



**FACULTY
OF MATHEMATICS
AND PHYSICS**
Charles University

BACHELOR THESIS

Name Surname

Thesis title

Name of the department

Supervisor of the bachelor thesis: Supersurname 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: Supersurname 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: Supersurname Supersurname, department

Abstrakt: Abstrakt práce přeložte také do češtiny.

Klíčová slova: klíčová slova, klíčové fráze

Contents

1	Introduction	7
1.1	Tower Defense	7
1.2	Roguelike	8
1.3	Original Vision	10
1.4	Current Scope and Goals	11
2	Game Design	12
2.1	Design goals	12
2.1.1	Strategic Depth in Every Battle	12
2.1.2	Strategic Depth in Every Run	13
2.1.3	Make Various Builds Viable	14
2.1.4	Force Exploration	16
2.1.5	Provide a Challenge	17
2.2	Procedural Generation	18
2.3	Battle	18
2.3.1	Attacker Behavior	18
2.3.2	Attacker Paths	21
2.3.3	World	23
2.3.4	Buildings	23
2.3.5	Towers	23
2.3.6	Economic buildings	23
2.3.7	Abilities	23
2.4	Battle Graphical User Interface	23
2.4.1	Waves Left and Fuel	24
2.4.2	Hull	24
2.4.3	Wave Preview	24
2.4.4	Materials and Energy	24
2.4.5	Blueprints	24
2.4.6	Info Panel and Selection	24
2.4.7	Highlights and Range Visualization	24
2.5	Attackers	24
2.6	Camera controls	24
2.7	Future Features	24
3	Analysis	25
3.1	Game Engine	25
3.2	Procedural Generation	25
3.2.1	Random Number Generators	25
3.3	Path Generation	27
3.3.1	Initial Paths	27
3.3.2	Final Paths	27
3.3.3	Path Visualization	27
3.4	Terrain Generation	27
3.5	Resources and Obstacles	29
3.6	Terrain Types	29

3.7	World Builder	30
3.8	Attacker Wave Generation	30
3.9	Simulation	30
3.10	Visuals and Interpolation	30
3.11	Attacker Targeting	30
3.12	Range Visualization	31
3.13	Game Commands	31
3.14	Blueprints	31
3.14.1	Attacker Stats	31
3.14.2	Dynamic Descriptions	31
4	Developer Documentation	32
4.1	Unity	32
4.2	Scenes	32
4.2.1	Battle	32
4.2.2	Loading	32
4.2.3	Main Menu	32
4.3	???	32
5	User Documentation — Designer	33
5.1	Terrain Types	33
5.2	Blueprints	33
5.2.1	Buildings	33
5.2.2	Towers	33
5.2.3	Abilities	33
5.3	Attackers	33
5.4	???	33
6	User Documentation — Player	34
6.1	?Introduction	34
6.2	?Controls	34
6.3	?Mechanics	34
7	?Playtesting	35
	Conclusion	36
	Bibliography	37
	List of Figures	38
	List of Tables	39
	List of Abbreviations	40
A	Attachments	41
	A.1 First Attachment	41

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], published in 2019. 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.

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, that we think 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 subsection, 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 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 material.

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.

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

- roguelike -> procedural generation
- to what degree?
- map will be random
- we want to procedurally generate levels
- terrain types
- balance levels - cannot feel unfair
- in sts, encounters are handcrafted, but picked randomly
- still has variety
- easy to balance
- generated waves
- attacker types
- hard to balance - must be described by a formula

2.3 Battle

As stated before (see 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 decisions.

2.3.1 Attacker Behavior

In *Plants vs. Zombies*, the zombies come continuously throughout a level, but only in small numbers. And then there is a few marked waves in which a lot of zombies come at once. In other games, the attackers come only in distinct waves, but the waves come right after each other based on a timer. One example of such a game is *Kingdom Rush* [9]. In figure 2.5, a skull icon in a circle is shown at the top of the screen with the text “INCOMING NEXT WAVE! CLICK TO CALL IT EARLY”. The ring around the skull fills with red color over time and once it is full, the next wave of attackers comes automatically. Both of these options make a game more action-packed. However, we will go with a third, also often used option, where the attackers come only in waves, but there is no timer. The player will have plenty of time to plan out their strategy, and they start the next wave of attackers when ready. We feel this fits our game more, making it more similar to the turn-based gameplay that is often featured in roguelike games.



