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

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 (might not happen).
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.

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. 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.

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.

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. Each terrain type will have its distinct look, keeping the levels from being all the same.

To summarize:

- The world each level takes place on will be a grid of 15×15 square tiles.
- There will be *small* or *large* obstacles on some tiles, large obstacles blocking certain towers' line of sight.
- The tiles will be at different heights in multiples of 0.5 units, and some tiles will be slanted, going between two heights.
- Attacker paths will consist of segments going from the center of one tile to the center of a neighboring tile.
- The player can build one building per tile, and only if the tile doesn't contain an obstacle or the attacker path.

- 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.

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*.

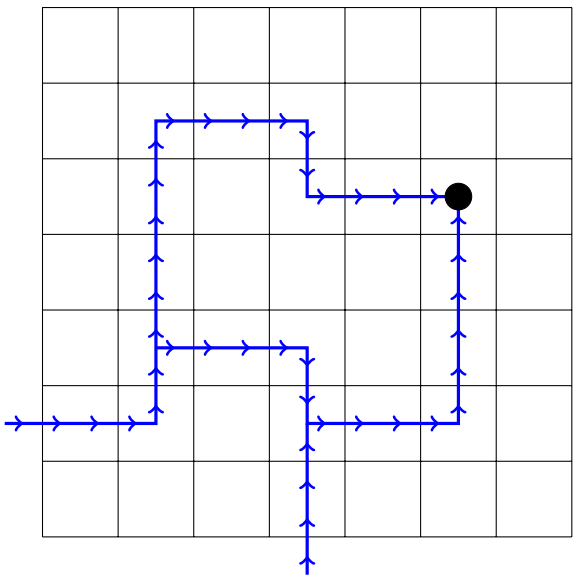


Figure 2.9 An example of a valid path network in a 7×7 game world.

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.

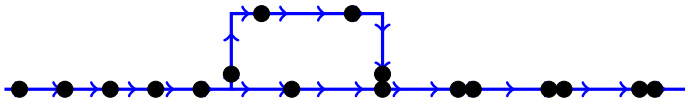


Figure 2.10 Attackers on a path that splits and joins.

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.

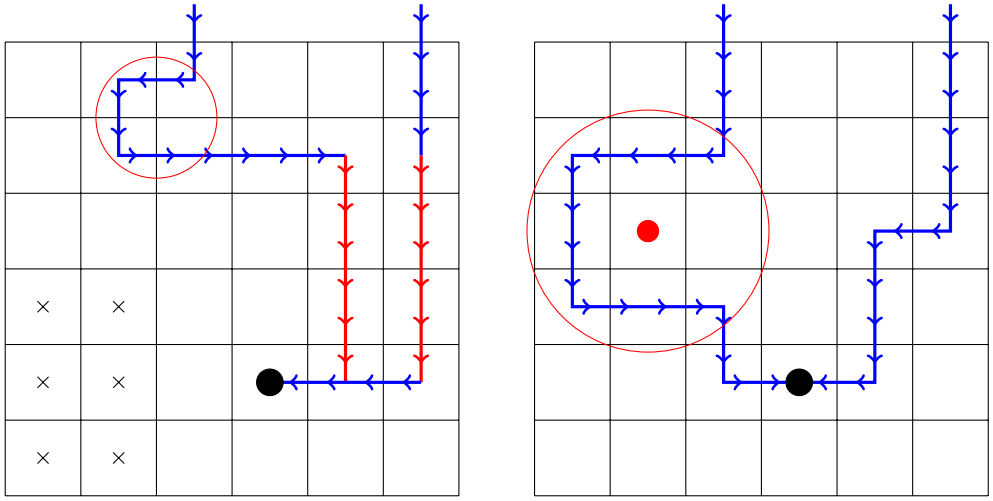


Figure 2.11 A path network with undesirable properties and a path network with great properties.

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 side branch must go through at least one tile that is not adjacent (by an edge or by a corner) to any already existing path.

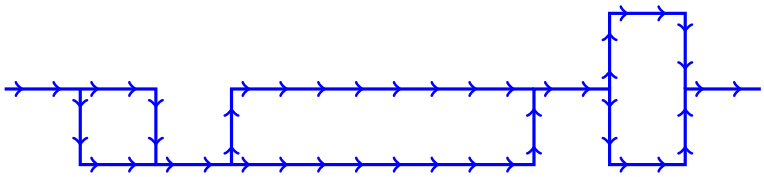


Figure 2.12 A path with undesirable side branches.

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

- Paths are formed by one-way segments, each from one tile to its neighbor.
- Paths start just outside the playable portion of the world, and there can be one or more path starts in each level.
- Paths can split or join.
- All paths must end on the tile with the *Hub*, no other dead ends can exist.
- 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.
- Paths should be spread throughout the playable world, not bunched up.
- Paths right next to each other or the edge of the world, and sharp U-turns (see figure 2.11), should be rare.
- Every side 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 your towers' range, 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 to the *Hub*.
- Last — the attacker that's farthest 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.

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 your 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.

