Computer Science & Electronic Engineering, University of Essex

Making a Tile-based RPG Engine

Capstone Project Final Report

Supervisor:
Richard Bartle

Second Assessor:
Mike Sanderson

Darby, Joseph R P J
Reg. No.: 2104349

Acknowledgements

I would like to thank my supervisor, Richard Bartle, for the aid he has given me throughout the duration of the Capstone Project. Both banter and serious discussions were had during our meetings, and dropping titbits of knowledge really grew my intrigue and respect for the games industry.

I would also like to express commiserations for the countless peers I bored with my numerous rants, frustrated at bugs and limitations of my system. It happened a lot, so I can only thank you all for sticking with me throughout it – whether voluntarily or not.

Abstract

Two-dimensional rendered tile-based games have existed since the 1970s, mainly due to the ease of drawing textures to a screen, and the ease of creating and storing said textures and tile-maps so that they don't take up much storage space. This project concerns the creation of a two-dimensional, top-down, tile-based game, with a focus on making a simple-to-modify game engine for a user to add their own levels and enemies.

The game and engine that has been developed is centred around a *Soulsborne*-esque style of gameplay. Soulsborne, or more accurately, *Soulslike* games, are often defined by the action role-playing game (RPG) gameplay, usually with a dark fantasy setting. Specifics of the gameplay include managing stamina, telegraphed enemy attacks, big bosses, and dodging through attacks. This makes for a fair, yet difficult game, that few gamers today would be ignorant to the existence of.

The engine was programmed using C#, with the aid of the MonoGame framework. MonoGame is a framework based on Microsoft XNA, which adds many useful features to code a game, including a .NET environment to run it. However, despite having some high-level functionality, MonoGame remains quite a lower-level framework, giving mostly bare necessities to the user to create their own higher-level game engine. Arduous tasks such as loading images and other forms of in-game content are managed by the Content Pipeline that MonoGame supplies, and there are basic drawing functions for said images, strings, and primitives (though there exists limitations with this).

Due to the limitations, this project also features implemented various low-level engine concepts such as rasterisation for shape drawing and Separating Axis Theorem (SAT) for collision.

Contents

List of Symbols

- **RPG** – Role-playing Game.
- **Soulsborne** – referring specifically to *FromSoftware*'s games. Namely: *Demon's Souls*, *Dark Souls* (the full trilogy), *Bloodborne*, and *Elden Ring*.
- **Soulslike** – games/gameplay similar to that of *Soulsborne* games.
- **I-frames** – invincibility frames, frames of collision calculations in the game where the player is given invulnerability and cannot take damage.
- **Combo** – consecutive attacks made in a specific sequence.
- **TTRPG** – Tabletop RPG/Tabletop Role-playing Game.
- **PC** – Player Character, the avatar in-game that the player controls.
- **NPC** – Non-player Character, mainly refers to friendly/non-hostile entities that the player doesn't control.
- **Stats** – statistics or attributes of a game entity.
- **IDE** – Integrated Development Environment, a code editor that helps programmers do their work efficiently via the use of tools and often debugging capabilities.
- **L**. – left, when referring to a mouse button or keyboard key (such as "Left Shift").
- **R**. – right, same as above.
- **px** – pixels.
- **SAT** – Separating Axis Theorem.

Ludography

- *Corpse Party: Blood Covered* – Team GrisGris, 2016 (worldwide release)
- *Pokémon* – (numerous versions) Game Freak, 1996 onwards
- *Dark Souls* – FromSoftware Inc., 2011
- *Dark Souls III* – FromSoftware Inc., 2016
- *Elden Ring* – FromSoftware Inc., 2022
- *Sekiro: Shadows Die Twice* – FromSoftware Inc., 2019
- *Hollow Knight* – Team Cherry, 2017
- *Tunic* - Isometricorp Games, 2022
- *Dungeons & Dragons* – Gary Gygax & Dave Arneson, 1974
- *Momodora: Reverie Under the Moonlight* – rdein, 2016
- *Little Witch Nobeta* – Pupuya Games, Simon Creative, 2022

1 Background

1.1 The Use of Tiles in Games

1.1.1 What Are Tiles?

Tile-based games make up some of the earliest types of games on home consoles. This is for good reason, too. *Tile-maps* were made to combat the expensive – in multiple ways – screen graphics model named *framebuffer*. This competition is discussed in *Before the Crash: Early Video Game History* **[1]**, and is outlined here:

The framebuffer model worked by mirroring the way computer screens worked; a screen is made up of pixels, and each of those pixels can be manipulated. This is flexible in terms of being able to draw exactly what is desired, but is also computationally expensive, considering that even a standard 1920x1080 screen would need to scan 2,073,600 individual pixels and draw them all. It is also memory intensive, as for detailed and varied game backgrounds, many different 2,073,600-pixel illustrations would need to be stored on the storage medium (that was quite short on bytes at the time). Although monitors of this size weren't used for home consoles back then, the disadvantages are still clear.

