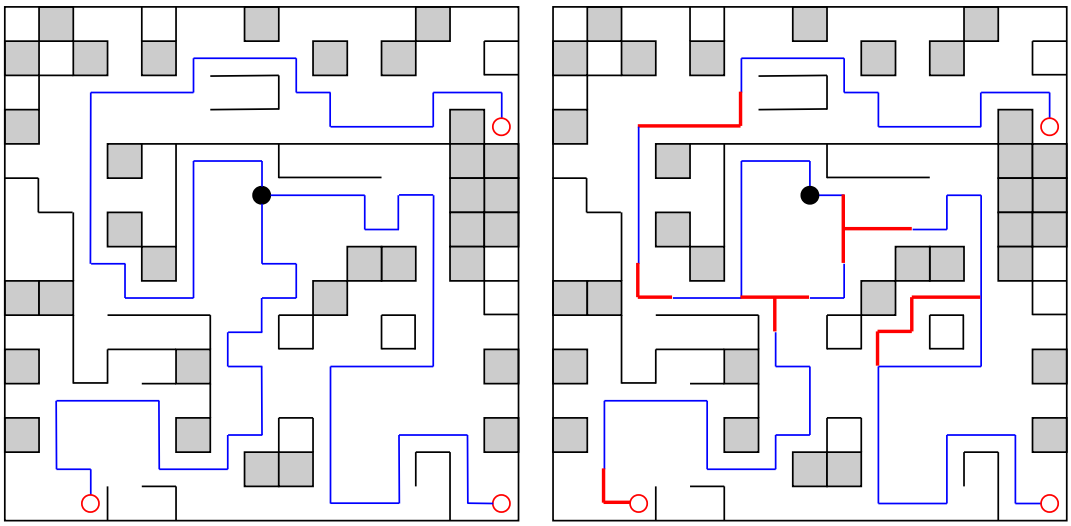


Figure 3.9 Final paths compared to the originally generated paths.

3.4 Terrain Generation

There are many techniques we could use for generating the terrain. However, we have pretty strict requirements for the terrain generation. We need to make sure the paths we have generated in the previous step are not blocked by terrain features like cliffs. We also want the algorithm to be able to generate different-looking terrain types.

This led us to a variant of a procedural generation algorithm called *model synthesis*, originally developed by Merrell [19]. The discrete version of this algorithm is better known by the name *wave function collapse* (or WFC in short), popularized by Gumin on GitHub [20]. Model synthesis is more general and focuses more on 3D models, whereas WFC applies the same concepts to generating 2D pixel art and tile maps. Since the name "wave function collapse" is more popular, we will use it in the rest of this thesis, even though it's not the name of the algorithm that came first.

We chose WFC, because it can generate randomized terrain, whilst fully respecting the initial constraints we give it. To see how this works, we will explain the algorithm first.

3.4.1 Wave Function Collapse

The original intent behind the algorithm is to replicate the structure of an example on a larger scale, making sure that the output is locally similar to the input, as shown in figure 3.10. We will limit our examples to 2-dimensional grids of tiles, however this algorithm works in more dimensions.

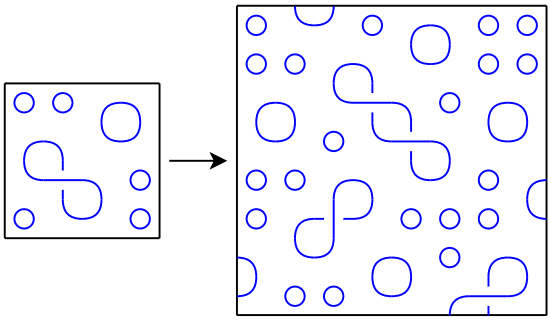


Figure 3.10 Example input and output of the wave function collapse algorithm.

The first step of the algorithm is to extract from the input which features can appear next to each other. The algorithm creates a set of *modules*[1], which are the building blocks the output will be built from. Each module comes with a set of constraints on its neighbors. The main portion of the algorithm then builds the output from these modules, such that all the constraints are satisfied, and each module appears in the output with a similar frequency to the input. However, we will create the modules for our generator by hand, including their constraints, in order to have greater control over the generated result. In figure 3.11, we can see a set of 7 modules and the resulting output, given only the constraint that the edges of directly adjacent modules must match.

[1]This is a naming convention used in an article by Marian [21].

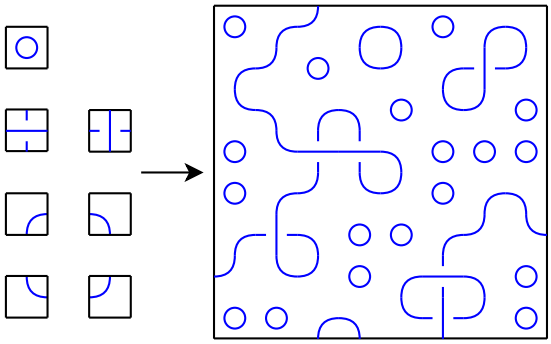
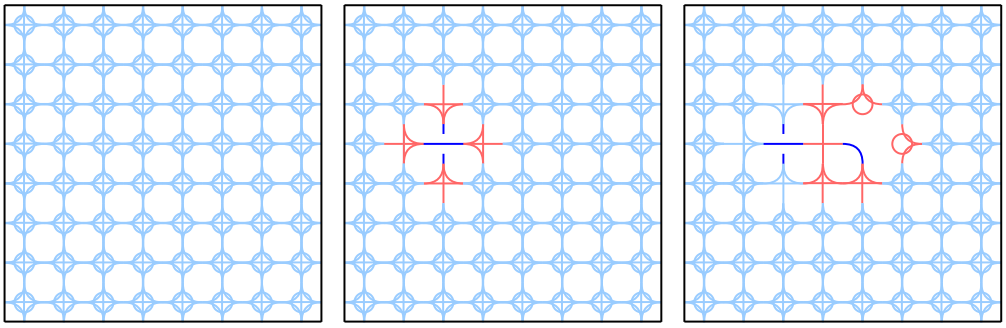


Figure 3.11 Example output of WFC, using the modules on the left and only the constraint that their edges must match.

We call each spot in the output where a module is supposed to be a *slot*. Each slot keeps track of all the modules that can be placed in it. At the start of the main part of the algorithm, all slots are initialized with all the modules. Figure 3.12a shows a visualization of this state. Then the algorithm repeats two actions: collapse a slot, propagate constraints. To collapse a slot, the algorithm removes all possible modules from the slot except for one, chosen at random.

Then it has to propagate constraints, which means that it removes from each slot all the modules which can no longer be placed there. For example, in figure 3.12b we see that a slot has collapsed to a module which has a line on each edge. Thus, the algorithm removes from the neighboring slots (marked in red) all modules which don't have a line at the corresponding edge. After propagating constraints, the algorithm collapses another slot and so on.



(a) Initial state **(b)** First step **(c)** Second step

Figure 3.12 Two steps of wave function collapse.

In figure 3.12c, we can see an interesting situation after collapsing a second slot. The slot between the collapsed slots can still contain two possible modules, however both of them have a line at the top and bottom edges. This means that the algorithm also has to propagate to the neighbors of this tile that they have to have lines at the corresponding edges. A change in one slot can affect slots even very far away from it.

This process repeats until all slots are collapsed, at which point we have successfully generated the output. This process is summarized as algorithm 3. We left out one detail: which element does POP select? This does not matter for the overall function of the algorithm, and it will be further discussed in section 3.4.3. We will call this algorithm WFC, even though it differs from both WFC by Gumin

and Merrell's model synthesis. The only notable difference is that we skip the feature extraction step, and the algorithm takes as input the modules directly.

Algorithm 3 A naïve version of wave function collapse

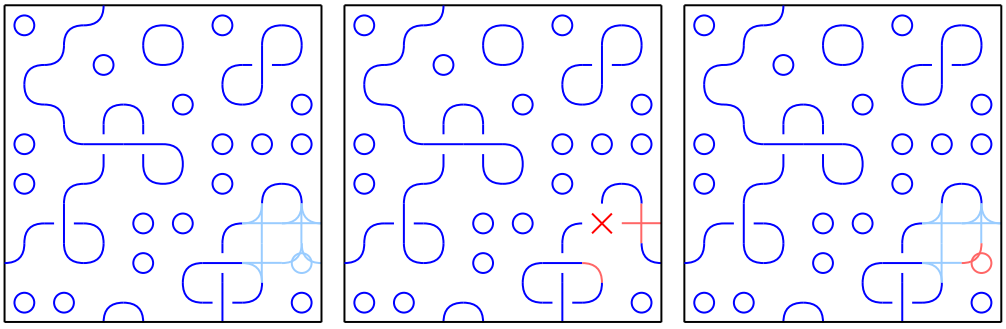
```
1: for each slot s in output do                          ▷ Initialize all slots.
2:     s.modules ← all_modules
3: end for

4: uncollapsed ← all slots
5: while uncollapsed is not empty do
6:     s ← uncollapsed.Pop                                ▷ Collapse a slot.
7:     s.modules ← {random module from s.modules}

8:     to_update ← neighbors of s                         ▷ Propagate constraints.
9:     while to_update is not empty do
10:        u ← to_update.Pop

11:        changed ← false
12:        for each module m in u.modules do              ▷ Remove invalid modules.
13:            if not IsValid(m) then
14:                u.modules ← u.modules − m
15:                changed ← true
16:            end if
17:        end for

18:        if changed then                  ▷ If u changed, enqueue its neighbors.
19:            to_update ← to_update ∪ neighbors of u
20:        end if

21:    end while
22: end while
```

However, it is possible for the algorithm to create a slot with no valid module. In that case, it is no longer possible to create a valid output. We call this situation a *conflict*. An example can be seen in figure 3.13. If we look at WFC as a *constraint satisfaction problem* solver, we can see that the constraint propagation only ensures *arc-consistency*, which is not enough to rule out conflicts. This is concisely explained in the Wikipedia article on local consistency [22].

We called algorithm 3 naïve, because it is unable to deal with any conflicts. What can we do to always produce a result, even when a conflict happens? One option is to simply restart the algorithm. For sufficiently small outputs, conflicts should be rare enough, only needing a few restarts.

Another option is to use backtracking. Whenever the algorithm runs into a contradiction after collapsing a slot, it returns to the state before collapsing. Additionally, it removes the module the slot collapsed to from its valid options, because it now knows it causes to a conflict. The state after backtracking is illustrated in fig 3.13c. This way, the algorithm can continue generating without

getting rid of all its progress.



(a) Before collapsing **(b)** After collapsing **(c)** After backtracking

Figure 3.13 Conflicts and backtracking in WFC. The red cross marks a slot with no valid modules.

3.4.2 Advantages and Disadvantages of WFC

The main advantage of WFC is that it offers a lot of control over the generated world. That's ultimately why we chose to use it. We always know what features can appear in the world, because we explicitly select them as building blocks of the terrain. We can also freely constrain the world that we are generating. For example, we can force the generator to not block the paths we've generated in the previous stage, and we can force the tile with the Hub to be flat. We also force a random tile to be at the lowest height level and another to be at the highest.

However, WFC has some disadvantages when used as a terrain generator. First, it is very slow. Even the naïve implementation has to do a lot of work for each slot in the world. However, the real problem are the conflicts. Merrell shows in section 3.3.5 of their thesis that deciding whether an incomplete output is consistent, i.e., it can be completed without running into contradictions, is an NP-complete problem. This necessarily means that WFC cannot run in polynomial time for all inputs, assuming $P \neq NP$.

The real problem is that the algorithm usually does run into contradictions, and the larger the generation task, the more likely it is to run into a contradiction. This is especially bad for online generation of infinite worlds, because we can't simply restart and generate a new world after the player has already seen a part of it. Backtracking also doesn't solve this issue, because the algorithm can collapse a slot in a way that is guaranteed to cause a contradiction, and then do arbitrarily many more steps before finally running into it. Of course, there are ways to circumvent this issue, namely by making the individual generation tasks smaller. In section 3.3.6 of their thesis, Merrell describes a technique called *modifying in parts* based on this approach. Luckily, this is not a problem for us, because our world is very small.

Another potential problem with WFC is that the individual slots or modules can be very apparent and repetitive. This can be solved by procedurally generating the resulting geometry the player sees, only based on the modules chosen by WFC. Another way to make the slots less apparent is to make the slots irregular. Both of these techniques are used by *Townscaper* [23], a game by Oskar Stålberg. This also isn't a problem for us: we don't mind that the tiles will be apparent, since the gameplay of our game is centered around them anyway

Also, WFC only uses local constraints, so it provides no control over the more global features of the output. On a large scale, the results are very homogenous. For larger outputs, WFC should only be used to generate the local features, guided by large-scale features generated by some other algorithm. Our game world is also too small for this to matter.