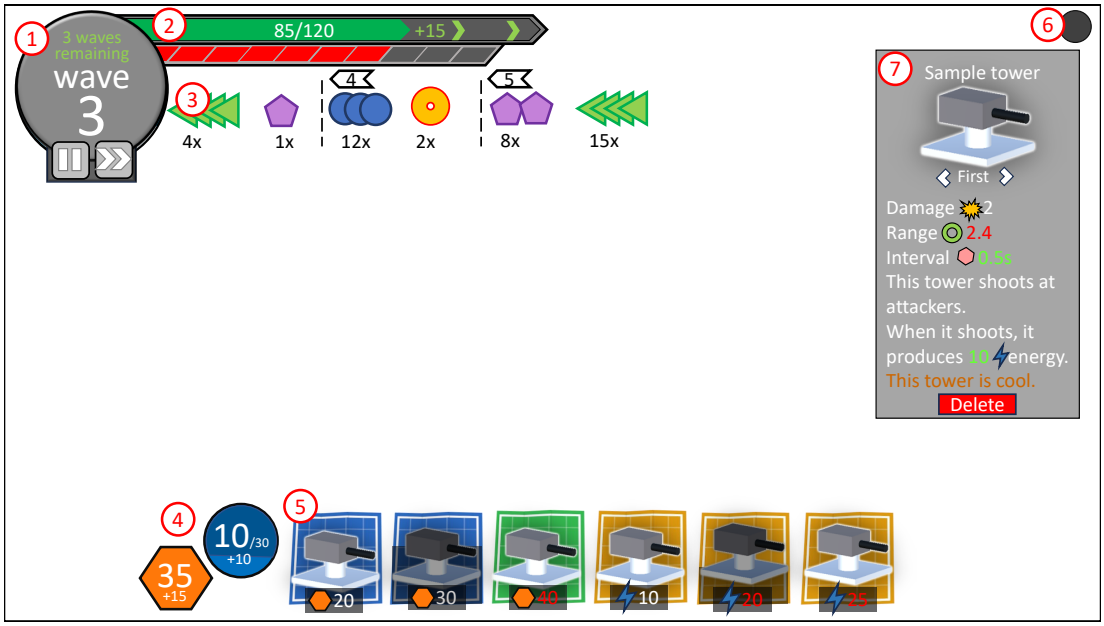


Figure 2.13 A mockup of the GUI during a battle. Red numbers in circles marking the individual components.

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 you encounter 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.

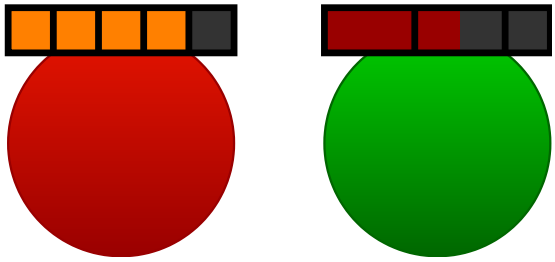


Figure 2.14 Two attackers with HP indicators. The one on the left has 4/5 HP, the one on the right has about 150/250 HP.

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 250 HP, their indicator will consist of two 100 HP sections and one 50 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. 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.

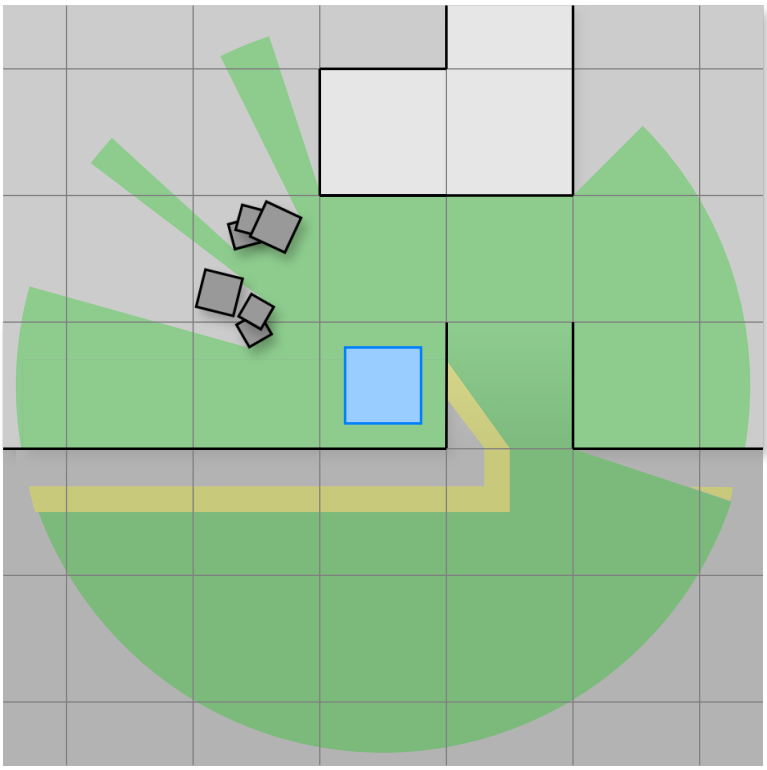


Figure 2.15 Example visualization of the range of a tower represented by the blue square.

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. From the player's perspective, all this and the rewards they receive will be randomized 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 want to allow the player to save the game between battles. Here, determinism is also useful, because it allows us to save just the seed of something that was generated, instead of saving it whole, which leads to simpler and smaller save files.

3.2.1 Random Number Generators

Randomized algorithms, like the ones we will use for procedural generation, depend on a *random number generator* as their source of randomness. A **random number generator** (or *RNG*) produces a sequence of numbers that looks random and is unpredictable. They are well explained in *Random Number Generators–Principles and Practices: A Guide for Engineers and Programmers* [15] by David 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` [16]. 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` [17]. 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 [15] 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* [18], 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.

3.3 Path Generation

The worlds we want to generate for the battles in our game 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 most restricted, because each part is additionally restricted by what was generated before it. Thus, the world generation algorithm will start by generating paths. There is a lot of rules the paths should follow, as described in section 2.3.3. Additionally, the number of paths and their lengths heavily influence the difficulty of the battle. A battle's difficulty should be decided by its position in the run, and it should be known to the player even before they select it, so they can decide which battle to fight. This means, that the number of paths and their lengths will be given to the world generation algorithm as an input, further restricting the paths that will generate.

One of the features the paths will have is that they can split into more and join together. When one path splits into two branches that go in separate directions, the battle would play out similarly to a battle with two distinct paths. However, for the player, branching paths are not as difficult to manage as distinct paths. They are usually closer together, because they have in common both the end point and the start point. Furthermore, there is a stretch of path before it split into two branches, and a few tiles after the split, the branches are still pretty close together. The player can concentrate their defenses around this portion of the level, partially mitigating the disadvantages of the split. Additionally, these splits in paths can vary widely in length.

This all means that it's hard to quantify the relative difficulty of various path networks. We believe that it's enough to specify the number of path starts, the lengths of each path, and the maximum number of extra branches to sufficiently define the difficulty range of the path network that will be generated.

We can split the path generation process into three stages:

1. Select sufficiently spaced out positions for the path starts, in order to spread the individual paths apart as much as possible.

