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

Tower defense is 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* [1], released in 2009.

In *Plants vs. Zombies*, the player defends their house from zombies. As can be seen in 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 picture, 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.

Roguelike is named after the game *Rogue* [2], 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 *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.

A more recent game that's a good example of this genre is *Slay the Spire* [3], 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 we can see the player character 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 ang 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

The game we intend to make will be single-player. 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.

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.

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.

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.

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. This is important, because some battles will be harder, but provide better rewards.

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 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. However, we will create a functional demo version, which can be used to playtest the core gameplay. The demo must be prepared for future development so that more content can be added later, but it will still contain some base content in order to be playable.

Our main goals will thus be:

1. Design the game's mechanics and features.

2. Implement the systems and mechanics necessary to play through a battle. As described in section 1.3, this includes attackers, towers, abilities, and the economy.

3. Implement a system to procedurally generate the level a battle takes place on.

4. Include tutorial to explain the game's mechanics to the player (might not happen).

5. Run a playtest (might not happen).

The demo version will allow the player to progress through battles and collect blueprints, however, there will be no map view to let them choose their path. 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.

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 demo version of our game won't achieve any of them well, but the goals still inform the design of the mechanics we will implement. 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 tens of 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 come across an 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 cards *Defend* and *Strike* in their deck. Compared to them, *Iron Wave* is a very cost-efficient card. We can see 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 you *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 your 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

The player should be able to beat the game with a lot of different combinations of blueprints. We want the player to experiment with different builds, but that

will never happen if only few of them are good enough.

- StS — some cards interact in interesting ways that make them stronger ; there are cards and relics which fundamentally change the way your deck works (ex. barricade and entrench) nerfing some builds is important

Figure 2.2 A small portion of the many interesting cards in Slay the Spire, viewed in the in-game compendium.

- PvZ — the interactions are not so strong, but you have to combine the plants such that you have no weak spots — cheap and expensive, for specific zombie types

Figure 2.3 All the plants of Plants vs. Zombies in the in-game almanac.

- make blueprints that have unique effects that change the way you play

2.1.4 Force Exploration

The player will have to use a different build every run. 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, we need to force the player to explore and make them put effort into learning other strategies.

- StS — you have to explore different strategies because you get different cards every time, some enemies ensure you are prepared for everything and intentionally break some strategies

- characters

Figure 2.4 Card reward screen in Slay the Spire. Here the player can choose on of three randomly selected cards to add to their deck.

- PvZ — you have to explore different strategies because you have to adapt to different zombies and level environments, not too deep; after the campaign, three seed slots are selected for you.

- choose from random blueprints, various attackers with abilities and terrain types

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.

- StS — not easy to beat; after you beat the game, you can play on a higher ascension

- we will have something similar to StS

2.2 Battle

- build stuff, send waves, use abilities

Figure 2.5 Seed select screen in a rooftop level. Here, plants must be planted in flower pots, and most plants cannot shoot uphill.

2.3 Attacker Movement

- free movement with individual pathfinding X

- too unpredictable for the player — The player needs to plan in advance in order to maximize the possibility to take calculated risks - solution: visualize enemy paths — Too many different paths - too much clutter; Still planning at most one wave in advance

- linear lanes (like in Plants vs. Zombies) X - not enough space for interesting building placement

- paths based on your buildings X - what if they block off the path?

- predefined paths ✓

- branching

- other qualities

2.4 World

- free placement X

- grid of tiles ✓- more granularity, easier to develop intuition, same for attacker paths

- hexagons X

- squares ✓

- 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

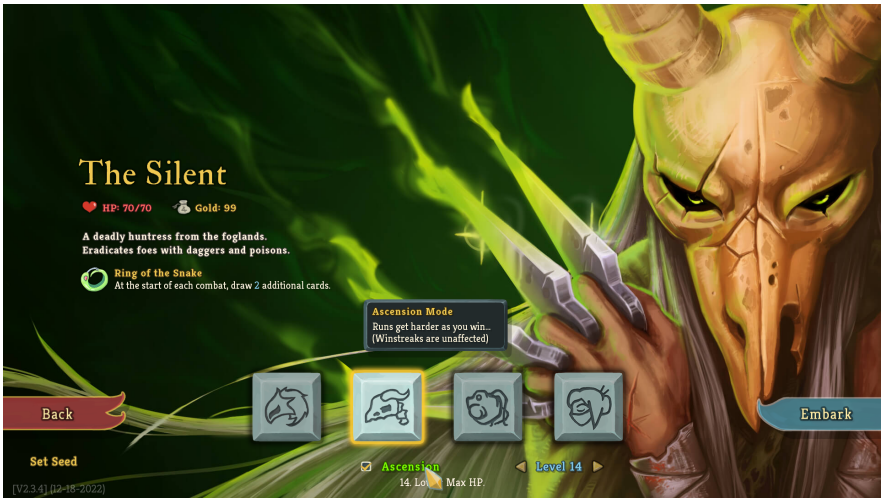

Figure 2.6 Character select screen in Slay the Spire. Here the player can also choose their ascension level, making the game harder.

2.5 Resources

for each subsection: what is it, why is it, how to acquire it, what are the constraints

2.5.1 Materials

2.5.2 Energy

2.5.3 Hull

2.6 Graphical User Interface

specify controls

2.6.1 Waves Left and Fuel

2.6.2 Hull

2.6.3 Wave Preview

2.6.4 Materials and Energy

2.6.5 Blueprints

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

2.6.6 Info Panel and Selection

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

2.6.7 Highlights and Range Visualization

2.7 Attackers

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

2.8 Buildings

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

2.8.1 Economic buildings

-provide resources

2.8.2 Towers

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

2.9 Abilities

- used mid-wave
- usually instant effects
- free placement, global placement, tile placement, use on a building

2.10 Camera controls

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

2.11 Future Features

- run structure
- map
- events and shops
- difficulty levels
- unlocks

3 Analysis

3.1 Game Engine

- unity because of familiarity
- mention the features we will use

3.2 Procedural Level Generation

- make each level unique
- one random seed dictates whole run
- in background thread
- debug visualizations (images)

3.3 Random Number Generation

- unity random X
- system random X
- custom random ✓
- why LCG

3.4 Path Generation

- how to make paths with the required qualities?

3.4.1 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

3.4.2 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

3.4.3 Path Visualization

- line renderer ✓
- why does it look this way

3.5 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 prevoiusly 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

3.6 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 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 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

3.7 Terrain Types

- what information is tied to the type
- why txt (inspector was not as legible)

3.8 World Builder

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

3.9 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

3.10 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

3.11 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

3.12 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)

3.13 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

3.14 Game Commnds

- we want various components to modify how other components function

- examples
- also react to events as a bonus

3.15 Blueprints

- why are they implemented this way

3.15.1 Attacker Stats

- blueprints for attackers

3.15.2 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. Plants vs. Zombies (video game). *Wikipedia* [online]. [N.d.] [visited on 2023-03-24]. Available from: `https://en.wikipedia.org/wiki/Plants_vs._Zombies_(video_game)`.
2. Rogue (video game). *Wikipedia* [online]. [N.d.] [visited on 2023-03-24]. Available from: `https://en.wikipedia.org/wiki/Rogue_(video_game)`.
3. Slay the Spire. *Wikipedia* [online]. [N.d.] [visited on 2023-03-24]. Available from: `https://en.wikipedia.org/wiki/Slay_the_Spire`.

List of Figures

List of Tables

List of Abbreviations

A Attachments

A.1 First Attachment