3.4.3 Using WFC for Terrain Generation

Even though we want to generate a 3D terrain, our output will consist of a 2D grid of slots. We want the tiles of the generated world to be at different heights, however, we don't want any tiles to generate above other tiles. Ultimately, this is a 2D generation task, with the addition that modules can appear at different height levels.

Slots and Modules

At first, it might seem sensible to have one slot per world tile. However, each tile on its own will be mostly a flat square. The interesting terrain features will appear on the boundaries between the tiles. For example, two tiles at different height levels next to each other will have a cliff separating them. If we wanted to incorporate the cliff into the tile module, which tile does it belong to? What about the features where the corners of four tiles meet?

We offset the slots in a way shown in figure 3.14, such that each slot is responsible for four quarter-tiles of the world. Tiles are drawn in black, slots in red. This way, the modules dictate how can the adjacent tiles connect to each other.

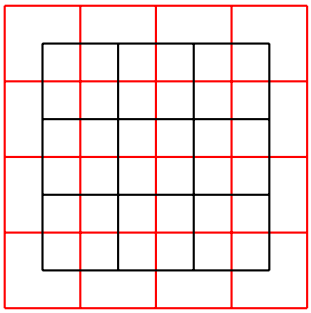
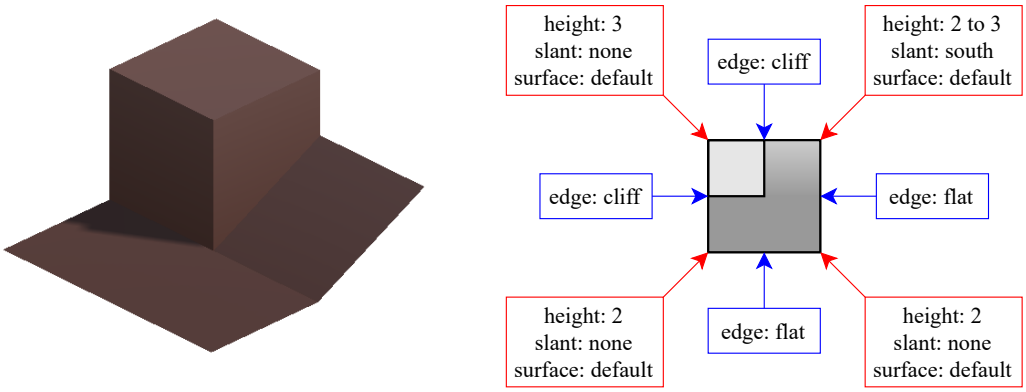


Figure 3.14 The slots for generating a 3×3 tile world.

One example of such module is shown in figure 3.15. Each module constrains the 8 adjacent slots. An edge type is specified for each edge, and modules which share an edge must have the same edge type. For each corner of the module, several tile constraints are specified. The modules that share a tile must agree on the tile's properties: its height, slant direction (if any) and surface type. Terrain types can have multiple surface types, each with a different set of modules and a few modules that allow to transition between them. For example, a *shore* module could have *ground* tiles on one side and *water* tiles on the other. Some surface types block paths and buildings and some edge types block paths. What modules are available for the generator will be determined by the terrain type that was chosen for this world.



(a) The 3D geometry of the module.

(b) A view from above with the constraints.

Figure 3.15 An example module and its constraints.

In figure 3.11, we show 7 modules as an input. However, when designing them, it would make more sense to think of them as 3 different modules that can each be rotated. Thus, the modules we use will also have an option to select the allowed reflection and rotations. Then, before generating the terrain, we will automatically generate all variants of each module. Each module can also be placed at different height levels, which is handled similarly. For example, the module shown in figure 3.15 will effectively become 24 different modules in a terrain type with 4 height levels (0, 1, 2, 3), because it has 8 different reflections and rotations, and it can appear at 3 different height levels (0–1, 1–2, 2–3).

For each module, we also specify a weight. This dictates how likely it is to be selected when collapsing a slot, compared to other modules. For example, when we collapse a slot that only has two valid modules, with weights 1 and 4, then the first module will be selected with a 20 % probability.

Backtracking

We have decided to implement backtracking to solve contradictions. If the algorithm only ever has to backtrack once before collapsing another slot, we say that it needs backtracking depth 1. However, it is possible that the algorithm collapses a slot, finds a contradiction, and after backtracking, removes the only remaining module in the slot that was collapsed. Thus, it creates another contradiction, which causes it to backtrack deeper.

We ran some quick tests with the set of modules which is going to be used to generate terrain in the demo version. The results are shown in table 3.1. For each test, we tried generating 2000 random worlds, each with three path starts with lengths 24, 28 and 33, making them pretty constrained.

Maximum backtracking depth	Fails	Successes	Success rate	Time (s)
0	2000	0	100.00 %	644
1	786	1214	60.70 %	740
2	783	1217	60.85 %	797
10	783	1217	60.85 %	819

Table 3.1 Success rate of terrain generation based on maximum backtracking depth.

No world was able to generate without running into a contradiction, i.e., with a backtracking depth 0. 60.7 % of the worlds only ever needed backtracking depth of 1 to successfully generate. Greater backtracking depth doesn't seem to help very much. Instead, it makes the generator spend more time trying to save worlds which are still going to fail. We decided to allow for backtracking depth 1 and otherwise just restart the generation to try again.

Deciding Which Slot to Collapse

We also need to decide which slot to collapse at each step of the algorithm. Merrell uses in their thesis [19] a scan line order, collapsing slots in the lexicographic order of their coordinates. Gumin in their implementation [20] always collapses a slot with the lowest *Shannon entropy*. This causes the generation to collapse slots outwards form the slot that was collapsed first. From our testing, the results tend to look unnatural, often creating regions at the same height, or repeating patterns.

Both approaches grow the collapsed portion of the world from one initial slot, similar to growing a single crystal solid from one seed crystal. To make the results more natural, we chose to select the slot to collapse at random. This leads to the generator first collapsing slots in various parts of the world to different configurations. Then it has to somehow connect these to make the world follow the rules we set. This leads to much more diverse results, however, the success rate becomes very low, leading to slower generation.

As a compromise, we tried to still select the slots randomly, but give slots with lower entropy a greater weight, so they are more likely to be selected. Specifically, the weight of each slot is $1/entropy$, making it more likely to select more constrained slots, but still collapsing unconstrained slots once in a while. This leads to results similar to the ones with uniform randomness, but it increases the success rate. However, this results in the algorithm being even slower, probably because calculating the entropy of a slot is not trivial.

To remedy this, we tried weighting the slots by the number of invalid slots $+1$ (the $+1$ at is there to make all weights non-zero). This is an approximation of the previous metric, still assigning greater weight to more constrained slots. From our testing, the resulting worlds look still as good as before, but this metric is trivial to compute, so the worlds generate slightly faster.

The tests we ran are summarized in table 3.2. For each method, we generated worlds until we successfully generated 1000 of them. This is to compare the average time it takes to successfully generate a world, which is a more useful metric for us.

Which slot to collapse?	Fails	Successes	Success rate	Time (s)
minimum entropy	213	1000	82.44 %	492
uniformly random	823	1000	54.85 %	691
weighted by 1/entropy	672	1000	59.81 %	728
weighted by invalid modules	676	1000	59.67 %	656

Table 3.2 Success rate of terrain generation based on how the algorithm decides which slot to collapse next.

In the end, we chose to select slots to collapse randomly, weighted by the number of invalid modules. This method is the fastest of the methods that produce worlds that look random. In figure 3.16, we show an example of terrain that was generated this way.

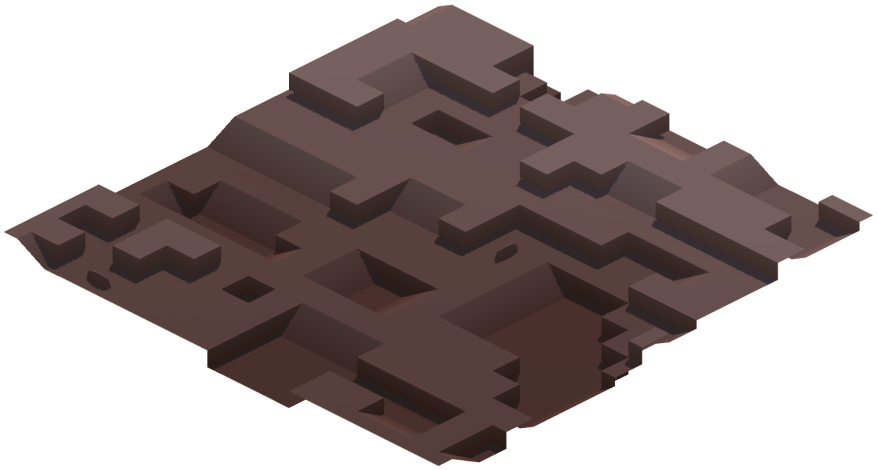


Figure 3.16 An example terrain generated using WFC.

3.5 Obstacle Generation

Now that we have generated the attacker paths and terrain, the next step is to generate obstacles. We know from section 2.3.2, that each tile can have up to one obstacle placed on it. They block the tiles, so they cannot appear on path tiles. Which obstacles can appear, and their placement is controlled by the terrain type. Before we select an algorithm to place them, we should formalize the parameters which influence the obstacle placement.

3.5.1 Obstacle Placement Parameters

In section 2.3.2, we also specified that some obstacles will be more common than others. In the terrain type specification, we can assign each obstacle a probability of it appearing on each tile. Some obstacles, for example the *mineral-rich* and *fuel-rich* ones will also have a maximum and minimum count.

Some obstacles will appear more often or less often around other obstacles. This means that an obstacle's probability to be placed on a given tile will be

influenced by the obstacles already placed on surrounding tiles. For each obstacle type, the terrain type can specify its *affinity* for each other obstacle type. When trying to place the obstacle on a given tile, its placement probability will be modified for each already placed obstacle by the affinity for that obstacle times the *closeness* to it. The closeness is some function that decreases with distance, similar to the *crowding penalty* we used in section 3.3.4 when generating paths.

Since each tile can only have one obstacle, the more important obstacles should be placed before others. For example, we want the resource providing obstacles to be placed first. So, the obstacle types will be divided into phases, and the first phase will be placed first, then the second, and so on.

In section 3.4.3, we mentioned that each tile will have a surface type from the set of surface types the terrain type allows. We also want to limit on which surfaces can each obstacle appear. For example, we don't want trees to appear in water. This should be all we need to procedurally place obstacles how we want.

3.5.2 Obstacle Placement Algorithm

Placing the obstacles should be fairly straightforward given the parameters we decided on. The algorithm should place them phase by phase. For each phase, it will loop through all tiles, and for each tile, it will assign an obstacle to it with the probability that is calculated for that obstacle for that tile. The probability is 0 for invalid placements, like path tiles, tiles with another obstacle, or tiles with the wrong surface. When multiple obstacles from the same phase want to be placed on the same tile, we just select one of them at random.

However, we also have a minimum and maximum count for some obstacles. To adhere to the maximum count, we can just stop placing the obstacle once the maximum count is reached. This creates a small problem. If the algorithm went through the tiles in some predefined order, the tiles at the start would have a higher chance of containing obstacles with a count limit than the tiles at the end. Because once the algorithm reaches the maximum count, it stops placing them. We can fix this by enumerating the tiles in a random order.

Enforcing the minimum count is also not difficult. If we finish a phase and there is not enough obstacles of some types, we can do another placement phase, this time only with the obstacle types which didn't reach their minimum yet. Assuming there is enough tiles with non-zero probability, we can repeat this process until the right amount obstacles is placed.

This is summarized as algorithm 4. Here, a phase is just a set of obstacle types. Here, the function GetPlacementProbability also checks if the placement would be valid. If not, it returns 0.

Algorithm 4 Obstacle placement

```
1: for each phase p do
2:     while p is not empty do

3:         for each tile t in random order do
4:             candidates ← ∅
5:             for each obstacle type o in p do
6:                 q ← GetPlacementProbability(o, t)
7:                 with probability q: candidates ← candidates + o
8:             end for

9:             if candidates is not empty then
10:                selected ← random obstacle type from candidates
11:                Place(selected, t)
12:            end if
13:        end for

14:        for each obstacle type o in p do
15:            if number of placed obstacles of type o ≥ o.minimum then
16:                p ← p − o
17:            end if
18:        end for

19:    end while
20: end for
```

Of course, there are some optimizations we can make. For example, we don't have to go through all tiles, only those that don't have an obstacle and don't have a path going through them.