2. Generate the main branch of each path.

3. Refine the paths in order to make them split into branches and join together.

We feel it's a good idea split stages 2 and 3 mainly because it is simpler to treat them as two separate problems. We believe this leads to simpler and faster algorithms without failing the requirements set in section 2.3.3. How to accomplish the goal of each of these stages will be described in the following subsections of this section.

3.3.1 Path Starts

In the first stage of path generation, we need to pick the path start positions.

One thing we should notice, is that starts of paths of even length have to have an even *Manhattan distance* from the *Hub*, and starts of paths of odd length have to be an odd distance from the *Hub*. This parity holds, because by going from one tile to another that shares an edge with it, the Manhattan distance always changes by 1. When a path goes from a tile to another tile l times, its Manhattan distance changes parity l times. The distance of the final tile must be 0, which is even, so the parity of the path distance and Manhattan distance must match on every tile, including the starting tile.

So, in the first step we will find all tiles along the edge of the world, and separate them into two sets — even and odd. Now, we could just pick a random tile from the corresponding set for each of the starts. However, there is one problem: a path of length l can start at most l tiles from the *Hub* in Manhattan distance, since the distance equals the length of the shortest path between the two tiles. We also want the path starts to be sufficiently spread out from each other, so that individual paths come from different directions. And we don't want the paths to start too close to the *Hub*, especially in levels with just one long path — it should ideally start as far from the *Hub* as possible, in order to fill the world more evenly.

We can simply use rejection sampling to pick starts that satisfy these conditions. When we select a candidate tile, we accept it only if it satisfies all of these conditions, otherwise we throw it away. One exception is when a candidate tile is too far from the *Hub*. Then we return it into the set after we're done with picking the start for this path, because other paths can be longer than this one, so the tiles might be valid for those.

If the minimum distances between path starts and from the *Hub* are small enough, this approach always yields a valid collection of path starts. However, with stricter parameters it is possible that the first few starts invalidate all other start positions. In that case we could use rejection sampling again — trying to randomly select collections of starts, until we get one that's valid. If the failure rate is great enough, it would be wise to select a different algorithm.

We choose the parameters based on the number of paths and their lengths, such that the path starts are sufficiently spread out for our use-case, and start picking never fails for 1 to 8 paths of lengths 10 to 100, which is a wide enough range for our needs.

3.3.2 Random Walks

In the current and the next (3.3.3) section, we will describe two approaches we tried to use for generating the main paths. The one we ended up using will be described in the section 3.3.4.

For generating the main paths, the simplest approach that comes to mind is to just do a random walk from the start to the end. Of course, we have some rules the paths need to follow, so we need to tweak the algorithm. First, the path cannot intersect itself, so we have to ban moves onto a tile where we've already been. We also need to ensure the path is the correct length. To do so, we can count how long is the part of the path that is yet to be generated, and ban the

option to go on a tile that's further away from the *Hub* than the remaining length. This means we have to recalculate the shortest distance to the *Hub* for every potential step we would take, since the path creates walls for itself.

Then there are all the suggestions about path spacing, which we can try to solve by introducing a *crowding* penalty. Whenever we mark a tile as containing path, we also increase the *crowding* penalty of tiles around it by an amount decreasing with distance from the original tile. The more crowded a tile is, the less likely it is to be selected during the random direction selection. We can also add a small amount of *crowding* penalty around the edges of the world to steer the paths away from the edges.

We have tried this approach and did many more tweaks, but the paths were always problematic. They were always too bunched up either at the start or at the end. With multiple paths, two often stranded the path in between them. This approach also did not work for long paths, where the path usually blocked itself off from getting to the *Hub*. So, we tried a different approach.

3.3.3 Selecting Points in Between

The previous approach selected the first tile of a path, then the next, and so on. We thought it might be better to select some tiles that will be in the middle of the path, then select tiles in between those, and continue refining the path until all of its tiles are selected.

In the algorithm we used, the path network can be viewed as a graph. Each node has a distance *dist* and a tile position *pos*. No two nodes can share a tile. The *length* of an edge connecting nodes a and b is defined as $|a.dist - b.dist|$. A simplified version is described in pseudocode as algorithm 1.

The procedure GetValidPositions(a, b, d) returns a set of valid positions for a new node with $dist = d$ that would connect to nodes a and b. For simplicity, let's say that a position p is valid, if we could travel from $a.pos$ to p in $|a.dist - d|$ steps, and from p to $b.pos$ in $|d - b.dist|$ steps, without going on a tile which already has a node on it. The details of how to determine this are omitted. The procedure Remove(n) connects the two neighbors of n with an edge and removes n. The procedure Subdivide(e, n) takes the nodes a, b connected by e and connects each of them to n with new edges. Then it removes the edge e.

Algorithm 1 Generating paths by selecting points in between.

$h \leftarrow$ create a node with $pos \leftarrow hub\_pos; dist \leftarrow 0$
for each path p **do**
    $s_p \leftarrow$ create a node with $pos \leftarrow p.start\_pos; dist \leftarrow p.length$
    create an edge between s_p and h
end for
$initial \leftarrow \{h, \text{all } s_p\}$

while $\exists$ edge with $length > 1$ **do**
    $e \leftarrow$ random edge with $length > 1$
    $a, b \leftarrow$ nodes connected by e
    $d \leftarrow$ random integer strictly between $a.pos$ and $b.pos$
    $valid\_positions \leftarrow$ GetValidPositions(a, b, d)

    if $valid\_positions = \varnothing$ **then**
        if $a \notin initial$ **or** $b \notin initial$ **then**
            $n \leftarrow$ random node from $\{a, b\} \setminus initial$
            Remove(n)
        end if
    else
        $p \leftarrow$ random position from $valid\_positions$
        $n \leftarrow$ create a node with $pos \leftarrow p; dist \leftarrow d$
        Subdivide(e, n)
    end if
end while

This algorithm inserts new path nodes between others, incrementally refining the paths. But once a new node cannot be inserted, it backs up by removing one of the nodes that are already present. This way, it can try different configurations, until it is finally able to complete the paths.