Figure 2.5 A screenshot from *Kingdom Rush*, featuring a wave timer.

Different tower defense games also use various types of attacker movement. 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. Each zombie is confined to its lane and most of the plants available to the player only affect the zombies in front of them in the same lane. The plants are planted directly in the way of the zombies and the zombies have to eat their way through the plants to reach their goal. This is, however very restrictive for the design of the towers in our game, and there is little depth to the tower placement. We can let the attackers move in two dimensions to allow for more interesting tower placement decisions.

In *Desktop Tower Defense* [10], for example, the attackers move from sides of the rectangular playing field to the sides that lie across. The playing field 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, as shown in figure 2.6. Since the player decides the path of the attackers, they have to learn what kind of path works well, but then they can use it 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. Also, a path always has to exist through, so *Desktop Tower Defense* doesn't let the player place a tower that would disconnect the attacker entrance from their goal. Some games allow the player to block off the path completely, and then the attackers attack the player's towers, similarly to *Plants vs. Zombies*. But for our game that would be just more unnecessary complexity, since we want to focus more on the offensive abilities of the towers.



Figure 2.6 A screenshot from *Desktop Tower Defense*.

Instead, we chose an approach that is most common in tower defense games — the attacker paths are predetermined, and the player builds their towers around the paths. The paths are different in every level, so the player has to think about placement of their towers differently, and it makes different towers more useful than others in each level. This is illustrated well on figure 2.7, where we can see two levels from *Bloons TD 6* [11]. 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, which pierce through many attackers in a straight line.



Figure 2.7 The levels *Park Path* and *Another Brick* from *Bloons TD 6* with the attacker paths highlighted.

More variety can also be added by having multiple paths in a single level. We can even have different attackers on each path in some waves to further increase variety. The paths in our game will also be able to split into more paths and join together into one path. 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. This creates segments with lower attacker density, which makes

certain towers less effective around this part of the path. On the other hand, this lets the player fit more towers near these two paths than they could fit around the path if it didn't split.

2.3.2 Attacker Paths

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. For reasons that will be explained in the next subsection, the paths and towers will be confined to a square grid of tiles. Let's start with a few basic rules:

- A level can have one or more paths, and paths can split apart or join together.
- Each path will be composed of segments that go from the center of one tile to the center of an orthogonally adjacent one.
- The paths will start just outside the world and end at the tile with the *Hub*.

In figure 2.8 we can see an example of a valid path drawn in blue on a backdrop of the square tiles the world is composed of. The *Hub* is represented by a black dot.

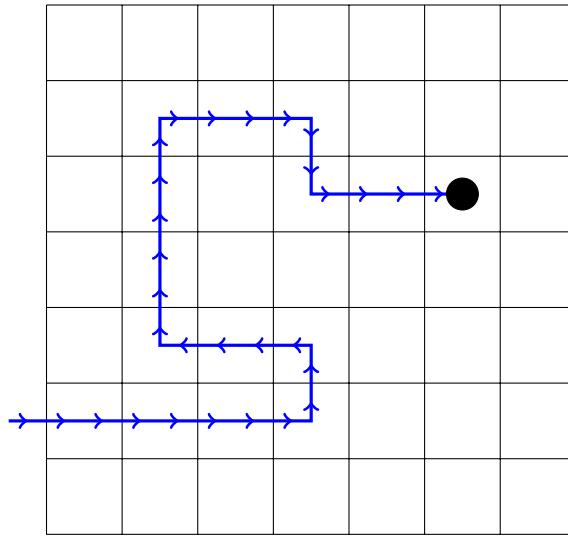


Figure 2.8 An example of a valid path in the game world.

However, we need more rules, since right now, nothing forbids a path going back on itself. This is illustrated with path **A** in figure 2.9, where we offset the endpoints of few segments to make it more readable. Paths segments have to start and end in the centers of the tiles, so in reality, it would look like path **B**, which is definitely unreadable.

Furthermore, no path should cross itself or another path. We want this because the paths can split and join, and we would need a way to distinguish these splits and joins from crossings. This is done to make sure the paths are readable to the player. For example, the paths **C** and **D** in figure 2.9 could very much be crossing each other, but they could also join together and split in two on the same tile.

Furthermore, path **E** is ambiguous, because it could cross itself, but it also looks almost as if it split and joined with itself on the same tile.