3.5.3 Generating Obstacle Models

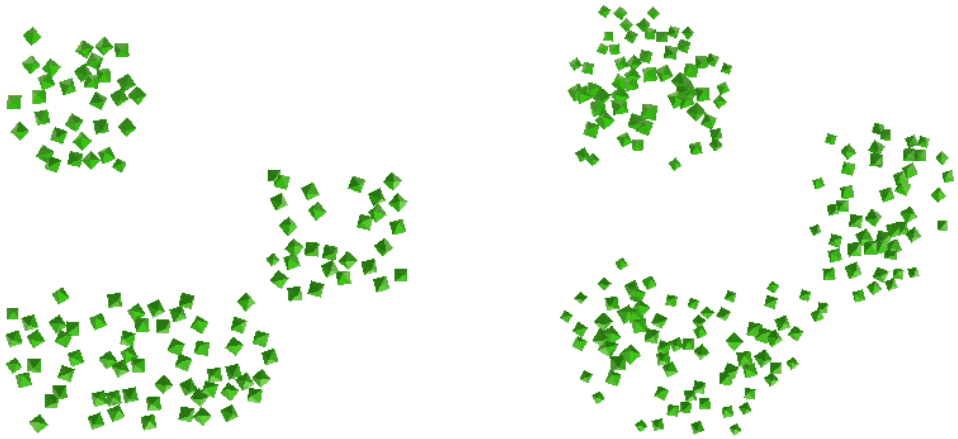
We can now determine which tiles are blocked by which obstacles. But by requirement RW3, "each tile with an obstacle will usually contain a whole procedurally generated cluster of obstacle models". How do we generate these models?

For any given obstacle type, we can through all the tiles with that obstacle type and place some number of the corresponding model at random points within the tile. We can also slightly randomize the rotation and scale of the models to achieve a more natural look. The terrain type then specifies the number of models per tile and the scale and rotation range.

Just placing the models at random points often creates clusters of intersecting models that don't look nice. So, we let the terrain type set some minimum distance the models should keep from each other. If it would place a model at a position that's too close to an already placed model, it doesn't place it.

However, this still doesn't feel very natural because the models create squares with hard edges, as shown in figure 3.17a. We would like the boundaries to be

more natural, and even make the models smaller and more spread out near the edges of the boundary. This is illustrated in figure 3.17b.



(a) Undesirable result with hard edges.

(b) More desirable result with smooth boundaries.

Figure 3.17 Comparison of generated obstacle models.

To place the models in a more natural fashion, we can use what the developers of the game *Factorio* [24] call *noise expressions*. Earendel has written a great overview of noise expressions on their blog [25]. The main idea is that we create a function that assigns a numeric value to each point in our world. For our use-case, the value represents "how much of an obstacle" is at any given point. We can then place the obstacle models only where the value is above some threshold, and we scale the models and their spacing by the value too.

We will specify a noise expression for each model in the terrain type. To do so, we will build up each noise expression from some basic functions, combined using arithmetic operations. First, we explain the basic functions we will use, and then we'll show how to combine them into a noise expression that is useful for us. In our examples, we will give each of the basic functions a pseudocode alias to let us express the operations on them better.

Height Function

A simple basic function we might want is a height function. This function simply tells us what is the height of the terrain at any given point. The output of this function is shown in figure 3.18a. This function can be useful for example when we want some model to appear only above some specific height. We denote this function in pseudocode as `height()`.

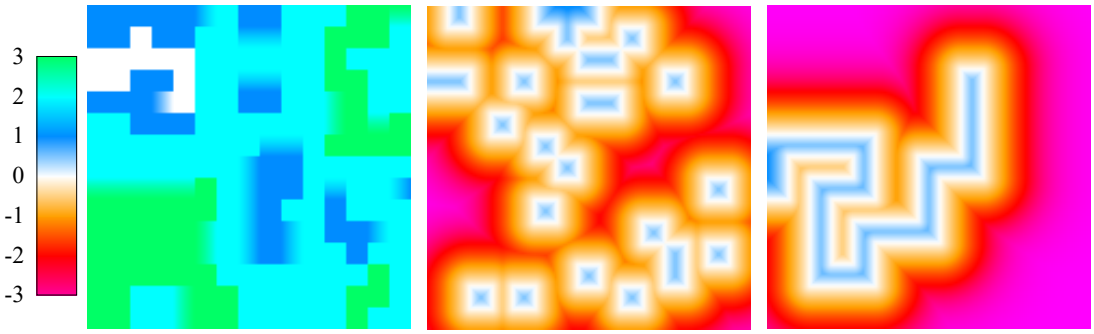
Signed Distance Functions

The most important basic function is the one that tells us which tiles have on them the obstacle we care about. After all, we want to place the models on tiles with the obstacle. We could make a basic function that returns 1 for points within the tiles with the obstacle and 0 otherwise. However, this doesn't help us make smooth transitions, because we cannot distinguish points that are just outside

a tile with the obstacle from points that are far away. Similarly, we would also like to know how far inside the tile with the obstacle is any given point. So, we need a *signed distance function*, or *SDF*, a function which tells the distance to a boundary of some region, but is positive inside the region and negative outside.

We will express this function in pseudocode as `obstacle()`. For greater convenience, we will also specify the scale of the values the function outputs separately for the values inside the region and outside the region. We will denote these parameters in the pseudocode as `in` and `out`. In figure 3.18b, we can see an example of this SDF. The function takes on positive values on the tiles with the obstacle and negative values otherwise, as specified by the parameters.

We can use SDFs to also express the distance to other tiles that are somehow special. For example the tiles with a path going through them. We will denote this function as `path`. An example of this function is shown in figure 3.18c.



(a) Height function. **(b)** Obstacle SDF. **(c)** Path SDF.

Figure 3.18 The values of the height function and signed distance functions.

Noise Functions

The basic functions we defined so far are useful, but they still don't let us create shapes without straight boundaries. For that we can use a noise function. However, we don't want a function that just gives us some random value for each point, we need the noise to be smooth and change its value gradually as we move in space. For example *Perlin noise*, developed by Perlin and used in their paper *An image synthesizer* [26]. It is easy for us to use this function, since it is included in the Unity engine.

In figure 3.19, we show what values it produces. We denote it with the alias `noise`, and it can be scaled up or down to get larger or smaller features using the `scale` parameter. With a large scale, it might seem too smooth and artificial, while with a smaller scale, the values fluctuate too much. To solve this, we can use arithmetic to combine the values of multiple noise functions. By adding together more layers of the noise, each with a different scale and amplitude, we can create a result that has details but doesn't look as random.

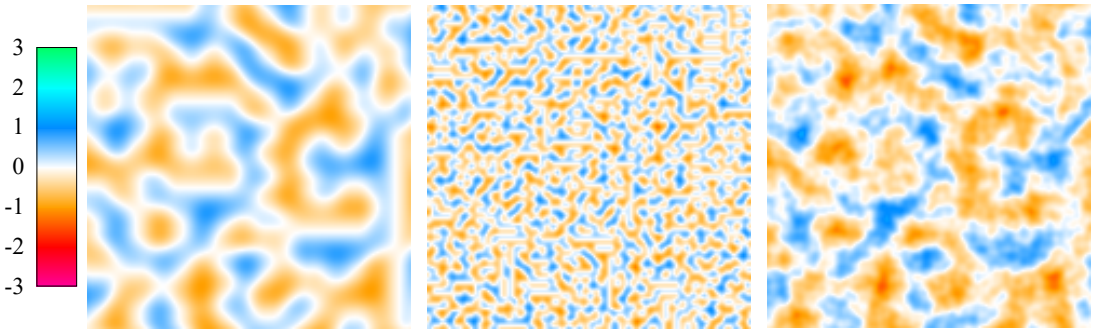


Figure 3.19 Examples of Perlin noise.

Putting it All Together

In figure 3.20, we show how we can build up a noise expression that we could use to place obstacle models. In addition to the basic functions and arithmetic operations, we also use a function called `clamp` which clamps a value between the specified minimum and maximum. We do this in three steps to show how the output of the noise expression changes with each addition. Each sub-figure represents one step, and later steps reference the noise expressions from the previous steps in their pseudocode, written as `STEP_#`.

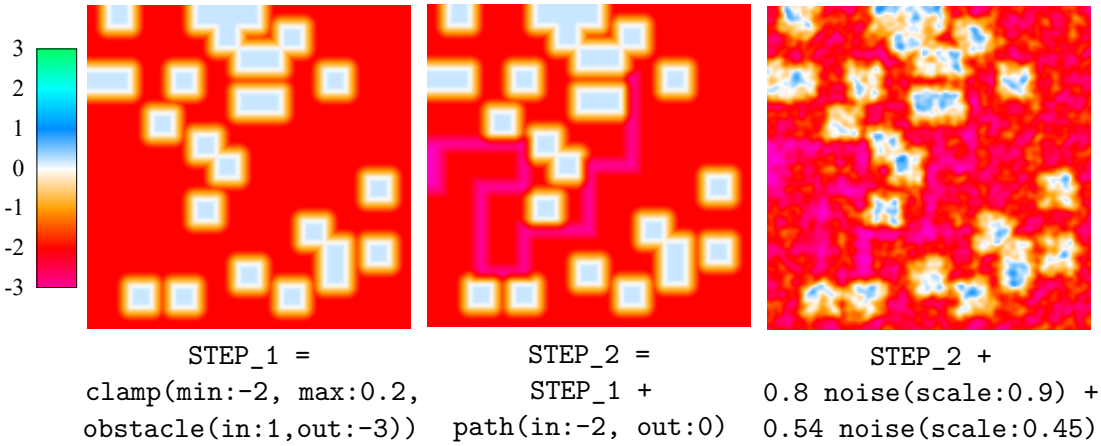


Figure 3.20 Building up a noise expression step by step.

When placing the models, we sample many points on each tile and place the model only if the value of the specified noise expression is for that point above some threshold. The value also determines how close together can the models be and how large they should be. This lets us be more expressive with model placement and create more natural-looking worlds. Using the noise expression from figure 3.20, we have generated the models shown in figure 3.17b.

3.6 Attacker Wave Generation

Now that world generation is done, attacker waves are the last remaining part of battles we want to procedurally generate. In section 2.3.1 we have outlined that

each wave will be composed of batches, and each batch will contain potentially different attackers spawning on each path. However, the spacing and count will be shared by the whole batch. We also mentioned, that the waves need to scale in difficulty, ideally slightly faster than the player could realistically handle. In section 3.2 we also decided that this difficulty will be controlled by the map generator.

Our goal in this section will be to create an algorithm for generating these waves, such that their difficulty somewhat matches the requested difficulty. We were able to find some relevant sources about procedural wave generation in tower defense games, but they generate waves dynamically to adapt to the player's skill, or to find weaknesses in their defenses. For example, in the paper *A neat approach to wave generation in tower defense games* [27], its authors train a neural network to decide the composition of a wave based on the player's defenses, with the goal to increase engagement. However, we want the waves to be only based on the requested difficulty. And, as stated in section 2.3.1, we let the player see them in advance, so the player is the one who adapts, not the waves.

There are also some already existing implementations. For example, even though the waves in the main game modes of *Bloons TD6* are always the same, there are also game modes with randomized waves. However, the nature of the algorithm used to generate these waves is not publicly known, and it probably wouldn't be useful us, because the generated waves vary in difficulty a lot. We were unable to find an example of a tower defense game with procedurally generated waves with a consistent difficulty.

To create a good algorithm that generates waves with a specific difficulty, we need to somehow quantify the difficulty of a wave first. In the following subsections we aim to find a way to express the difficulty of a wave, that is as simple as possible, but it is close enough for the waves to not feel unfair. It is important to stress that depending on the player's defenses, some waves will be always harder for them to deal with and others will be easier. However, if a player is unable to beat some wave, we want them to feel like there were ways to prevent this. In other words, that the wave was beatable, they just didn't play the best way they could.

As we discussed in section 3.2, the world on which the battle takes place greatly influences the difficulty. However, we will eliminate this variable and try to quantify the wave difficulty in vacuum. It will be the responsibility of the map generator to select the right wave difficulty based on the parameters it selected for the world.

Let's simplify our problem to as much as we can to make it easy to find an exact solution. Then we'll gradually add more complications until we get an approximation of the real problem that is good enough.