However, it is possible for this algorithm to never finish when it gets to a state from which a valid path cannot be created. For example, this will happen when two paths completely separate another path's start from the *Hub*. We can partially mitigate this issue by letting the algorithm run only for a limited number of steps. If it doesn't finish before then, we restart the whole process, including picking new path starts.

We have used this algorithm and extended it for various heuristics to make it produce better paths (according to the design requirements) and succeed more often. It worked somewhat well, but it really struggled with long paths, being unable to create paths over 50 in reasonable time. This lead us to search for a simpler and better alternative.

3.3.4 Simulated Annealing

The algorithm we ended up using is based on an optimization technique called *simulated annealing*. A great description of this technique can be found in the

article *Optimization by Simulated Annealing* [19]. Here, we will describe the main ideas, and then we will describe the algorithm we used to generate paths.

Simulated annealing can be used to find an approximation of the global optimum much faster than it would take to find the exact global optimum. The problems it can be used on 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, in the *travelling salesman problem*, each state is usually defined as a permutation of the cities to be visited. The cost function 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 in algorithm 2. 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 1 in the first step, to 0 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 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 ($f(s') > f(s)$). How often a worse state is accepted depends on the temperature. At high temperatures, almost any state is accepted, at temperatures near zero, worse states are accepted very rarely. Overall, the algorithm explores widely different states at the start, but settles into a local minimum over time, which is hopefully a global minimum thanks to the exploration at the start.

Algorithm 2 Simulated annealing

```
1: s ← s_0
2: for k from 0 to max_steps − 1 do
3:     t ← 1 − 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
```

To generate paths using simulated annealing, we need to define what is a state, a cost function and a neighbor function. Each state will be a path network, where each path is a sequence of nodes. Each node has a tile position, and each tile can contain multiple nodes. Two consecutive nodes must be on neighboring tiles. Additionally, each path starts at the given path start, has the given length, and ends at the tile with the *Hub*. Since we are going to change the paths only by a small amount at every step, we chose not to check for intersections. Otherwise, we would lose too much freedom during the simulated annealing, and the final state would always end up close to the initial state.

The neighbors of a state are the states we obtain by changing the position of single node to a different position, such that the result is still a valid state. This

set is not too difficult to generate. First, we can notice that the first and last node of every path can never change position. Additionally, any node can only ever change its position to a diagonally adjacent one, or horizontally by two tiles, based on the nodes immediately before and after, as shown in figure 3.1. We only draw the nodes before and after the node which we want to modify, because the other nodes are not relevant.

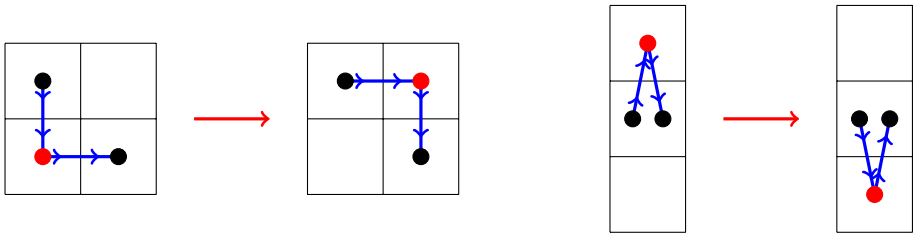


Figure 3.1 Possible node position changes to create a neighbor state.

Now we select a cost function that gives a better score to paths that are more desirable. A great starting point is to calculate a crowding penalty for each tile as we did for our random walks algorithm (see section 3.3.2). We could then calculate the cost function of a state as the sum of the crowding penalties of each tile for each node on it. However, to save on computation, we don't compare the values of the current and next state. Instead, we only calculate the *relative improvement* of the position change as $crowding(\text{crurrent position}) - crowding(\text{future position})$ without even calculating the new crowding penalties. We only recalculate the crowding penalties for a state after we select it as the new state. This means that the relative improvement is a very crude approximation of the real relative improvement, and it is biased towards switching the position, because the current position has greater crowding penalty from the node itself, while the future position is not affected as much. Later, we will show how we changed the algorithm to mitigate this bias.

All that's left is to produce an initial state. This is not trivial, because of our constraints on what's considered a valid state. However, we can easily produce a valid initial state using a random walk, as described in section 3.3.2. This time we don't even have to check for self-intersection.

We used a slightly different version of the algorithm, as seen in algorithm 3. Before (algorithm 2), temperature went from 1 to 0, however it can have any initial and final value without loss of generality. Here, we specify this explicitly, because we use the temperature directly in the calculation of the *improvement* a change would lead to. We use the temperature to offset the bias in our *improvement* function, by setting the final temperature to an adequate negative value. This value does not have to be exact, because the temperature changes, so it will always be too high or too low anyway. Furthermore, we always move to a new random state, but better states are more likely to be selected. This is almost equivalent to doing multiple steps of the original algorithm until it moves to a new state, without actually doing them.

Algorithm 3 Simulated annealing for generating paths

$s \leftarrow$ GenerateInitialPaths
$crowding \leftarrow$ CalculateCrowding(s)

for k from 0 to $max\_steps - 1$ **do**
    $t \leftarrow$ Lerp($t_{initial}$, t_{final}, $k/(max\_steps - 1)$)

    $candidates \leftarrow \varnothing$
    for each node swap $x = a \to b$ in GetPossibleNodeSwaps(s) **do**
        $improvement \leftarrow crowding(a) - crowding(b) + t$
        if $improvement > 0$ **then**
            $candidates \leftarrow candidates + (x,\ improvement)$
        end if
    end for

    $x \leftarrow$ random node swap from *candidates* weighted by *improvement*
    $s \leftarrow s$ with x applied
    $crowding \leftarrow$ UpdateCrowding($crowding$, x)
end for
return s

However, this algorithm still sometimes produces paths that intersect themselves. This is because it is difficult for the algorithm to fix a loop in the path, as shown in figure 3.2. However, we can add a step that *untwists* crossings by reversing the section of the path that forms a loop, as shown in the figure. This still leaves two nodes on the same tile, but annealing can drive these apart without any issue.