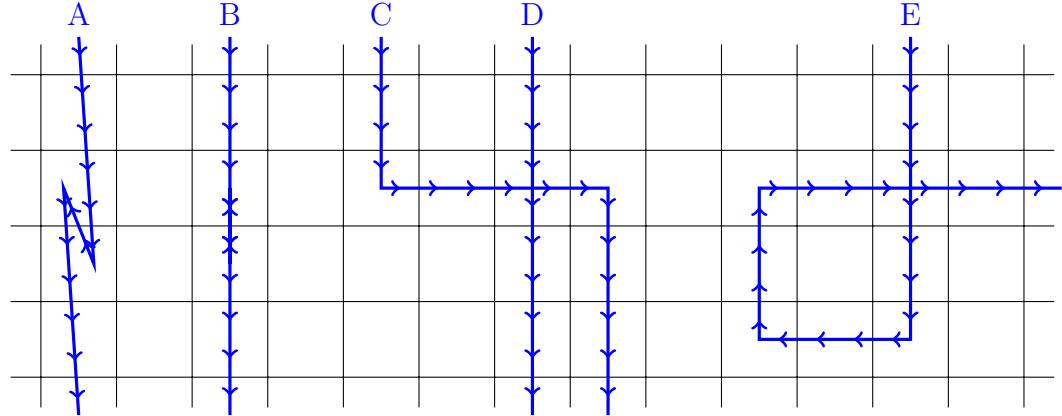


Figure 2.9 Several paths that would be confusing or ambiguous if paths could cross.

Path **F** in figure 2.10 is however also problematic. It does not cross over itself, but if we were to center all the segment endpoints, it would look almost like path **E**, which would be equally confusing. We will redefine make no-crossing rule stronger and define it as the following:

- Each path segment makes a one-way connection from a tile to its neighbor. The distance of each tile connected to the *Hub* must be the same, no matter which path an attacker takes.

This makes path **F** also illegal, since it goes through a tile twice, and the second time it visits it, it is 8 segments closer to the *Hub* than before. This also has implications for paths that split or join, for example path **G** is invalid, as demonstrated by the distances shown in red, while the path network **H**, **I** is valid. The new rule helps here, because we don't want the spacing between attackers to change only because some took a longer path before merging with the attackers that took a shorter path.

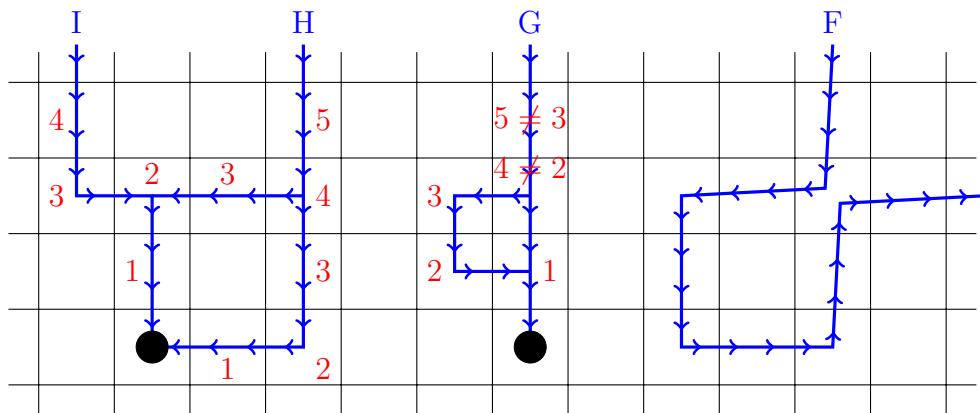


Figure 2.10 Several paths illustrating the rule that each tile can only have one distance to the *Hub*.

- paths near each other are undesirable
- tight turns are undesirable
- short splits are banned
- fill the world as much as possible

2.3.3 World

- grid of tiles ✓
- more granularity, easier to develop intuition, same for attacker paths
- free placement X
- preset places
- 3D ✓ - Simple and intuitive way to make the level itself more interesting
Some towers won't be able to shoot uphill or downhill - more interesting decisions for the player
 - Terrain types (for now, just one)
 - Obstacles

2.3.4 Buildings

- how and when to build, one per tile - only between waves
- special building have special abilities
- main types:

2.3.5 Towers

- attack attackers
- range
- targeting
- projectile types
- damage types

2.3.6 Economic buildings

- provide resources
- materials - what, why, consequences and constraints
- fuel?