3.6.1 Model 1: Single Attacker

For our simplest model, we assume that a wave contains only one attacker with some amount of HP, denoted as h, and a speed s in tiles per second. The attacker has no other qualities or special abilities. We say the player's defense beats this wave when the attacker is killed before it reaches the Hub.

We also imagine each defensive tower t to always deal some fixed amount of

damage d_t per second to the first attacker in its range which covers some region of the path that is l_t long. This means that the attacker will spend l_t/s seconds in its range, taking $d_t \cdot l_t/s$ damage in total.

A defense T that consists of multiple towers can then deal damage equal to the sum of the individual towers. This means that the defense can beat the wave if and only if the following inequality holds:

$$\sum_{t \in T} d_t l_t / s \geq h \tag{3.1}$$

We will refer to the total damage the towers would deal to an attacker with speed 1 as *damage capacity* C. It can be calculated as follows:

$$C = \sum_{t \in T} d_t l_t \tag{3.2}$$

This lets us write the condition more concisely as $C \geq hs$.

From this equation we can see, that the difficulty of this wave is exactly proportional to hs. The defenses that can beat an attacker with 10 HP and speed 1 are exactly the same as those that can beat an attacker with 5 HP and speed 2.

This is a great result, but our waves can contain more attackers than one, so we have to generalize. Next, we will take a look at a case with an infinite amount of attackers.

3.6.2 Model 2: Infinite Waves

For this model, everything is the same as in the previous model, but each wave consists of infinitely many attackers of one attacker type. We will denote the spacing (or gap) between the attackers as g, measured in seconds. Here, we say the player's defense beats this wave when no attacker ever reaches the Hub.

The towers still deal damage only to the first attacker. Imagine the first attacker is killed after it has travelled p tiles. Assuming at least g seconds have passed, the second attacker has already spawned, and it has traveled $p - sg$ tiles, so the defenses have less time to kill it than they had for the first attacker. If the towers progressively kill the attackers later and later, eventually, one of them will reach the Hub. Intuitively, the towers need to be able to kill an attacker every g seconds to beat the wave.

When each attacker spends at least g seconds within the range of each tower, then each tower t deals in total $d_t g$ damage to each attacker. However, when the spacing is larger, some towers won't be able to deal damage to each attacker for g seconds, but only l_t/s as we determined in the previous model. So, each tower can only deal d_t damage per second for $\min(l_t/s, g)$ seconds. So, to each attacker, the towers will deal a total damage amount expressed by the following formula:

$$\sum_{t \in T} d_t \cdot \min(l_t/s, g) \tag{3.3}$$

It is more useful to think about the damage per second, so let R be the total damage per second the player's defenses can deal to the attackers, or the

damage rate. One attacker spawns each g seconds, so we can calculate the damage rate by dividing the previous formula by g:

$$R = \sum_{t \in T} d_t \cdot \min(l_t/(sg), 1) \tag{3.4}$$

We see that R is at most $\sum_{t \in T} d_t$, and it is equal to it only when $l_t/(sg) \geq 1$ for each tower t.

R is also at most $\sum_{t \in T} d_t \cdot l_t/(sg)$, and it is equal to it in the other cases. Since s and g are not dependent on the tower t, we can bring them in from of the sum:

$$\frac{1}{sg} \sum_{t \in T} d_t \cdot l_t/(sg) \tag{3.5}$$

This sum is by definition equal to C (see equation 3.2), so we can substitute to get the following relation between R and C:

$$R \leq \frac{C}{sg} \tag{3.6}$$

Since one attacker spawns every g seconds, the towers can deal at most Rg damage to each attacker. Each attacker has h HP, so the player will beat the wave if and only if $Rg \geq h$.

As described before, sometimes R does not depend only on the player's defenses, but also on the attacker speed s and spacing g. However, this should be very rare, because the maximum spacing we will use is $g = 2$ seconds, and no attacker will be faster than $s = 4$ tiles per second, which is still extremely fast compared to most attackers. This means that every tower has to have $l_t \geq 8$ for g and s to never influence R, which is easily achievable with most tower types. For now, we can think of R as being a constant determined by the player's defenses.

3.6.3 Model 3: Finite Waves

In this model, we add another parameter to our wave. The wave is once again composed of attackers with h HP and speed s, coming with a spacing of g seconds, but this time, there is exactly n of them. How do we judge the wave difficulty now?

The best way to think about this is that the player's towers need to deal nh damage in total. From model 1 we know that if the whole wave was a single attacker, the player's defenses would deal C/s damage in total. So, they would defeat the wave if and only if $C/s \geq nh$.

The towers still target only the first attacker, so they will kill it first. Then, they can deal with the second attacker, but the second attacker is g seconds behind the first, so they have g more seconds to deal with it. This is also true for each other attacker, so the towers effectively get g seconds of firing for free for each attacker beyond the first. Thus, a more accurate condition for beating the wave is the following inequality:

$$C/s + (n-1)Rg \geq nh$$

When we put all wave-dependent terms on the right and rearrange, we get that the player beats the wave if and only if the following condition is satisfied:

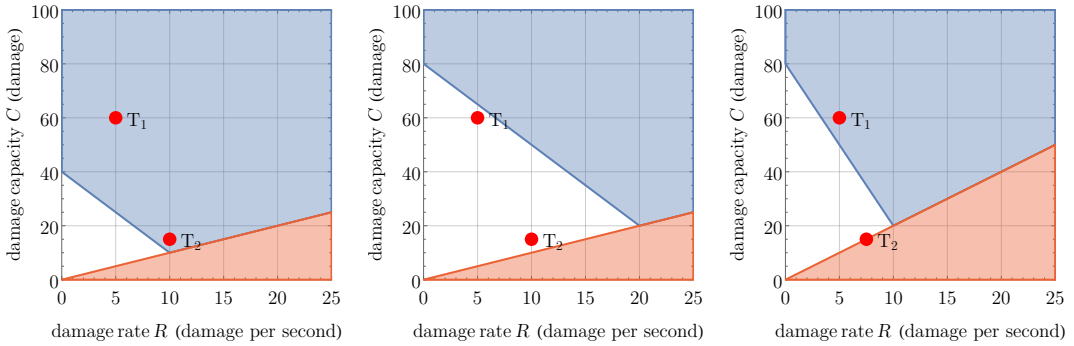
$$C \geq s(nh - (n-1)Rg) \tag{3.7}$$

We cannot quantify the difficulty of the wave with one number anymore. For example, there exists a defense T_1 which beats the wave, and another defense T_2 which also beats it despite having lower C, because it has greater R. So, we have to accept that each defense has two characteristic values.

We aim to illustrate how the required damage capacity and rate change based on the wave parameters using figure 3.21. The blue area represents the defenses which can beat the wave, and the white area represents those which can't. The orange area shows defenses that can't exist because they would violate inequality 3.6. We also highlight two points in each plot, each of these points represents one example defense. T_1 consists of one tower with $d_t = 5$ and $l_t = 12$. T_2 consists of one tower with $d_t = 10$ and $l_t = 1.5$ which is an unusually small path coverage.

Figure 3.21a shows a wave which we will use as a base case for our comparison. We can see that both T_1 and T_2 are able to beat it. In figure 3.21b we can see what happens when the attacker health becomes 20 HP. Neither of the defenses are able to beat it now. Figure 3.21c shows that increasing the speed s to 2 causes problems for towers with small path coverage l_t. We can see that the damage rate of defense T_2 has decreased because each attacker spends less than g seconds in the range of the only tower.

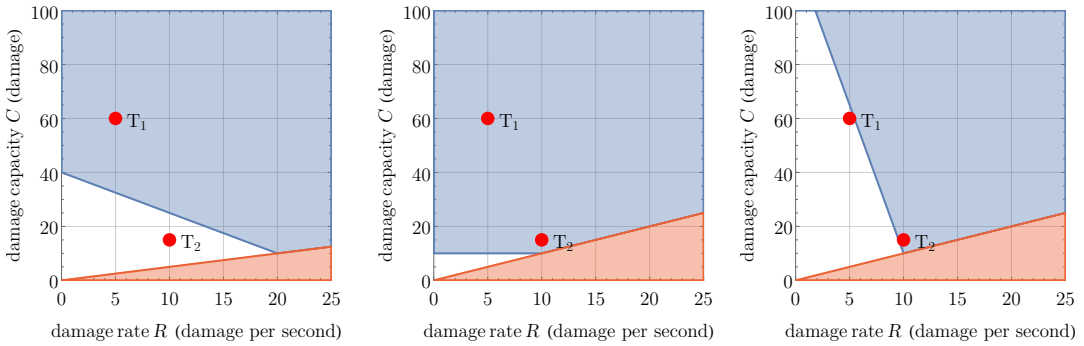
In figure 3.21d we can see that with smaller spacing of 0.5 seconds between the attackers, damage rate becomes less effective. A similar effect can be seen when we decrease the number of attackers. As shown in figure 3.21e, when the wave contains only one attacker, damage rate is irrelevant. On the other hand, when there are many attackers, the damage rate becomes much more important than capacity. In figure 3.21f we can see T_1 cannot beat a wave with 12 attackers, but T_2 can.



(a) $h = 10, s = 1,$ $g = 1, n = 4$

(b) $\boldsymbol{h = 20}, s = 1,$ $g = 1, n = 4$

(c) $h = 10, \boldsymbol{s = 2},$ $g = 1, n = 4$



(d) $h = 10, s = 1,$ $\boldsymbol{g = 0.5}, n = 4$ **(e)** $h = 10, s = 1,$ $g = 1, \boldsymbol{n = 1}$ **(f)** $h = 10, s = 1,$ $g = 1, \boldsymbol{n = 12}$

Figure 3.21 Which defenses can beat various attacker waves, according to model 3.

Thus, we need both parameters R and C to accurately describe which defenses can beat the wave and which cannot.

3.6.4 Model 4: Damage in an Area

Many towers will be able to deal damage to all attackers in some area. These towers are much more effective against large groups of attackers, especially when they are close together. We can model these towers similarly to what we have done so far, but we will add one more parameter, the damage range r_t. Now, each tower t deals d_t damage per second to the first attacker, but also all the attackers at most r_t tiles behind it. This still makes all calculations much more difficult, so we will choose to ignore many edge cases.

Intuitively, for the first attacker of every wave, nothing changes. To kill it, the condition $C/s \geq h$ still holds. Killing the second attacker is easier as it has already been damaged by collateral damage. The attackers spawn g seconds apart, so each can travel sg tiles before the next one spawns, making the gaps between them sg tiles long. So, the second attacker has been receiving collateral damage from all towers t with $r_t \geq sg$. But how much damage is that?

To simplify our calculations, we will introduce some new variables R_k. R_1 is the damage rate (see equation 3.4) of towers which hit one extra attacker with collateral damage, R_2 for towers which deal damage to two extra attackers, and so on. Formally,

$$\forall k \in \mathbb{N}^+ \text{ let } R_k \text{ be the damage rate of towers } t \text{ with } r_t \geq ksg.$$

Killing the first attacker will take $t_1 = h/R$ seconds as usual, and in the time, the second attacker will take $t_1 R_1$ collateral damage. In general, the kth attacker will take $t_1 R_{k-1}$ collateral damage from towers shooting at the first attacker. Let t_k be the time it takes for the kth attacker to be killed by towers continuously shooting at it. It is equal to the remaining health the attacker will have after the ones before it have been killed, divided by R:

$$t_k = \frac{1}{R}\left(h - \sum_{i=1}^{k-1} t_i R_{k-i}\right) \tag{3.8}$$

This formula isn't useful yet, because we still need too many variables to specify the player's defenses. However, this calculation can be simplified with the

right assumptions. The game should force the player use towers that deal damage in an area, in addition to those with a single target. So, when generating the waves, we can just assume the player has these towers. This in turn makes the player use them in order to beat the wave.

The greater the tower's damage range r_t, the less damage it should deal and the less often it should appear. Let's assume that in a balanced defense, the towers' damage rate decreases exponentially with increasing r_t.

> We say that a defense T is (α, β)-*balanced* for some $\alpha, \beta \in (0, 1)$ when $\forall x \in \mathbb{R}^+$ the total damage rate of towers $t \in T$ with $r_t \geq x$ is equal to $\alpha\beta^x R$.

We will select the right α and β based on playtesting. Since towers with larger damage range are generally going to be more expensive than single-target towers, we will make these parameters very small at the start of each level, only reaching their intended values after a few waves.