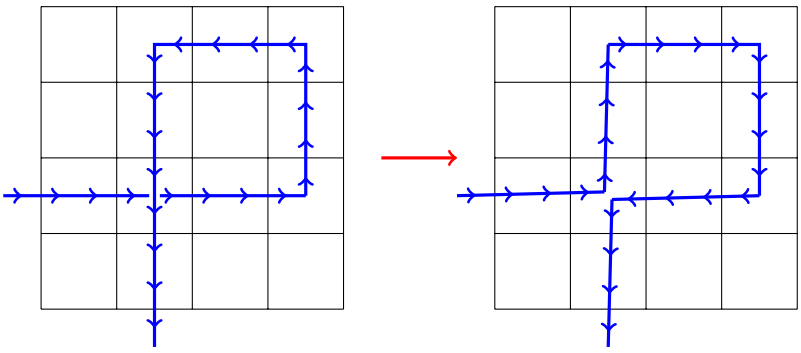


Figure 3.2 Untwisting a self-intersecting path.

To only accept valid path networks, we can modify the algorithm to return the last valid path network, if any. In case no valid network is generated, we can restart the path generation algorithm, including picking the starting positions.

This leaves us with an algorithm that can generate high quality path networks often, even with long paths. From our testing, $0.5l^2$ steps, where l is the total length of all paths, is usually enough for high quality paths. Usually, when the algorithm is unable to create a valid path network in $0.05l^2$ steps, it will fail to do so even in $0.5l^2$ steps, given the same initial state. We modified the algorithm

to fail early if it doesn't find a valid path network in $0.05l^2$ steps, and try again without wasting more time.

3.3.5 Final Paths

Now, that we can generate the main paths, we just have to generate the side branches. It is useful we don't have many requirements on the count or length. We can generate the rest of the world first, and fill in more paths after that. Since the world generation is random enough, our algorithm can be relatively simple and still lead to widely varying results.

After generating the world, some tiles are blocked by obstacles, so a path cannot go through them. Additionally, some edges between tiles are also blocked, usually because the two tiles are at different height levels, separated by a cliff. The rest of the world generation has made sure that at least the main paths are still traversable, however we don't have to adhere to them completely. In fact, we can try to make the paths follow the rules even better, but we can always fall back on the previously generated ones.

The first step of the algorithm we came up with, is to compute the shortest distance from each tile to the *Hub*, respecting the blocked tiles and edges. For this, we use breadth first search from the *Hub* tile, but the paths start with the path distances already filled in, and we disallow the algorithm from visiting these tile sooner than their already marked distance. The result can be seen in figure 3.3.

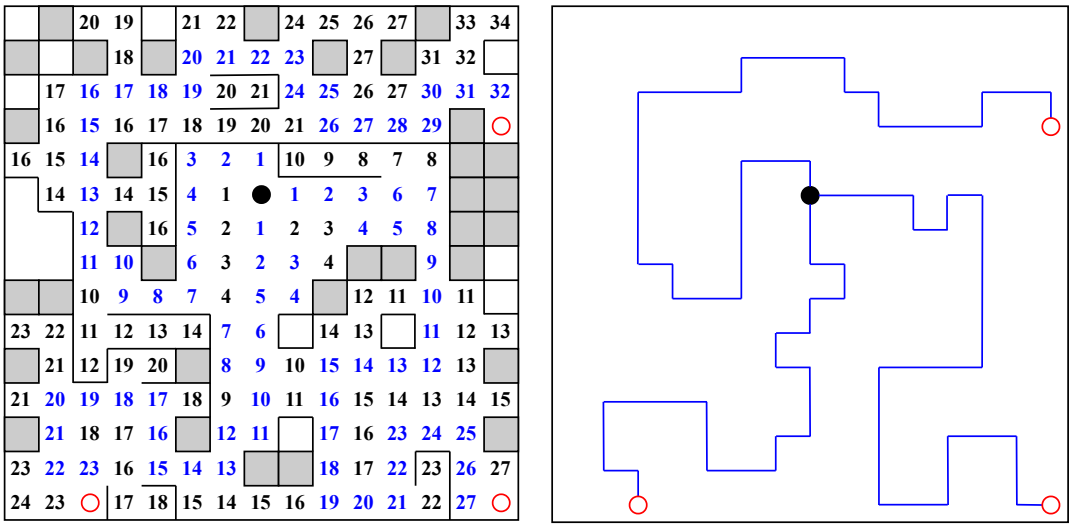


Figure 3.3 Calculated tile distances compared to the originally generated paths.

It then traces paths one by one from the start, basically doing a depth first search. At every step, it pops the last item on a stack of path prototypes, then it finds all valid one-step continuations, and it pushes them onto the stack. Which continuations are valid and how they are ordered will be explained later. Once it reaches the *Hub* or an already existing path, it checks whether it is valid to finish the current path. If this check fails, it pops the last item on the stack and continues from there. If the check succeeds, it marks the new path section and continues by making another branch, this time taking the first item in the stack. We do this to minimize the length of path the new branch shares with the branch it came from. This process is described with pseudocode in algorithm 4.