This is where the tile-map model started becoming more popular. The tile-map model works by segmenting the screen into a grid of equally sized squares, and using a large image of multiple square textures the same size of the grid squares called a *tile-set*. The level is then read in as input, as an array of indices, each mapping to a texture on the tile-set. This significantly reduces the computation needed to draw to the screen, and allows repeated textures, like the walls of a castle, to be stored as a single tile and repeated only in the level input array as an index, additionally leading to significantly smaller storage space used.

1.1.2 Adoption of Tiles

Since the initial adoption of tile-maps in 1979 with the game *Galaxian*, there has been a plethora of tile-based games being released. The widespread use of *RPG Maker* (as seen by it having over 2 million copies sold worldwide by August 2005**[2]**, in addition to the several cornerstone games released, such as *Ib*, *Corpse Party*, and *Omori*), and subsequent releases under a similar name, for indie game developers in the early 2010s is one of many examples of the fact that tile-based games are here to stay. There may not be much of an argument of computation power anymore, but the simple and low volume storage of levels as indices makes game creation by solo and indie developers a much more appealing and easy-to-get-into task.

The perceived issue by the author of this report, however, is that game engines such as *RPG Maker* don't give much freedom in gameplay type. They almost all use a system where players and other mobile entities can only move tile-by-tile, and that fights aren't real-time. Instead, *encounters* occur when the player and an enemy step onto the same tile, creating a *battlefield* instance where turn-based attacks follow. One of the reasons for this being the leading method of fighting/battling in top-down tile-based games may be the fact that early Japanese games greatly popularised this style of enemy encounter as well as focussing heavily on the RPG genre, such as the widely popular *Pokémon* and *Final Fantasy* franchises.

1.2 Defining 'Soulslike'

Soulslike is a genre of video game based off the gameplay elements of *FromSoftware*'s collection of games fitting under the *Soulsborne* umbrella. This collection includes and is limited to: *Demon's Souls* (both 2009 version and the 2020 remake), *Dark Souls* (2011), *Dark Souls II* (2014), *Bloodborne* (2015), *Dark Souls III* (2016), and *Elden Ring* (2022). Some regard *Sekiro: Shadows Die Twice* (2019) as part of this umbrella term, but for the sake of exploring the term *Soulslike*, it won't be included. The main reasons being the lack of a stamina system, and lack of emphasis on exploration compared to the other titles.

Soulsborne games have certainly made their mark in the history of video games, with the newest entry, *Elden Ring*, having 23 million copies sold as of February 2024**[3]**. This can also be seen on Steam, as games tagged as 'Souls-like' make up a total of 1.9% of **all** games released on the platform**[4]**. Considering that 14,402 games were released on Steam in 2023**[5]**, that's around 275 new Soulslike releases last year alone (see **Appendix 1**). The sheer quantity of games it easier to find the common elements between these so-called Soulslikes:

1. **Real-time Combat** – Soulslikes are often praised for their hard, unapologetic difficulty. This comes from a variety of factors, but all Soulslikes being action RPGs subsequently leads to them all having real-time combat, where timing and awareness of surroundings/other enemies is crucial to progress and win.
2. **Stamina** – One of the many micromanagements that makes the genre difficult is making the player keep an eye on their stamina meter. Stamina is usually displayed via a bar underneath the health bar, and dictates how many actions the player can take. For example, stamina reduces by a set amount every time the player attacks, and once the stamina has reached 0, the player is unable to attack again until it begins to regenerate naturally after a short period of not performing any stamina-using actions.
3. **Dodge-rolling** – Arguably the cornerstone gameplay mechanic of the Souls titles, rolling through enemy attacks grants invincibility against damage for the period of the dodge. This makes dodge timing an important skill to learn when playing Soulslikes. Some Soulslikes, however, don't grant i-frames. They may still have a dash mechanic to quickly move out of the way of an attack, like that in *Immortal Planet*, but players must be more aware of enemy attacks to properly dodge without invincibility. Some games take it a step further, like *Hollow Knight*, where a dash is gained as a new ability partway through the game, and upgrading said dash to give i-frames only occurs in the late-game.
4. **Telegraphed Enemy Attacks** – Enemies in Soulslike games, even if attacking in randomised combos, all have fair, telegraphed attacks – either or both via windup animations and audio cues. For example, in *Tunic*, the spider enemies will lurch backwards before performing their pouncing attack, and ghosts will wail before their long slash attack (which is the sole way to tell if they're going to attack, as they only attack whilst invisible). As for the Soulsborne games, this fairness of telegraphing attacks is emphasised further with attacks being designed to follow an almost rhythmical scheme. In simple music theory terms, and using the example of the beginning of the spinning attack by the Abyss Watchers (*Dark Souls III*) enemy: in a *bar* with 8 equally spaced *beats* (eighth-notes), the animation for beat 1 and 2 is the Watcher readying his sword, beat 3 is a thrust attack, beat 4 is a rest, beat 5 and 6 are both swinging attacks, and beats 7 and 8 are readying up for the following