Now we can precisely determine all R_k for a given wave. From the definition, R_k is the damage rate of towers with $r_t \geq ksg$, so $R_k = \alpha\beta^{ksg}R$. We can substitute this into equation 3.8 to get the following recurrence relation:

$$t_k = \frac{1}{R}\left(h - \sum_{i=1}^{k-1} t_i \alpha\beta^{(k-i)sg} R\right) \tag{3.9}$$

We can calculate its closed-form solution[2]:

$$t_k = \frac{h}{R}\left(\frac{1 - \alpha - \beta^{sg} + \alpha\beta^{sg} + \alpha(1-\alpha)^k\beta^{sgk}}{(1-\alpha)(1-\beta^{sg}+\alpha\beta^{sg})}\right) \tag{3.10}$$

Now we can finally construct an updated condition for beating this wave. By definition of t_k, we know that each attacker will have Rt_k HP when all attackers before it die. So, the player's defenses needs to deal in total $\sum_{k=1}^{n} Rt_k$ damage to all the attackers while they are the first attacker alive, instead of hs. Otherwise, the derivation is the same as for model 3, so the player will beat a wave if the following inequality holds:

$$C/s + (n-1)Rg \geq \sum_{k=1}^{n} Rt_k \tag{3.11}$$

If we put all wave-dependent terms on the right side of the equation and rearrange, the following inequality:

$$C \geq s\left(Rt_1 + \sum_{k=2}^{n}(Rt_k - Rg)\right) \tag{3.12}$$

However, there is still one problem we overlooked. We are using Rg as the extra damage each attacker beyond the first will take, because it is g seconds behind the attacker before it. However, this damage can never be greater than the attacker's health, Rt_k in this case. This wasn't a problem in model 3, because

[2]To calculate this, we first transformed the recurrence into one that depends only on the previous term, and then we used *Wolfram Mathematica* [28] to calculate the closed-form solution.

this would only happen in waves the player would beat anyway. So, we get the following updated condition:

$$C \geq s\left(Rt_1 + \sum_{k=2}^{n} \max(Rt_k - Rg, 0)\right) \tag{3.13}$$

If we substitute the definition of t_k, we get the following inequality:

$$C \geq s\left(h + \sum_{k=2}^{n} \max\left(h\frac{1-\alpha-\beta^{sg}+\alpha\beta^{sg}+\alpha(1-\alpha)^k\beta^{sgk}}{(1-\alpha)(1-\beta^{sg}+\alpha\beta^{sg})} - Rg, 0\right)\right) \tag{3.14}$$

We would like to get rid of the summation, but this is tricky because of the maximum function. However, for the parameters we allow, t_k is always decreasing with k. This means that for some n^*, the first n^* terms of the summation will be greater than zero and the rest will be zero. This lets us discard the maximum function and simplify the condition further:

$$C \geq s\left(h - (n^*-1)Rg + h\sum_{k=2}^{n^*} \frac{1-\alpha-\beta^{sg}+\alpha\beta^{sg}+\alpha(1-\alpha)^k\beta^{sgk}}{(1-\alpha)(1-\beta^{sg}+\alpha\beta^{sg})}\right) \tag{3.15}$$

Finally, we can calculate the closed-form solution[3] for the summation to obtain that the player's defenses will beat a wave if and only if the following condition is satisfied:

$$C \geq s\left(\frac{n^*(\beta^{sg}-1) - \alpha\beta^{sg}((1-\alpha)^{n^*}\beta^{sgn^*} + n^*(\beta^{sg}-1) - 1)}{(1+(\alpha-1)\beta^{sg})^2}h - (n^*-1)Rg\right) \tag{3.16}$$

Where n^* is the smallest k between 1 and n, such that the following inequality holds:

$$\frac{1-\alpha-\beta^{sg}+\alpha\beta^{sg}+\alpha(1-\alpha)^k\beta^{sgk}}{(1-\alpha)(1-\beta^{sg}+\alpha\beta^{sg})} \leq \frac{Rg}{h} \tag{3.17}$$

In case no such k exists, we set n^* to n.

3.6.5 Model 5: Multiple Batches

So far we have considered only waves with one attacker type. In section 2.3.1 we mentioned that waves can consist of up to three batches. Each batch contains some number of attackers of the same attacker type, spaced apart by some spacing. We can specify a set of wave parameters for each batch b_k instead of having one for the wave as a whole. These parameters have a subscript which denotes which batch they correspond to, for example the first batch (b_1) contains n_1 attackers with h_1 HP and speed s_1, with a gap of g_1 seconds between them.

To determine which defenses will beat a wave with N batches, we can extend the condition for model 4. We will refer to the right-hand side of inequality 3.16 as the total value of the attackers. For simplicity, we can then denote the total attacker value with the parameters for batch b_k as the batch value V_k. The defenses now have to beat all the batches one by one, but they still have the same damage

[3]Again, made easier by *Wolfram Mathematica*.

capacity C to work with. So, we can say the defenses will beat the wave when the following inequality holds:

$$C \geq \sum_{k=1}^{N} V_k \tag{3.18}$$

We also specified that the spacing between batches will be 1 second, but the previous condition acts like the next batch begins right as the previous one ends. So, we need to add 1 additional second of firing to each wave beyond the first. An updated condition could look like this:

$$C \geq V_1 + \sum_{k=2}^{N} \max(V_k - s_k R, 0) \tag{3.19}$$

Again, we limit the batches to have a non-negative value, because the towers cannot "restore the damage capacity" by dealing damage to dead attackers.

Even though they are not visible now, the collateral damage calculations are all wrong. We never investigated what happens with an uneven attacker spacing, however we don't even know what's the spacing going to be like. We know how far each attacker spawns behind the attacker before it, but attackers from different batches can have different speeds, so the spacing between them changes as they move, and they can even overtake each other. There really isn't a way to capture this behavior in our model without making it much more complicated. So, we will ignore it, and hope that the waves we generate don't deviate in difficulty too much from our estimate. If this ends up being a problem, we can adjust the wave generation algorithm to only produce waves that aren't problematic.

3.6.6 Model 6: Multiple Paths

The levels in our game can have multiple paths, and we can even spawn different attackers on each path. Since all paths in a single level will have similar lengths, we can treat them all the same. To start, we'll assume that the paths don't interact in any way and that each tower can only shoot at attackers on one path. This means that each path p_k will have its own damage capacity C_k and rate R_k. The player's defenses then beat a wave if they beat it on each path.

However, when generating a wave, we generate it based on the difficulty determined by the map generator. We want the difficulty of each path to vary between waves, but the overall difficulty to smoothly increase. It would be weird for the map generator to dictate the difficulty of each path separately, the wave generator should take care of that. So, it only specifies the total difficulty using one damage capacity C and rate R and for each wave, the wave generator will divide them between the paths.

3.6.7 Model 7: Abilities

So far, we've only considered attackers that have no abilities, towers with no special abilities, and we ignored the abilities the player can use (see section 2.3.7). In this section, we will incorporate these into our wave difficulty estimate.

Usable abilities usually provide one-time damage to the attackers, so they effectively contribute to the damage capacity of the player's defenses. Similarly,

special abilities of towers or other buildings which have some effect on the attackers, usually just make the defense more effective, thus increasing its damage capacity or rate. These should all be balanced based on playtesting, so they aren't too weak or too strong. Thus, they don't need any special treatment in our calculations.

On the other hand, attacker abilities is something the wave generator should take into account. We don't want it to spawn lots of attackers which have very strong abilities just because of their low HP and slow speed. These abilities can vary widely in their effect and can interact in unpredictable ways. However, we think it's enough to assign each attacker a value which will be used in all the calculations we did so far instead of their HP. For example, an attacker with no special abilities and 10 HP will still have a value of 10. If it had an ability that makes it take 50% less physical damage, its value might be about 13. Also, the damage rate of a defense (see equation 3.4) is influenced by the attacker speed, but we treated it as a constant the whole time. We can increase the value of very fast attackers to offset this, treating super-speed as a kind of special ability. The important thing is that this value can be tweaked based on playtesting.

This way of quantifying the wave difficulty should be good enough to let us produce waves that vary widely in their feel, but are consistent in their difficulty, so they don't feel unfair to the player.

3.6.8 Wave Generation Overview

In the previous sections we decided that the map generator will specify the difficulty of each wave using the value rate and capacity. Rate basically represents how much damage we expect the player's defenses to deal to an attacker per second. Capacity basically represents how much damage we expect the defenses to deal to an attacker during its travel from the path start to the Hub. We derived some formulas which tell us the total attacker value, and we say that a defense can beat a wave if its value capacity is greater than the total attacker value. Now that we can quantify the difficulty of a wave, we can start generating them. Our goal for each wave is to generate a random wave, but it has to have a total value very close to the expected capacity without going over.

We also know how many paths are in the level and what attackers we are allowed to use. In addition to the parameters prescribed by the map generator, the wave generator itself is going to have more parameters which we can tweak based on playtesting, in order to produce the best results. The first parameters the wave generator will have are limits on the maximum wave length, and the maximum number of attackers, because we don't want waves that are way too long or have way too many attackers. In section 3.6.4 we also mentioned that the wave generator will have parameters α and β that dictate the expected distribution of collateral damage.

In section 2.3.1 we specified that each wave doesn't have to spawn attackers on every path. Of course, each wave should always spawn attackers on at least one path. So, for each wave we will select one path as forced, and each path from the rest will be also selected for this wave with some probability which will be specified as another parameter.

We also decided that there will be two types of waves. We will call these types *sequential* and *parallel*. Sequential waves consist of up to three batches of

attackers, each with a different attacker type. Parallel waves consist of only one batch, but they can spawn a different attacker type on every path. There is no reason to make the first few waves complex, so we limit waves 1 and 2 to just one batch, and waves 3 and 4 to two batches. Additionally, we will allow each wave to only use the attacker types the waves before it used plus at most one new attacker type.

In section 3.6.6 we decided that we will evaluate the value rate, capacity and attacker value separately for each path. And that it is up to the generator how it will split up the value rate and capacity between the individual paths. However, that doesn't mean it should be just random. The towers the player builds are permanent, so each tower contributes its firepower to some paths from the moment its built until the battle ends. It would be really difficult for the player, if one wave spawned all its attackers on one path, and then another wave expected the player to have as many defenses on another path. So, we keep track of the expected rate and capacity per path. For every wave, only the extra rate and capacity it has over the previous wave is to be distributed between the paths freely.

However, the towers can cover multiple paths, and abilities can also be used on any path the player wants. So, we will reserve a fraction of the capacity as *global*, which can be used by any path and is not locked in afterwards. How big of a fraction will this be is another parameter to be determined by playtesting.

The generator starts each wave by selecting which paths it will use. Then the generator randomly picks which wave type this wave will be, with the distribution also specified as a parameter. Finally, it will generate the random wave, such that a defense with the expected value rate and capacity can beat it. Any capacity that's left over then gets added to the next wave to compensate. Since sequential and parallel waves will each be generated with a different algorithm, we will describe them separately in the following subsections.

3.6.9 Generating Sequential Waves

A sequential wave consists of up to three batches of attackers, each with a different attacker type. We want to select a random number of a random attacker type with a random spacing for each batch, and we are trying to use up as much of the capacity without going over. Notably, the same attackers will spawn on each of the selected paths. This means that we also expect the selected paths to have the same rate and capacity.

So, the first thing the algorithm does is distribute the new rate and capacity. We would like all paths to be equal, but we have to respect the rate and capacity values they have now, and we can distribute only the rate and capacity that are new for this wave. So, we at least make them as even as possible. This can be done by always giving to all paths with the lowest amount the amount required to reach the next lowest amount. Once they are all even, we split evenly the rest. However, there is no guarantee that the rates and capacities of all paths will be equal.

Now we can actually generate the wave. The most straightforward approach would probably use rejection sampling. Generate the specified number of completely random batches and check if whether the total value doesn't overshoot the capacity. If it does, we try again. To use up the capacity as much as possible, we

can generate many waves and select the one which has the greatest total attacker value. This approach could work, but it has obvious flaws that cause it to produce invalid results very often. We can improve this by removing them from the options for random selection.

For each batch, we first find all the valid attacker types this batch could consist of. We cannot use an attacker we have already used in a previous batch of this wave, so we filter out those. Also, if an attacker's value is so great, that spawning only one on each path overshoots the capacity, it is invalid. When we select a random attacker now, it is guaranteed to be valid.

Another problem a wave might have is that it has simply more than the maximum allowed attacker count, or it might be too long. It is also possible the wave overshoots the maximum capacity. So, after picking a random attacker type and a random spacing for a batch, we calculate the maximum number of these attackers the batch can have while staying within these limits. We then select a random attacker count between 1 and this maximum count.