Algorithm 4 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 position in p contains a path or the Hub then
11:            success ← TryFinishPath(p)
12:        else
13:            success ← false
14:            for each position c from GetValidContinuations(p) do
15:                stack.PushLast(p extended by c)
16:            end for
17:        end if
18:    end while
19: end for
```

For valid continuations of a path prototype, we consider the neighbors of the last position in the path prototype. A tile is a neighbor of another tile, only if the passage between them is not blocked and the tile itself is not blocked. Additionally, the distance we computed for this tile before, must be less than the target path length minus the length of the current prototype. In other words, if the path prototype has to go n more tiles before it can connect to the *Hub*, this tile's distance from the *Hub* must be at most n. To generate the best paths, we order the valid continuations such that the most preferable one is popped from the stack first. Again, we can use crowding penalties to encourage paths to spread out. However, above that we will prioritize the continuation that goes straight — in the same direction as the last step in the path prototype so far. With this ordering, 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 exactly the right distance to the *Hub*, which was denoted as n in our previous example.

Now, How do we know when it's valid to finish a path prototype when it connects to an existing path or the *Hub*? First, it must have the correct length. If it does, and the path start this prototype originates from is not connected to the network yet, it is valid. However, for branches other than the first, the requirements are stricter. As per the last rule in the summary of section 2.3.3, every side branch must go through at least one tile that is not adjacent to any already existing path.

As we mentioned at the start of section 3.3, we also get a maximum number of extra branches as a parameter for path generation. Thus, once the algorithm

reaches the maximum count, it stops. The results of the algorithm are displayed in figure 3.4, again, compared to the initially generated paths.

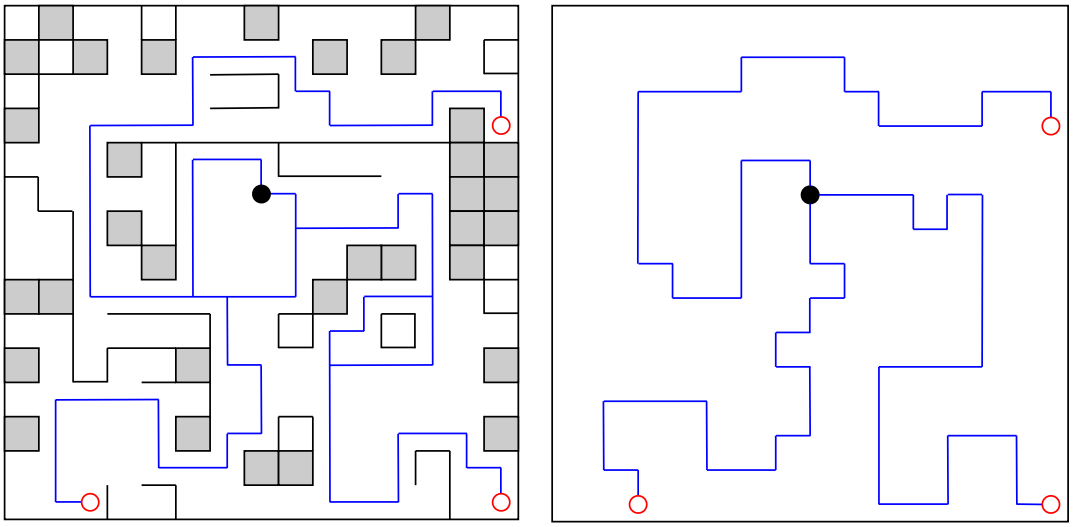


Figure 3.4 Final paths compared to the originally generated paths.

3.4 Terrain Generation

There are many techniques we could use for generating the terrain. We chose a variant of *model synthesis* originally developed by Merrell [20]. The discrete version of this algorithm is better known by the name *wave function collapse* (or WFC in short), popularized by Gumin on GitHub [21]. 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 original name. Before we explain why we chose to use a variant of this algorithm, we would like to explain how it works.

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 3.5. We will limit our examples to 2-dimensional grids of tiles, however this algorithm works in more dimensions, and even for irregular cells.

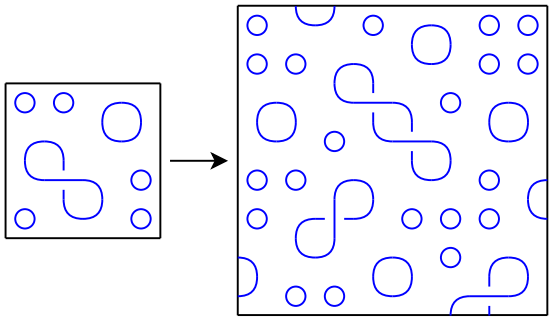


Figure 3.5 Example input and output of the wave function collapse algorithm.

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 3.6, 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.

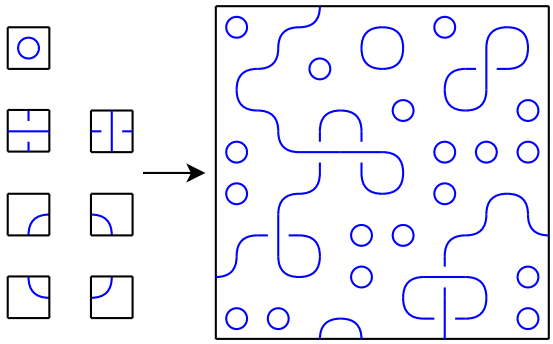
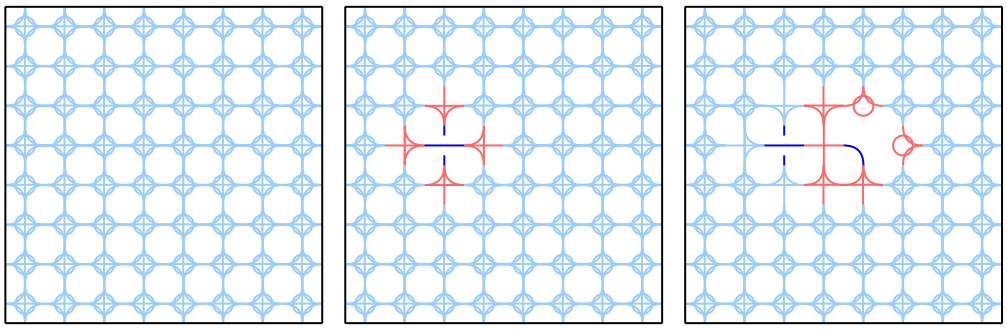


Figure 3.6 Example output of WFC, using the modules on the left and only the constraint that their edges must match.

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 3.7a 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 3.7b 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 3.7 Two steps of wave function collapse. Uncollapsed slots are drawn in light blue, uncollapsed slots that changed are drawn in red.

[1]This is a naming convention used in an article by Marian [22].

In figure 3.7c, 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 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 5. 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 TODO. We will call this algorithm WFC, even though it differs from WFC by Gumin. The most notable difference is that we skip the feature extraction and take as input modules directly.

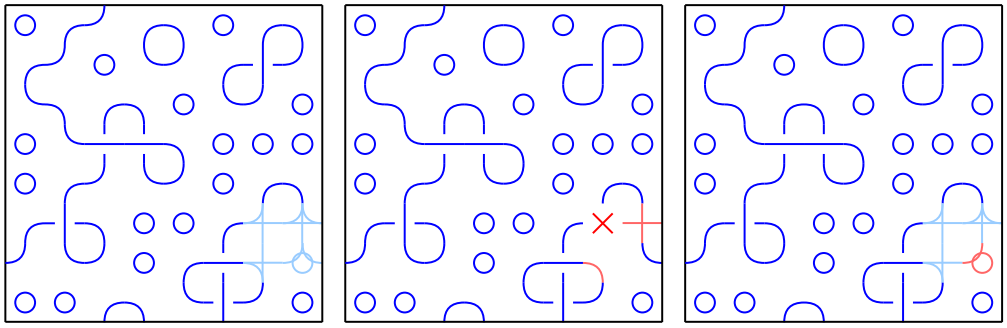
Algorithm 5 A naïve version of wave function collapse