spinning attack in the next bar. This shows that fairness is achieved in telegraphing by having the attacks/attack frames fall onto steady rhythmical subdivisions of a bar.

5. **Levelling Up** – Like the popular TTRPG *Dungeons & Dragons*, each player, most enemies, and all NPCs have certain numbers, connected to what's referred to as *ability scores* in D&D, which dictate the rest of their stats. In *Elden Ring*, Vigor controls health, and levelling up that score increases the health for the player. Most, if not all, Soulslikes have some way to level up/increase scores/stats. Soulsborne titles opt for the 'level-up girl' route, where a **resource** dropped by felled enemies can be exchanged for incrementing a score by an **NPC** (not a vendor, however) in whatever **location**(s) they are placed. For *Dark Souls III*, players exchange **souls** for levels via **Fire Keeper** at **Firelink Shrine**, and for *Elden Ring*, **runes**, via **Melina**, at **Sites of Grace**. Another method of levelling up, however, is seen in *Momodora: Reverie Under the Moonlight*, where health upgrades are found around the game world as secrets or rewards for completing a challenge. *Tunic* combines the two approaches, and has the player find Offering Items around the world, in which they must traverse back to a Check-Point to trade the item, with the addition of money (dropped by enemies and looted from chests), to increase the stat coupled with the item (the Fang increases ATT (attack), the Flower increases HP (health points), for instance).
6. **Emphasis on Exploration** – *Dark Souls* was the poster child for this particular aspect. It blended a three-dimensional world with metroidvania-style progression (that is, until late-game when it's speculated that production was rushed considerably) which was previously unseen in three-dimensional games. Metroidvania, a portmanteau of the games *Metroid* and *Castlevania*, is a subgenre of action/adventure games that focusses on non-linear gameplay routes, rewards exploration with new items and abilities, and uses said abilities to progress to new areas. Although Soulsborne games rewards exploration less than combat, as abilities are scarcely rewarded via exploration, and instead when defeating a sequence of bosses. Later *Souls* titles guide players much more linearly through the game, a grand return was made with *Elden Ring* being completely open-world. The impact of the first of the Souls games – *Dark Souls* – being a metroidvania, however made a big impact on Soulslike releases from there on. On Steam, a huge 20.4% of all games tagged as "Souls-like" are also tagged as "Metroidvania" (see **Appendix 2**), only accentuating the fact that Soulslikes place an apparent emphasis on exploration.

With the knowledge of what these common elements are, both from Soulsborne titles, and games hailing from them (that are referred to as 'Soulslike' on Steam), preliminary design decisions have been made about the game and engine.

1.3 Subsequent, Preliminary Design

When conceiving what to feature in the game engine, there must be consideration of what exactly can be added. Since its creation was based off merging both a tile-based model/system and Soulslike gameplay, compromises had to be made. The simplicity of tile-based collision isn't compatible with real-time combat, or rather, they aren't desirable to combine. When moving around a truly tile-based world, the player can move in eight directions: up, right, down, left, and in between. This is undesirable as the real-time combat/action part of the design would be hindered, as difficulty and fairness in the combat design comes from the micro-adjustments made in movement and turning/facing specific directions. It's moments like dodging an attack by only a couple of pixels that really make exhilarating and satisfying gameplay in the opinion of the author. For this reason, the game engine doesn't use simple tile-based collision, but 'real distance' collision.

Making a game engine specifically to place an emphasis on exploration doesn't make much sense; it's more of a design decision for the user/developer to make. For this reason, the only design decision to emphasise exploration was making the creation of new levels/rooms to be an easy and simple task. This was done by having a parser read text files whenever a new level is loaded, of which the syntax of the files is straightforward. The tile-based part of this was the fact that only rows of comma-separated numbers are needed to make the map (with the addition of a metadata section explaining which tiles correspond to walls/are impassible).