This guarantees that each wave we generate is valid. However, we still need to generate many waves to find one that uses up the capacity somewhat well. To fix this, for the last batch of the wave, we select the spacing and attacker count that minimize the remaining capacity. This, however, isn't enough to guarantee that every wave we generate is good. It is possible for the last batch to select an attacker and spacing, such that it's impossible to use up the remaining capacity. This can happen because with collateral damage, some attackers don't contribute to the total attacker value anymore after some count, as described in section 3.6.4. Or simply because the previous batches used up too much of the wave duration or attacker count.

To remedy this, we add another condition when determining the valid attackers for the last batch of the wave. We remove the attacker types with which we cannot use up the remaining capacity. However, both the maximum amount of attackers we can use, and their value depends on the spacing we choose. So we have to check each spacing, and reject the attacker types for which none of the spacings work. Thus, for the last batch, we don't enumerate only the valid attacker types, but for each we also enumerate all the valid spacings.

This still doesn't ensure the previous waves didn't use up too much of the attacker count or wave duration. So, if a batch is unable to use up the remaining capacity completely, given the selected attacker type and spacing, we reduce its maximum count by one third, to leave some space for the next batches. Assuming there is enough different attacker types available, so that we never encounter a situation where there are no valid attackers available, this approach will always produce a valid wave that uses up most of the value capacity.

The whole process of generating a sequential wave is summarized as algorithm 5.

Algorithm 5 Generating a sequential wave

distribute new rate and capacity evenly

for i from 1 to batch count N **do**
    find all valid attacker types and spacings
    $a \leftarrow$ random valid attacker type

    if $i \neq N$ **then**
        $s \leftarrow$ random valid spacing for a
        $max \leftarrow$ calculate maximum attacker count
        $c \leftarrow$ random attacker count between 1 and max
        $b_i \leftarrow (a, s, c)$

    else
        for each valid spacing s for a **do**
            $c_s \leftarrow$ calculate maximum attacker count
        end for
        $b_i \leftarrow (a; s$ and c_s that maximize total value)
    end if
end for
return $b_1 \dots b_N$

3.6.10 Generating Parallel Waves

For generating a parallel wave, we will use a different approach. A parallel wave consists of only one batch that can spawn different attacker types on every path. But because it is one batch, all paths have the same number of attackers and the same spacing. Since we don't yet know the relative difficulty of the attackers we'll spawn on each path, we will keep the new value rate and capacity for this wave unassigned for now, and we will distribute them to the paths only after we have generated the wave.

Some requirements are the same as for a sequential wave: the wave cannot be longer than the maximum duration, it cannot have more than the maximum attacker count, and we want to use up as much of the capacity without going over. We will add some more that naturally extend these: Since we want to use up most of the capacity, it would make sense to use up all individual path capacities. We will again use up the global capacity by increasing the number of attackers as much as we can. To do this effectively, we will want from each path to have enough capacity for at least few attackers on its own. This allows for more granularity, for example the difference in value between 3 and 4 attackers is smaller than between 1 and 2 attackers, so we can get closer to the capacity limit.

To generate the wave, we first select a random spacing. Then we want to find a random set of attacker types, one attacker type for each path, such that it produces a valid wave, if we select the maximum amount that doesn't go over the maximum capacity. Similarly to a sequential wave, we can filter out all the attacker types that are invalid given the selected spacing. However, this time they can be different for every path. Based on our requirements, at least some

attackers of the type must fit within the current path capacity plus the current global capacity. This amount will be specified as a parameter based on playtesting. We also check that it is possible to use up the current path capacity using this attacker type and spacing.

Again, we could select random sets of attackers from these options until we find one that is valid. However, these validity tests can help us determine which attacker to change in case the set is not valid. We start by selecting a random attacker for each path from the set of valid attackers for that path. Then we enter a loop, where we do some tests, and if the set fails one of the tests, we change one attacker type and try again. If all tests succeed, we then produce the finished wave and distribute the new rate and capacity accordingly.

First, we test that it's possible to use up every path capacity: We calculate the minimum number of attackers the wave must have in order to use up every path capacity. We then check if this count is even a viable option for this wave. Specifically, we check whether this number of attackers would fit within the total capacity, including the new rate and new capacity we still haven't assigned. To do this, we first calculate the value of the attackers for each path and sum these values together. We now compare this value to the total combined value of all the capacities, including the new capacity and new rate. Since we know how many attackers are in the wave, we can convert the new rate to the maximum amount of extra capacity it will provide by multiplying it by the number of attackers after the first. If the total value of the attackers is greater than the final capacity, it means that we need at least some amount of attackers to fill each path capacity, but this amount would not fit within the final capacity, so this set is not valid.

This usually happens because some attacker had too much value relative to its path's rate and capacity, compared to the other attackers. So, this is the one attacker will change for another random attacker. Specifically, we change the attacker type of the path whose value in the previous test overshot its path capacity by the most. Then we try the test again with this new set.

If the test succeeds, we know the minimum number of attackers the wave can have in order to use up every path capacity. We also want to ensure this set of attackers can use up most of the capacity, given the constraints on the maximum wave duration and attacker count. So, as a second test, we calculate the maximum number of attackers these restrictions let us use, and then we check whether the value of this many attackers plus one is more than the total capacity in the same way as in the previous test. If the value is less than the capacity, it means that the maximum allowed attacker count still doesn't let us use up the capacity as much as we would want.

This time, the issue is that the attackers we want to use don't have enough value to use up the capacity. So, we change the attacker type of the path whose value in the previous test overshot its path capacity by the least.

If both tests succeed, we have found a good enough set of attackers for this wave. Now we just find the exact maximum number of attackers that will fit within the total capacity, and the wave is finished. We don't have a formula for this, but we can determine if a specific count fits within the total capacity or not, as we did in the tests. So, we can find the exact maximum count using binary search.

We have now generated a valid parallel wave, but we still have to somehow

distribute the new rate and capacity to the individual paths. To mimic what the player might do to handle this wave, we calculate by how much the attacker value overshoots the capacity of each path, and we distribute the new rate and capacity in the same proportion.

Again, assuming there is enough different attacker types, this process always produces a valid wave that uses up most of the capacity. The whole process of generating a parallel wave is summarized as algorithm 6.

Algorithm 6 Generating a parallel wave

1: select a random spacing s
2: for each path find all valid attacker types
3: $S \leftarrow$ random valid attacker for each path

4: $min \leftarrow$ calculate minimum attacker count to use up the all path capacities
5: **if** value of min attackers $>$ total capacity **then**
6: $\quad p \leftarrow$ path with the greatest value over the path capacity
7: $\quad S_p \leftarrow$ random valid attacker
8: $\quad$ **go to** 4 ▷ Try again.
9: **end if**

10: $max \leftarrow$ calculate maximum attacker count
11: **if** value of $max + 1$ attackers $<$ total capacity **then**
12: $\quad p \leftarrow$ path with the lowest value over the path capacity
13: $\quad S_p \leftarrow$ random valid attacker
14: $\quad$ **go to** 4 ▷ Try again.
15: **end if**

16: $c \leftarrow$ calculate maximum count that doesn't overshoot total capacity
17: distribute new rate and capacity proportionally to the attacker values
18: **return** (S, s, c)

3.7 Random Number Generators

The algorithms we use for procedural generation depend on a *random number generator*, or *RNG*, as their source of randomness. Their uses and how they work is explained well in *Random Number Generators–Principles and Practices: A Guide for Engineers and Programmers* [29] by Johnston. Some RNGs use specialized hardware to generate truly random data using an external source of entropy, these are called *true random number generators*. However, we want a *deterministic RNG*, also known as a *pseudorandom number generator* (*PRNG*). These produce the random data using a completely deterministic algorithm, but unless we know the current internal state of the generator, the outputs still can't be predicted. The initial state of a PRNG is called the *seed*, and a generator will always generate the same sequence of outputs when *seeded* with the same value.

Each query advances the generator's state, so the value a deterministic random number generator returns depends on the number of previous requests. If we used

one generator for generating everything, the outcomes of different systems would depend on the order they were generated in. For example, when a player triggers some effect that uses randomness *before* generating a level, the level would be different than if the player triggered the randomized effect *after* the level was generated. To remedy this, we will utilize a simple trick we call *seed branching* all throughout the procedural generation. Whenever we want more systems to be independent of each other, we create a new RNG instance for each system, and we seed them with each with a seed generated from the old RNG in advance. For example, we will have a master RNG seeded with the seed of the run, from which we will generate the seeds for the map generator, reward systems, etc. The map generator itself will generate the run map and then assign a new seed to each of the levels planned on the map, and so on.

We can determine what properties are required of the RNG we are going to use from our use-case. First, obviously, the numbers generated by the generator should be random enough. However, the RNG doesn't have to be cryptographically secure or pass strict statistical tests, since we aren't going to use them for cryptography or scientific simulations. Since we will create many instances of the RNG, it should be lightweight and fast to initialize. Some of them, for example the ones used by the reward system, will persist throughout the whole run, so we need an easy way to save the RNG's current state. So, what options do we have?

Since we are using Unity, the first RNG that comes to mind is Unity class `Random` [30]. It is designed to be easy to use, but it is very limited — for example, we have access to only one instance of the class and the same instance is used for other systems within the game engine. This is a dealbreaker for us, because we want to create more instances, and we want to have complete control over them to ensure determinism.

Another option that's on-hand is .NET `System.Random` [31]. According to the documentation, instantiating a random number generator is fairly expensive. Furthermore, there are no methods to read and set the internal state of the generator. This becomes a problem when we want to save the state of an instance to restore it later, for example when loading a save file. We would have to serialize and deserialize the instance, which isn't a big problem, but it feels inelegant and inefficient.

Instead, we chose to go with a more straight-forward option — making our own RNG. This way, we can make the generator have all the features we need. There are many algorithms a PRNG can use. Johnston describes in their book [29] some most commonly used non-cryptographic PRNGs, namely:

- Linear congruential generators (LCG),
- Multiply with carry (MWC),
- XORSHIFT,
- Permuted congruential generators (PCG).

All of these are random enough for our use-case, provided we use the right parameter values, so we chose an LCG, because it seemed the most simple to implement. In the article *Tables of Linear Congruential Generators of Different Sizes and Good Lattice Structure* [32], the author explains the statistical tests they used to

measure the randomness of the LCGs and tabulates the best-performing parameter combinations. From there we took the parameters for our LCG implementation.

3.8 Battle Simulation and Visuals

With procedural generation done, we will now focus on some interesting problems that we need to solve to implement the gameplay of the battles. First, we need to decide how will the battle play out, given the constraints specified mostly in section 2.3.12. We mention that the game will let the players pause or change the speed of the game. We want the game simulation to be deterministic, frame rate independent, and game speed independent. But, we want everything to look and feel smooth.

Games usually operate in a game loop, repeatedly taking inputs from the player, updating the game logic, and then outputting the new state to the player, like rendering a new frame. In Unity, the game logic has to be implemented in methods `Update` and `FixedUpdate` of our `MonoBehaviour` scripts, which Unity calls in the game loop. `Update` gets called before rendering every frame, whereas `FixedUpdate` gets called on each fixed update. Fixed updates are performed a set amount of times per second, and the Unity physics engine also uses them. This means that Unity is built with the separation of game logic and visuals in mind, and we don't have to do anything complicated to achieve this.

By default, `FixedUpdates` are performed 50 times per second. If we decrease the number of ticks per second, we decrease the performance cost of our game, which is especially useful if we want to speed the game up. We don't need any kinematics from the physics engine, only collision detection, so we don't need this much temporal resolution. For our game, 20 ticks per second should be enough. This means the shortest interval between any two events will be 0.05 seconds. This can be problematic, for example for towers with fast fire rate. For example, we cannot have a tower that shoots 8 projectiles per second, because it would need a 0.125 second interval between shots, which is not a multiple of 0.05. There are ways to circumvent this, but we think that this quantization isn't a problem for our game.

Everything in our game will happen in 0.05 second intervals. For example, a projectile will jump forward 20 times per second. We want everything to look as smooth as possible, so we will have to interpolate the positions of all moving objects and more. The game simulation doesn't need to know about any of this, so we will completely separate this logic from the game logic. The visualization logic will react to the simulation logic.

Unity offers many useful systems for visuals, for example particle systems or animations. Most of the visuals should change speed with the simulation speed. Luckily for us, Unity has a built-in global variable that lets us change the speed of the game, including both the simulation speed and the speed of visuals. Some visuals will be game speed agnostic, like instant animations or the game user interface animations. This also isn't a problem since Unity lets us specify to use the unscaled time for individual effects.