```
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 this case, it is no longer possible to create a valid output. An example can be seen in figure 3.8. 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.

One option is to simply restart the algorithm. For sufficiently small outputs, this 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 and removes the module the slot collapsed to from its valid options. The state after backtracking is illustrated in fig 3.8c. This way, the algorithm can continue generating without getting rid of all the progress.



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

Figure 3.8 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 allowed them to exist. 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 scales poorly. Merrell shows in their thesis [20] 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 isn't very fast in the worst case. In section 3.4.4, we show that algorithm 5 runs in $O(k^3 \cdot n)$ time, where k is the number of modules, and n is the number of slots. This time complexity also holds for a version which can handle contradictions, assuming it never runs into any, but each contradiction can only slow the algorithm down.

The real problem is that the algorithm usually does run into contradictions. In fact, the larger the generation task, the less likely it is to succeed. 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 their thesis, Merrell describes a technique called *modifying in parts* based on this approach.

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, only based on the modules chosen by WFC. Another way to make the slots less apparent is to make them irregular. Both of these techniques are used by *Townscaper* [23], a game by Oskar Stålberg.

Also, WFC uses only 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, or an instance of WFC with larger slots.

Luckily, none of these disadvantages matter for our use case. Our worlds are very small, and we don't mind that the tiles will be apparent, since the gameplay of our game is centered around them.

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.

At first, it might seem sensible to have one slot per world tile. However, each tiles on it sown 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 3.9, such that each slot is responsible for four quarter-tiles of the world. This way, the modules dictate how can the adjacent tiles connect to each other.

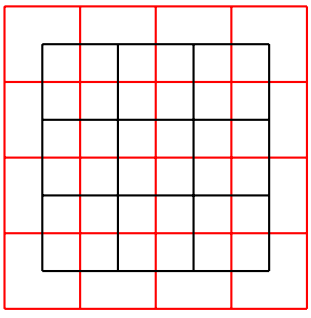


Figure 3.9 The slots for generating a 3×3 tile world. Tiles are drawn in black, slots in red.

One example of such module is shown in figure 3.10. 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. For each terrain type, it is also specified which surface types and which edge types block paths.

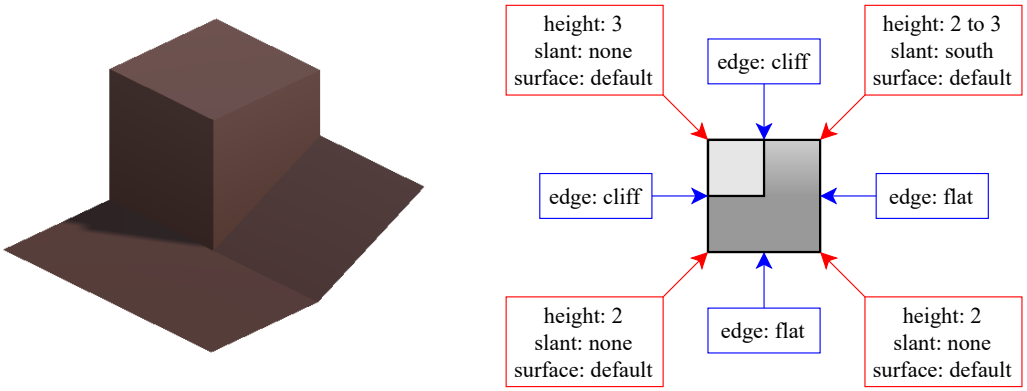


Figure 3.10 An example module and its constraints.

In figure 3.6, 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 3.10 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.

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 run some quick tests with the set of modules which is going to be used to generate terrain in the demo version. During these tests, we were unable to generate a single world, without running into a contradiction. However, most worlds only ever needed backtracking depth of 1 to successfully generate. After that, we ran into diminishing returns. Of the worlds that required backtracking depth 2, most required backtracking depth greater than 10. So we decided to allow for backtracking depth 1 and otherwise just restart the generation, which makes it faster on average.

I should probably provide the data I measured. Do I also need to explain the whole setup to make it reproducible?

We also need to decide which slot to collapse at each step of the algorithm. Merrell uses in their thesis a sort of scan line order, collapsing them in the lexicographic order of their coordinates. This could theoretically introduce a

directional bias in the results. Gumin in their implementation always collapses a slot with the lowest *Shannon entropy*. However, from our testing, the results tend to look unnatural, often creating rectangular regions at the same height or repeating patterns.

To make the results more natural, we chose to select the slot at random. This means that the generator first sets slots in various parts of the world to different heights. Then it has to somehow connect these to make the world follow the rules we set. This leads to more diverse results, however, the failure rate becomes very high, leading to slower generation.

As a compromise, we chose to weight the slots by the number of invalid slots $+1$. This is an approximation of entropy that is trivial to compute, making it more likely to select more constrained slots, but still collapsing unconstrained slots once in a while. The 1 is added just to make all weights nonzero. This leads to results similar to the ones with uniform randomness, but decreases the failure rate substantially.