2.3.7 Abilities

- used mid-wave
- usually instant effects
- free placement, global placement, tile placement, use on a building
- energy - what, why, consequences and constraints

2.4 Battle Graphical User Interface

specify controls, use images

2.4.1 Waves Left and Fuel

2.4.2 Hull

2.4.3 Wave Preview

2.4.4 Materials and Energy

2.4.5 Blueprints

- what they represent
- limited number
- cost and cooldowns
- rarities
- make them unique
- lenticular design

2.4.6 Info Panel and Selection

- what it looks like and what's on it
- select blueprints
- select buildings
- select attackers

2.4.7 Highlights and Range Visualization

2.5 Attackers

- move, have health
- sizes
- special abilities - passive, repeating, reactive

2.6 Camera controls

- zoom to look closely, rotate so the terrain doesn't hide stuff

2.7 Future Features

- setting
- run structure
- map
- events and shops
- saving the game
- unlocks
- difficulty levels

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 [12].

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 will look identical and if the player makes the same decisions and choices, the outcomes will be the same. This allows the players to share seeds of the runs they found interesting and compare their skill without the influence of luck. 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. This leads to simpler and smaller save files, but we might still want to save some of these components whole when the procedural generation takes significantly longer than just reading the serialized data.

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* [13] 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` [14]. 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` [15]. 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 [13] 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, given 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” [16], the author explains the statistical tests they used to measure the randomness of the LCGs and tabulates the best-performing parameter combinations. From here we took the parameters for our LCG implementation.

3.3 Path Generation

I should probably write the design part first so I can build on it here - how to make paths with the required qualities?

3.3.1 Initial Paths

- generate fixed number of paths with given lengths
- these are just plans to ensure paths of required length exist
- first pick *start points* from along the edges of the level (all paths end in the center of the level)
 - choose randomly from positions with the correct parity (even / odd path length)
 - spread them out by removing all positions near already chosen one
 - then generate paths
 - first trace it randomly, with bias away from the start and from path tiles
 - simulated annealing
 - there are just plans to make sure the paths exist and the terrain in a way that ensures that the paths are not blocked

3.3.2 Final Paths

- after terrain generation, the paths are traced for real
- guaranteed shortest length
- DFS, find all branches, but continue from bottom of the stack when a paths is found

3.3.3 Path Visualization

- line renderer ✓
- why does it look this way

3.4 Terrain Generation

- fractal noise X
 - WFC ✓(don't forget to mention disadvantages)
- this is gonna be multiple subsections
 illustrate with images (from the videos I made)
 - *slots* offset compared to *tiles*

- less distinct variants
- more control over transitions
- prepare *modules*
- generate grid of *slots*, mark them as *uncollapsed*
- each *slot* can become one of many *modules*
- the *modules* that can be placed in a given slot are its *domain*
- at the start each *slot* has all *modules* in its *domain*
- from the *domain*, compute all possible *boundary conditions*
- for example - there is a *module* in the *domain* with a cliff on its east boundary, so mark cliff to the east as possible
 - mark all *slots* as *dirty*
 - then repeat:
 - propagate constraints
 - for each *dirty slot*, until there are none:
 - mark as *not dirty*
 - find out which *modules* from its *domain* can be placed here and remove the rest from its *domain*
 - decide only by neighbors' (orthogonal and diagonal) *boundary conditions*
 - update *boundary conditions*
 - if *boundary conditions* changed, mark all *uncollapsed* neighbors as *dirty*
 - *collapse* a slot
 - save the current *state* of all *slots* on a stack
 - pick a *slot*
 - pick one *module* from its *domain* at random
 - weighted - provides control
 - remove each other *module* from its *domain*
 - update *boundary conditions*
 - mark *uncollapsed* neighbors as *dirty*
 - if a *slot* ends up with 0 *modules* in its *domain*, backtrack
 - pop a previously saved *state* from the stack and revert to it
 - remove the previously chosen *module* from the *domain* of the previously *collapsed slot*
 - Which slot to collapse?
 - fail fast approach
 - prioritize slot with least options
 - changed to slot with least entropy, because that's more accurate
 - slots near most constraining modules were prioritized, making them more common and leading to repetitive terrain features
 - better to just collapse a random slot
 - in the end still weighted by entropy
 - at first I tried to prefer slots with more entropy
 - define overall structure first by sparsely covering the world, then fill in details
 - often led to deep dead-ends with a lot of backtracking
 - in the end, tiles with less entropy are preferred
 - Limited backtracking depth
 - usually when more backtracking is required, the search would take too long and it's faster to restart the algorithm