3.9 Targeting Attackers

An important question we should consider is during a battle, how will the towers determine which attackers are in range, and which attacker to shoot at? As described in section 2.3.6, most towers will have a range in the shape of a cylinder, but some will have a different shape. Most towers will select only one attacker as their target, but some don't need a specific target, for example those that damage all attacker in their range. The towers will have unique behaviors, but the vast majority will need to know which attacker is in range at all times, so it makes sense to separate targeting from the rest of the tower's logic. The tower can then ask its targeting system which attackers are in range, or which attacker to shoot at.

Technically, a tower needs to know which attackers are in range and which one of them to target just before it shoots. However, we usually want to make the tower visually turn towards the attacker's position before it shoots, so we want to know its preferred target at all times. Checking every attacker in the world on every tick would cause performance issues, since there can be a lot of towers and attackers at once. So, we need to come up with a more efficient solution. Physics engines need to perform similar checks many times per second, so they are optimized to do them as efficiently as possible. Specifically, we can think of our problem as a collision detection problem: we want to know which attacker colliders collide with the tower's range collider. So, we can use the Unity physics engine to detect which attackers are in range by representing the range with a collider.

Unity provides callbacks for us whenever another object enters the collider of our range and whenever an object leaves the collider. Thus, we don't have to ask which attackers are within range on every tick. Instead, we keep a list of the attackers in range and add or remove attackers when they enter or leave.

We can also use the physics engine whenever we need to enumerate all attackers in some area, for example when using an ability that affects all attackers in some radius. Unity physics lets us query which attackers are in some region using the built-in functions, which should be faster than checking each attacker one by one. For example `Physics.SphereCast` returns all colliders that intersect with a given sphere.

In section 2.3.6 we also mentioned that some towers will require a line of sight to the attacker they want to shoot at. In other words, if we draw a line segment from the tower to the attacker, it can't collide with anything. We can again use Unity physics, specifically `Physics.Linecast` which does exactly this.

However, linecasting is still somewhat expensive, so we don't want to test each attacker in range on every tick. Most towers will have a configurable targeting priority that determines which attacker to target if multiple are in range, especially those that require line of sight. So, we can save on many linecasts by testing the attackers in order from the highest priority until we find one for which the test succeeds. This means that we need to sort the attackers by priority. Their order usually won't change between subsequent ticks, so we keep them in a sorted order in the list of all attackers in range.

3.10 Range Visualization

In section 2.7 we described that we want to draw the range of the tower that's currently selected, or the tower or ability that is being placed, directly on the terrain. We want to show if the tower could shoot at a small or large attacker, depending on their position in the world. These regions will then be differentiated using different colors. The tower ranges can have complicated shapes, and some towers need a line of sight to the attacker to be able to shoot them. To produce the visualization, we need to solve two problems: how to determine this range shape, and how to communicate it to the terrain shader.

3.10.1 Determining The Range Shape

The shape of the range of a tower usually isn't so complex. This is because we are building them from primitive collider shapes, but it is also intentional, because complex shapes would be unnecessarily confusing to the player. However, we don't show only the bounds of the range, we show explicitly where an attacker can be for the tower to target it. This is complicated by line of sight checks, some tower being unable to shoot upwards or downwards, etc.

The simplest approach is to construct an approximation by testing points on the world in a square grid. The result of the point test will then determine the color of a small square centered on the point. Theoretically, if we sample enough points, we can create visualization that has greater resolution that the player's screen.

So, for each special range type, we need an implementation of some sort of oracle that tells us for each point whether an attacker at that point would be visible or not. For some towers, this oracle will perform a simple check whether the point lies within the range, but for others, more checks will be performed, notably the line of sight checks which are expensive. Doing thousands of these point tests whenever the player selects a tower which performs line of sight checks would cause a noticeable stutter. So, we will create a low-resolution visualization first, and slowly refine it by performing more tests over the next frames.

We will probably perform only few hundred line of sight checks per frame, and we don't want the visualization to keep changing for several seconds, so we will go with a rather small final resolution. A grid of 256×256 squares, which is 65536 in total, should be enough, giving us 16×16 samples per tile. The visualization will be obviously pixelated, but it should communicate the towers' range well.

We can build this visualization as a *quadtree*. A quadtree is a data structure in the form of a rooted tree, where the root node represents the whole world as one large square, and each internal node has four children which split the parent node into four smaller squares. We can start our visualization with the root node, and over time refine each leaf node by adding its four children. For each child we perform a point test in its center and assign to it the obtained value. This is repeated until all leaf nodes are of the required depth, which would be 8 for a 256×256 square grid. This means we perform one third more point tests, but we can have a valid representation of the range after every frame that gets better over time.

We can choose in which order to expand the leaf nodes by calculating a priority

for each and storing unrefined nodes in a priority queue. We should prioritize nodes which are larger, but also nodes whose siblings have different results than them, because this means they are near a boundary where the results change from one state to another.

This representation also lets us easily perform *quadtree compression* to save memory: Whenever all four children of an internal node match, and they are leaves that won't be refined further, we can delete them and mark their parent as also finished. This can happen recursively, so that large squares with the same color can be represented by just one node.

Furthermore, we don't have to refine some nodes at all, when we know for sure that all points tests within their square will yield the same result. To help with this, the point test oracles won't return only the result at that point, but also the guaranteed minimum distance to a point which gives a different result. When the distance is greater than half of the current square's diagonal, we don't have to refine it further. This distance is easy to determine for simple range shapes, but for towers with line of sight checks, it will always be reported as 0 for points inside the range.

3.10.2 Representation for The GPU

Now that we can compute the visualization, we also need to render it. To do that, we need to send it to the GPU and display it using a shader. But how do we represent it in the GPU?

The first option that comes to mind is to just take the quadtree we constructed and send it to the GPU. Of course, we would have to put the data into some shader buffer, probably by creating an array of nodes. We want to reduce the bandwidth as much as possible, so all children of a single node could be consecutive in the array, so each node would only need to store one index to the array to reference all its children. Theoretically, we would need at most $4/3 \times 65535$ nodes in order to represent the 256×256 resolution we selected, but thanks to quadrant compression, we usually won't as many. So, each node could be represented by 2 bytes, since we only need to save one index to the children or the result of the point test for leaf nodes.

However, this approach is expensive on the CPU, because we need to prepare this data structure. And on the GPU too, because it needs to traverse the tree to its leaf to determine the color for each pixel with the terrain, but that isn't as big of a problem, since overall, the GPU is much more powerful.

A better option might be to store the data directly in a texture. In the GPU, we only have to access the texture once to determine the color of a given position on the terrain, making it much simpler. On the CPU side, it will be simpler too. For each node, we fill a square that is represented by it the texture with the value of the node, all the way to single pixels for the deepest nodes.

This is much better, but we can save some more CPU computation with a trick: *Mipmapping*, first described in the paper *Pyramidal parametrics* [33], is a widely used technique where we generate several lower-resolution variants of a texture, which can then be used to reduce aliasing and performance cost when rendering surfaces from a great distance or when viewed at a shallow angle. The width and height of each additional image is a half of the previous level. These are

usually automatically generated from the main texture, but we can make them look how we want, and we can use them how we want.

If we create a texture with mip levels all the way to a 1×1 image, and we imagine all the layers stacked on top of each other, stretched to be the same size as the terrain, we essentially get a quadtree. The root node of our quadtree is the only pixel in the highest layer, and each other pixel represents another node, each in the layer corresponding to its depth, directly below its parent. Now, the CPU only has to change one pixel for each node. However, the GPU has to access the texture multiple times per each pixel again, but that shouldn't be a problem. Including the mipmaps also makes the texture one third larger than without them. Whenever we change a part of a texture, it has to be copied to the GPU again. Since we do this every frame, bandwidth will probably be the limiting factor, but for our small 87 KiB texture this shouldn't be an issue.

3.10.3 Computing Everything on the GPU

We would like to mention that only after implementing the demo version of our game, we thought of a solution that can be used if higher resolution of the visualization is needed, or just to improve performance. This is the exact kind of task that GPUs are great at: computing the same thing for many different points at once. So, it would be the best to only send the world geometry to the GPU, and compute these range visualizations there. Testing which points are in the line of sight of some tower is the same, as if the tower was a point light, and we wanted to draw the shadows. This is a problem that has been heavily studied and there are many techniques we could use for it.

3.11 Blueprints and Info Panel Text

In section 2.4 we described that in our game, the player will be able to build buildings or use abilities, only if they have their blueprint. Each blueprint will include a description that explains its function, including the exact values of important statistics.

3.11.1 Blueprint Representation

We also mentioned that the blueprints will be modifiable, be it permanently or temporarily. For example, we could have a building that increases the range of all buildings adjacent to it, or an augment that increases the damage of the blueprint it augments. To do this, we would like to know from each blueprint whether it has a given statistic and its value. It is also important to note that we want to access these statistics even without creating an instance of the thing the blueprint represents. So, all the statistics we could ever want to know or modify will not be stored in the class that represents blueprints, separately from the implementation of their behavior.

In Unity, we will implement blueprints as *scriptable objects*. Unity allows us to edit the values of these in the editor, and to save them in the project as separate files. We can also include references to other resources in our project, for example the *prefab* of the object that get instantiated when the blueprint is used in a

battle. These prefabs are then where we put the scripts that implement the actual behavior of the building or ability. The blueprint itself will only hold the stats and the description. When the prefab is created, it will be assigned a copy of the blueprint, in order to have access to the blueprint statistics, which can be modified separately from the source blueprint.

Similarly, we will use scriptable objects to represent attacker types. They will hold the stats and the description of the attacker type, and a reference to the prefab of the attacker itself.

3.11.2 Info Panel Text

In section 2.5.7, we describe that the information about anything the player has selected will be displayed on an info panel on the right side of the screen. For blueprints, this will be mainly their statistics and the description. For buildings, the info panel will display the up-to-date stats and description of their blueprint, but the buildings can also include some information of their own, for example, each tower will show the damage it has dealt throughout the battle. We also specified that the statistics that have changed from the original blueprint will be highlighted using colors, indicating whether they changed for better or for the worse. Additionally, any new or removed abilities will also be highlighted.

To dynamically update the statistics, we will use placeholder tags in the descriptions, and then a preprocessor, which will find these tags and replace them with the up-to-date values, including the correct formatting. For example, the tag `[DMG]` will be replaced by the damage statistic, correctly formatted, and prefixed with an icon that symbolizes the damage stat, to show that if something changes damage, this value will change. Of course, to compare the new value, we will also need to store the original value.

To display this formatting and icons within text, we will use the features of the Unity package *Text Mesh Pro*. It lets us format the text with tags similar to HTML. For the icons, we can create a *Text Mesh Pro sprite asset*, and then insert the icons into the text using the tag `<sprite=ID>`.

3.12 Modifiable Commands and Queries

In our game, we want various objects to modify how other objects function. However, this can be very problematic if we don't select the right approach.

For example, we could have a building that slows down attackers in its range by 25%, or an attacker that speeds up by 1 tile per second once its HP drops below half. However, this can cause issues when implemented incorrectly: Imagine the attacker that speeds up has a base speed of 1 tile per second. Then, it enters the range of the building that slows it down to three quarters, dropping its speed to 0.75 t/s. Then its speed increases by 1 t/s, because its HP drops, so the new speed is 1.75 t/s, but we would expect it to be 1.5 t/s. Finally, when it leaves the range of the building, its speed increases by third of the current speed to revert the slowdown, ending up at 2.333 tiles per second. This is obviously not equal to the expected speed of 2 tiles per second.

To illustrate another problem we might run into, we imagine a building that makes it so whenever you would get energy, you get that many materials instead.

How would we implement it? We obviously don't want to modify the blueprints of other buildings, and there might be other sources of energy too. We could add a special case to the code that handles the player's current material and energy balance to check specifically for this building, and handle the conversion if needed. That cannot be a good solution. Furthermore, the sources of the energy would still show pop-ups that they produced some energy, even though they produced materials instead.

3.12.1 Modifiable Commands

Our solution for these issues is what we call *modifiable commands and queries*. This system is an extension of the observer pattern. In this pattern, many *subscribers* can subscribe to an *event*, to be notified when the event occurs. This event can be triggered by any other object that has a reference to it, also known as *publisher*. The advantage of this pattern is its weak coupling. The subscribers don't have to know anything about other subscribers or about the publishers and vice versa.