Can I show screenshots of the generated worlds with different settings? More testing data to compare the success rate of different approaches

3.4.4 Time Complexity of Naïve WFC

In this section, we will show that algorithm 5 runs in amortized $O(k^3 \cdot n)$ time, where k is the number of modules, and n is the number of slots. We will assume that the sets *uncollapsed*, *to_update* and the *modules* of each slot have amortized $O(1)$ queries, additions and removals (including Pop). Another assumption we will is that IsValid runs in $O(k)$ time, because each slot has $O(1)$ neighbors and each neighbor can only have k different modules, so we have to check at most $O(k)$ constraints.

This means that checking the modules a slot can have takes $O(k^2)$ time. Since each change removes at least one module, each slot can change up to k times, and each change leads to $O(1)$ neighbors being checked. There are only n slots which means at most kn changes, which means constraint propagation takes $O(k^3 \cdot n)$ time over the execution of the whole algorithm.

All other steps can are asymptotically faster: Initialization is done in $O(kn)$ time. Collapsing n slots takes $O(kn)$ time, assuming picking the random module takes $O(k)$ time.

3.5 Resources and Obstacles

- after terrain generation, place blockers on tiles
- materials for the player to mine
- just rocks for variety - the player cant build on these
- set up as a few layers
- each stage has:
- one or more types of blocker (e.g. ore, small rocks, big rocks)
- *min* and *max* amounts
- *base chance* to place
- whether they can be placed on slanted tiles
- which surface Types they can be on (currently there is only one)

- *forces* - effect on chance based on already placed blockers

- for example: negative force with magnitude *m* from stage *s* means the chance to place a blocker on a given tile is decreased by *m/d* for each blocker placed in stage *s*, where *d* is its distance from the considered tile

- for each stage:

- repeat until at least min blockers have been placed (in this stage)

- for each tile without a path or blocker (in random order):

- if random number between 0 and 1 < modified chance:

- place the blocker of the given type

- if there are max blockers (placed in this stage), end the stage

- scattering models

- unity physics engine X

- parallel

For the blockers, I didn't want repetitive obstacle models, so they are generated proceduraly by scattering many simpler models (decorations) on each tile

- first compute weights based on various factors (images!!!)

- distance to path

- height

- distance to other blockers

- customizable thanks to modular approach

- then scatter decorations in stages, each stage again having one type of decoration and many parameters

- for each tile in random order repeat x (specified for this stage) times:

- pick a random position within it

- calcualte the weight at this position (based on settings)

- check that it is greater than some threshold (based on settings)

- calculate the minimum distance to other decorations (from weight, based on settings)

- check that the position is far enough from other decorations

- calculate the decoration size (from weight, based on settings)

- place the decoration on this position, with the given size

3.6 Terrain Types

- what information is tied to the type

- why txt (inspector was not as legible)

3.7 World Builder

- builds the world from the generated data, it needs to be done in the main thread

- the rest will be in background threads

3.8 Attacker Wave Generation

- creates a randomized plan of waves

- two types of waves
- combine different attackers in sequence
- combine different attackers in parallel (only possible with multiple paths, rarer)
- each wave gets some throughput budget and buffer
- each attacker has a given cost
- when planning a wave, select attackers and spacing, such that the througput budget is exceeded
- for each attacker subtract the throughput overshoot from buffer
- fit such that as much of buffer gets used without going over
- branching

3.9 Simulation

- use fixed updates for game logic
- why?
- 20Hz = fixed time step 0.05s
- options to speed up or possibly pause - changing fixed update rate - not yet implemented

3.10 Visuals and Interpolation

- interpolate positions and visuals on Update
- many visuals are game-speed agnostic - TODO: use unscaledDeltaTime
- I thought about some custom mini-framework for this, but many of the simulated variables the visuals are based on should be handled on case-by-case basis

3.11 Attacker Targeting

- Towers use it to acquire targets
- handles which Attackers are in range and which one is chosen as the current target
- can require line of sight to the enemy
- different targeting types
- rotation
- heights
- possibly ensure a trajectory
- preferred target (configurable)
- composite colliders

3.12 Range Visualization

- IMAGES!!
- Draw the range on the terrain mesh
- Draw on which parts of paths will Attackers be targeted

- green - all sizes
- yellow - only large
- Terrain shader uses compressed texture format instead of raw texture
- Options:
- quadrant compression format, 2bytes per node
- less CPU time, because the data is already in this format
- up to 48KiB per frame
- more GPU time
- 256x256 texture, 1byte per pixel
- more CPU time
- 64KiB per frame
- fast on GPU
- only 1 channel - cannot interpolate
- mipmaps -> one additional state
- less CPU time
- 33% more data
- more pixels per byte
- possible future optimization
- less data
- more difficult indexing and stuff both on CPU and GPU
- interpolation could work with more than one channel and without mipmaps

3.13 Game Commands

- we want various components to modify how other components function
- examples
- also react to events as a bonus

3.14 Blueprints

- separation of stats from behavior
- why are they implemented this way

3.14.1 Attacker Stats

- blueprints for attackers

3.14.2 Dynamic Descriptions

- explain what things do and their stats
- attackers and blueprints
- dynamically reflect the changes made by other components

4 Developer Documentation

4.1 Unity

4.2 Scenes

4.2.1 Battle

4.2.2 Loading

4.2.3 Main Menu

4.3 ???

5 User Documentation — Designer

5.1 Terrain Types

5.2 Blueprints

5.2.1 Buildings

5.2.2 Towers

5.2.3 Abilities

5.3 Attackers

5.4 ???

6 User Documentation — Player

6.1 ?Introduction

6.2 ?Controls

6.3 ?Mechanics

7 ?Playtesting

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. Wikipedia contributors. *Slay the Spire — Wikipedia, The Free Encyclopedia* [online]. 2024. [visited on 2024-05-06]. Available from: https://en.wikipedia.org/w/index.php?title=Slay_the_Spire&oldid=1221499787.
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. 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.

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

17. 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.

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

19. 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.

20. 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].

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

22. 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/.

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.

List of Figures

List of Tables

List of Abbreviations

A Attachments

A.1 First Attachment