3.5 Resources and Obstacles

- after terrain generation, place blockers on tiles
- materials for the player to mine
- just rocks for variety - the player can't build on these
- set up as a few stages
- each stage has:
 - one type 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 Terrain 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
 - unity physics engine X
 - parallel

For the blockers, I didn't want repetitive obstacle models, so they are generated procedurally 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
 - calculate 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

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 throughput budget is exceeded
 - for each attacker subtract the throughput overshoot from buffer
 - fit such that as much of buffer gets used without going over

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)

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

- 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. Tower defense. *Wikipedia* [online]. [N.d.] [visited on 2024-04-05]. Available from: https://en.wikipedia.org/wiki/Tower_defense.
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. Plants vs. Zombies (video game). *Wikipedia* [online]. [N.d.] [visited on 2024-03-24]. Available from: [https://en.wikipedia.org/wiki/Plants_vs._Zombies_\(video_game\)](https://en.wikipedia.org/wiki/Plants_vs._Zombies_(video_game)).
4. BYCER, J. *Game Design Deep Dive: Role Playing Games*. CRC Press, 2023. ISBN 9781000966718.
5. Rogue (video game). *Wikipedia* [online]. [N.d.] [visited on 2024-03-24]. Available from: [https://en.wikipedia.org/wiki/Rogue_\(video_game\)](https://en.wikipedia.org/wiki/Rogue_(video_game)).
6. BYCER, J. *Game Design Deep Dive: Roguelikes*. CRC Press, 2021. ISBN 9781000362046.
7. Slay the Spire. *Wikipedia* [online]. [N.d.] [visited on 2024-03-24]. Available from: https://en.wikipedia.org/wiki/Slay_the_Spire.
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. Desktop Tower Defense. *Wikipedia* [online]. [N.d.] [visited on 2024-04-16]. Available from: https://en.wikipedia.org/wiki/Desktop_Tower_Defense.
11. NINJA KIWI. *Bloons TD 6* [online]. [N.d.]. [visited on 2024-04-16]. Available from: <https://ninkakiwi.com/Games/Mobile/Bloons-TD-6.html>.
12. UNITY TECHNOLOGIES. *Unity Documentation* [online]. [N.d.]. [visited on 2024-04-07]. Available from: <https://docs.unity.com>.
13. 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>.
14. UNITY TECHNOLOGIES. *Unity - Scripting API: Random* [online]. [N.d.]. [visited on 2024-04-07]. Available from: <https://docs.unity3d.com/ScriptReference/Random.html>.
15. 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>.
16. L'ECUYER, Pierre. Tables of Linear Congruential Generators of Different Sizes and Good Lattice Structure. *Mathematics of Computation*. 1999, vol. 68, no. 225.

List of Figures

1.1	A level in <i>Plants vs. Zombies</i>	7
1.2	The map screen in <i>Slay the Spire</i>	9
1.3	A fight in <i>Slay the Spire</i>	9
2.1	<i>Defend</i> , <i>Strike</i> , <i>Iron Wave</i> and <i>Double Tap</i> cards from <i>Slay the Spire</i>	14
2.2	All the plants of <i>Plants vs. Zombies</i> in the in-game almanac.	15
2.3	Card reward screen in <i>Slay the Spire</i>	16
2.4	Seed select screen in a rooftop level in <i>Plants vs. Zombies</i>	17
2.5	A screenshot from <i>Kingdom Rush</i> , featuring a wave timer.	19
2.6	A screenshot from <i>Desktop Tower Defense</i>	20
2.7	The levels <i>Park Path</i> and <i>Another Brick</i> from <i>Bloons TD 6</i> with the attacker paths highlighted.	20
2.8	An example of a valid path in the game world.	21
2.9	Several paths that would be confusing or ambiguous if paths could cross.	22
2.10	Several paths illustrating the rule that each tile can only have one distance to the <i>Hub</i>	22

List of Tables

List of Abbreviations

A Attachments

A.1 First Attachment