To create modifiable commands, we add the option for subscribers to modify the data of the event, and we invoke the subscribers' callbacks in an order specified by the priority that was specified when they subscribed to the command. For example, when a building wants to produce energy, it invokes the modifiable command to add the given amount of energy. The command then notifies all the registered modifiers' callbacks, one by one. Each subscriber alters the amount of energy how it wants, and finally, the last subscriber, which we'll call the *handler*, reads the energy amount and adds it to the total. The handler also doesn't have to be the last subscriber, there can be more subscribers after it which react to the event, for example by displaying the amount produced. We will call these *reactions*, and we will refer to the subscribers that get notified before the handler as *modifiers*. We will also let modifiers cancel the command completely.

It doesn't make sense to have more than one handler for the command, so we will say that the handles always has priority 0, and there can only ever be one. The reactions shouldn't need to modify the command anymore, since it has already been handled by the handler, so we don't let them modify the command, only react to it. This means the signature of their callback will be different, and they will be registered separately from the modifiers. We will also force callbacks registered as a reaction to have a positive priority, so they happen after the handler. And that leaves negative priority values for modifiers.

It is now easy to see how we would implement the example building that turns energy into materials. It would simply register a modifier to the command for producing energy, which would cancel the command and invoke a command to produce materials instead.

3.12.2 Modifiable Queries

We could solve the first example with the attacker speed similarly. Instead of just reading the speed value from the attacker stats, we will have a modifiable command for reading it. So, when anyone wants to know the attacker's speed, they invoke the command with the base speed 1 as the initial data, and after

series of modifications, they handle the updated value. This ensures that multiple simultaneous modifiers don't produce an invalid value, and the order of application no longer matters, because they are ordered by their priority.

However, we don't ever need to react to a query, furthermore it would make no sense for a modifier to cancel a query. Also, it makes no sense to handle the result using a callback that is then immediately unsubscribed. So, we remove these features and streamline the interface to get *modifiable queries*.

It would be very inefficient to go through this whole process a thousand times per second, when reading a value that is not currently changing, as can happen when creating the range visualization for a tower. So, we also cache the result of the modifiable query to save on computation. The cache gets invalidated whenever a modifier is subscribed, unsubscribed, or whenever someone calls the function to explicitly invalidate the cache.

3.12.3 Event Reaction Chain

For modifiable queries, we used only the first half of the modifiable command pipeline. If we take just the second half, we get just an event system, but with reactions ordered by priority. This is also useful for us, because the reactions could create race conditions otherwise. We call this the *event reaction chain* just to differentiate it from Unity events, Unity event system and .NET events.

4 Developer Documentation

empty

5 User Documentation — Designer

empty

6 User Documentation — Player

- overview
- installation
- hotkeys
- attacker encyclopedia
- blueprint encyclopedia

7 Playtesting

- why playtest
- how did it go
- we fixed some bugs and tweaked the difficulty
- common problems - info panel obstructs, hard to inspect attackers, more...
- maybe fuel is unnecessary (and energy limit)
- compare with design goals

Conclusion

Bibliography

1. Wikipedia contributors. *Tower defense — Wikipedia, The Free Encyclopedia* [online]. 2024. [visited on 2024-05-06]. Available from: https://en.wikipedia.org/w/index.php?title=Tower_defense&oldid=1212378185.
2. Avery, Phillipa; Togelius, Julian; Alistar, Elvis; Leeuwen, Robert. Computational Intelligence and Tower Defence Games. In: 2011, pp. 1084–1091. 2011 IEEE Congress of Evolutionary Computation. Available from doi: 10.1109/CEC.2011.5949738.
3. Electronic Arts Inc. *Buy plants vs. zombies - PC & mac - EA* [online]. [N.d.]. [visited on 2024-05-06]. Available from: https://www.ea.com/games/plants-vs-zombies/plants-vs-zombies.
4. Bycer, J. *Game Design Deep Dive: Role Playing Games*. CRC Press, 2023. isbn 9781000966718.
5. Wikipedia contributors. *Rogue (video game) — Wikipedia, The Free Encyclopedia* [online]. 2023. [visited on 2024-05-06]. Available from: https://en.wikipedia.org/w/index.php?title=Rogue_(video_game)&oldid=1191758861.
6. Bycer, J. *Game Design Deep Dive: Roguelikes*. CRC Press, 2021. isbn 9781000362046.
7. Mega Crit. *Mega Crit – Slay the Spire* [online]. [N.d.]. [visited on 2024-05-06]. Available from: https://www.megacrit.com/.
8. Dictionary.com. *Build* [online]. [N.d.]. [visited on 2024-04-13]. Available from: https://www.dictionary.com/browse/build.
9. Ironhide Game Studio. *Kingdom Rush* [online]. [N.d.]. [visited on 2024-04-16]. Available from: https://www.kingdomrush.com/kingdom-rush.
10. Ninja Kiwi. *Bloons TD 6* [online]. [N.d.]. [visited on 2024-04-16]. Available from: https://ninjakiwi.com/Games/Mobile/Bloons-TD-6.html.
11. Wikipedia contributors. *Desktop Tower Defense — Wikipedia, The Free Encyclopedia* [online]. 2024. [visited on 2024-05-06]. Available from: https://en.wikipedia.org/w/index.php?title=Desktop_Tower_Defense&oldid=1220758442.
12. Wikipedia contributors. *Isometric video game graphics — Wikipedia, The Free Encyclopedia* [online]. 2024. [visited on 2024-05-29]. Available from: https://en.wikipedia.org/w/index.php?title=Isometric_video_game_graphics&oldid=1224963052.
13. Shiny Shoe. *Monster Train* [online]. [N.d.]. [visited on 2024-05-29]. Available from: https://www.themonstertrain.com/.
14. Unity Technologies. *Unity Documentation* [online]. [N.d.]. [visited on 2024-04-07]. Available from: https://docs.unity.com.

15. Wikipedia contributors. *Maze generation algorithm — Wikipedia, The Free Encyclopedia* [online]. 2024. [visited on 2024-06-04]. Available from: https://en.wikipedia.org/w/index.php?title=Maze_generation_algorithm&oldid=1223006272.

16. Boris the Brave. *Random Paths via Chiseling* [online]. [N.d.]. [visited on 2024-06-04]. Available from: https://www.boristhebrave.com/2018/04/28/random-paths-via-chiseling/.

17. Boris the Brave. *Chiseled Paths Revisited* [online]. [N.d.]. [visited on 2024-06-04]. Available from: https://www.boristhebrave.com/2022/03/20/chiseled-paths-revisited/.

18. Kirkpatrick, S.; Gelatt, C. D.; Vecchi, M. P. Optimization by Simulated Annealing. *Science*. 1983, vol. 220, no. 4598, pp. 671–680. Available from doi: 10.1126/science.220.4598.671.

19. Merrell, Paul C. *Model synthesis*. 2009. PhD thesis. The University of North Carolina at Chapel Hill. Also available at https://paulmerrell.org/wp-content/uploads/2021/06/thesis.pdf [visited on 2024-06-01].

20. Gumin, Maxim. *WaveFunctionCollapse* [online]. [N.d.]. [visited on 2024-06-01]. Available from: https://github.com/mxgmn/WaveFunctionCollapse.

21. Marian. *Infinite procedurally generated city with the Wave Function Collapse algorithm* [online]. [N.d.]. [visited on 2024-06-01]. Available from: https://marian42.de/article/wfc/.

22. Wikipedia contributors. *Local consistency — Wikipedia, The Free Encyclopedia* [online]. 2023. [visited on 2024-06-05]. Available from: https://en.wikipedia.org/w/index.php?title=Local_consistency&oldid=1179702336.

23. Stålberg, Oskar. *Townscaper* [online]. [N.d.]. [visited on 2024-06-02]. Available from: https://rawfury.com/games/townscaper/?portfolioCats=30%2C23%2C20%2C25%2C18%2C26%2C28%2C16%2C17%2C11.

24. Wube Software. *Factorio* [online]. [N.d.]. [visited on 2024-06-08]. Available from: https://factorio.com/.

25. Earendel; Genhis. *Friday Facts #390 - Noise expressions 2.0* [online]. 2023. [visited on 2024-06-08]. Available from: https://factorio.com/blog/post/fff-390.

26. Perlin, Ken. An image synthesizer. In: *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*. Association for Computing Machinery, 1985, pp. 287–296. SIGGRAPH '85. isbn 0897911660. Available from doi: 10.1145/325334.325247.

27. Hind, Daniel; Harvey, Carlo. A neat approach to wave generation in tower defense games. In: *2022 International Conference on Interactive Media, Smart Systems and Emerging Technologies (IMET)*. IEEE, 2022, pp. 1–8.

28. Wolfram. *Wolfram Mathematica: Modern Technical Computing* [online]. [N.d.]. [visited on 2024-06-18]. Available from: https://www.wolfram.com/mathematica/.

29. Johnston, D. *Random Number Generators–Principles and Practices: A Guide for Engineers and Programmers*. De Gruyter, Incorporated, 2018. ISBN 9781501506260. Available also from: https://books.google.cz/books?id=yniVDwAAQBAJ.

30. Unity Technologies. *Unity - Scripting API: Random* [online]. [N.d.]. [visited on 2024-04-07]. Available from: https://docs.unity3d.com/ScriptReference/Random.html.

31. Microsoft. *Random Class (System) - Microsoft learn* [online]. [N.d.]. [visited on 2024-04-07]. Available from: https://learn.microsoft.com/en-us/dotnet/api/system.random.

32. l'Ecuyer, Pierre. Tables of Linear Congruential Generators of Different Sizes and Good Lattice Structure. *Mathematics of Computation*. 1999, vol. 68, no. 225.

33. Williams, Lance. Pyramidal parametrics. In: *Proceedings of the 10th Annual Conference on Computer Graphics and Interactive Techniques*. Association for Computing Machinery, 1983, pp. 1–11. ISBN 0897911091.

List of Figures

1.1 A level in *Plants vs. Zombies*. 8
1.2 The map screen in *Slay the Spire*. 10
1.3 A fight in *Slay the Spire*. 10

2.1 *Defend*, *Strike*, *Iron Wave* and *Double Tap* cards from *Slay the Spire*. 15
2.2 All the plants of *Plants vs. Zombies* in the in-game almanac. . . . 16
2.3 Card reward screen in *Slay the Spire*. 17
2.4 Seed select screen in a rooftop level in *Plants vs. Zombies*. 18
2.5 Next wave indicator from *Kingdom Rush*. 20
2.6 The levels *Park Path* and *Another Brick* from *Bloons TD 6* with the attacker paths highlighted. 21
2.7 Attackers being funneled between towers in *Desktop Tower Defense*. 22
2.8 Wave preview from *Desktop Tower Defense*. 22
2.9 An example of a valid path network in a 7×7 game world. 25
2.10 Attackers on a path that splits and joins. 25
2.11 A path network with undesirable properties and a path network with great properties. 26
2.12 A path with undesirable side branches. 27
2.13 A mockup of the GUI during a battle. Red numbers in circles marking the individual components. 35
2.14 Two attackers with HP indicators. The one on the left has 4/5 HP, the one on the right has 25/35 HP. 39
2.15 Example visualization of a tower's range. 41

3.1 A path network in a 7×7 tile world. 46
3.2 Real path starts compared to the ones we work with. 47
3.3 All the possible tile changes. 50
3.4 Untwisting a self-intersecting path. 52
3.5 Evolution of paths during simulated annealing. 53
3.6 Blocked edges and tiles after generating the terrain and obstacles. 55
3.7 Randomly generated paths which respect the original path distances. 56
3.8 Calculated minimum distances. 57
3.9 Final paths compared to the originally generated paths. 58
3.10 Example input and output of the wave function collapse algorithm. 59
3.11 Example output of WFC, using the modules on the left and only the constraint that their edges must match. 60
3.12 Two steps of wave function collapse. 60
3.13 Conflicts and backtracking in WFC. The red cross marks a slot with no valid modules. 62
3.14 The slots for generating a 3×3 tile world. 63
3.15 An example module and its constraints. 64
3.16 An example terrain generated using WFC. 66
3.17 Comparison of generated obstacle models. 69
3.18 The values of the height function and signed distance functions. . 70
3.19 Examples of Perlin noise. 71

3.20 Building up a noise expression step by step. 71
3.21 Which defenses can beat various attacker waves, according to model 3. 76

List of Tables

3.1 Success rate of terrain generation based on maximum backtracking depth. 65
3.2 Success rate of terrain generation based on how the algorithm decides which slot to collapse next. 66

List of Abbreviations

A Attachments

A.1 First Attachment