Levelling up was a fairly simple concept to use; exchange some sort of resource for increased stats. For this project, however, the author desired a specific flair for the small example game produced. This flair/gimmick was the use of hit points as a resource. Health, in the context of the game, named *vitality*, is gained when defeating enemies, and can be used to increase the amount of stamina the player has. Vendor trading using vitality was also planned, but the idea was discarded due to time constraints and over-scoping. Upgrading weapons, and finding different weapons and armour around the world was discarded also, but for other reasons. The author took inspiration from *Little Witch Nobeta* and wanted the upgrading of weapons to be attached to the character, rather than an external weapon that can be swapped out. This reduces complexity and confusion around different builds, but unfortunately also limits the creativity of the player for the build they wish to go for.

With dodge rolls being a genre standard, a new, but not unseen, direction was taken. Similar to *Bloodborne*, a dash was added to the player's moveset (only aesthetically different to a roll), but the functionality of *Hollow Knight*'s Mothwing Cloak was used. Found in early-to mid-game, the cloak, as described previously, though granting quick movement to move out of the way of an attack, doesn't inherently grant i-frames. The dash in the game similarly doesn't allow the player to dash *through* enemy attacks, but instead, rapidly exit the area of the attack. This was done because, as a top-down, two-dimensional game, it would have possibly made dodging with the use of i-frames too easy. Now, more positional awareness is required of the player, instead of purely reactional timing, making the game more challenging and satisfying to defeat enemies as a result.

All attacks are telegraphed overtly. As sound wasn't implemented, audio cues weren't able to be used. Instead, more effort was directed into telegraphing the attacks – not through character animation, but an area of effect lighting up in red. First, the windup places the attack polygon in the direction the assailant is facing, and until halfway through the windup, the

polygon grows in opacity, visually, and clearly, signalling to the player that the attack is winding up. Once the windup reaches the halfway point, though, the opacity is locked at 50%, with a black outline being drawn now (the attack area polygons needed more complex drawing methods than what was provided by MonoGame and will be discussed later). This is to aid the player in "feeling the attack rhythm," as discussed earlier. If the player sees the windup from the start, and halfway through it changes it's visuals, then it would be easier for the player to place the 'next beat' and dodge away accordingly, sensing exactly when the attack will come. This is done for the reason of fairness. It should never be the game's fault that the player fails, it should be the player's own mistake that drives them to failure. That way, it's easier to see how to get better, and avoids the frustration against the system. At least, that is what the author believes, hence why they designed the game that way. This is also why there is no randomness in the game. For example, even though enemy attack times *seem* somewhat random, they are actually deterministic, and depend on how far away the player is from the enemy at the moment the attack timer is reset.

Stamina was easy to implement and was an obvious choice to include in the design. It increases the difficulty but keeps the game interesting by having the player micro-manage the stamina meter. It also discourages reckless play, and promotes patience and learning attack windows, which are both features of Soulslikes. For the design, there's a cooldown time for when it begins to regenerate, and the cooldown time is always reset to 0% when a stamina-draining action is performed, such as attacking or dodging. When the player doesn't have enough stamina to perform an action, there simply won't be any action performed. This was one of the ways that players were challenged with requiring calculated inputs instead of frantically performing actions.

The other way that players were challenged with this was via an input buffer. Input buffers are common in the three-dimensional Soulslikes, with it being notoriously long in *Elden Ring*, but are rarely seen, if at all, in the two-dimensional games. The input buffer reads inputs and puts them onto a queue, so that after the current action has completed, the next one in the buffer immediately starts. This is both a blessing and a curse for the player. It means that if an attacking is coming, and the player knows this, but is in the middle of an attack animation themselves, they can press the dodge button before their attack is over to immediately dodge once it is over, giving them a better chance at not being hit. However, this is also a curse, as pressing the wrong input at the wrong time can open the player up for failure. For example, repeatedly pressing the attack button during a short attack window can lead to the player being stuck in attack animations, unable to dodge, whilst the enemy begins to wind up a counterattack (as the attack window against the enemy was shorter than the player assumed). This means that the player must be more vigilant and patient, giving only calculated and thought-out inputs.

2 Tools

2.1 Language

C# was used for this project. It was chosen for a few reasons, one of those being familiarity. The author is most familiar with two high-level languages, C#, and Java, both being quite syntactically similar. The reason Java wasn't used was due to the author's personal, strong distaste for the language, and the fact that it doesn't support type reification. Instead, it uses type erasure, making the use of generics (which is important when it comes to game objects) severely limited, requiring a lot of unnecessary bookkeeping to achieve the same effect.

A high-level language was used in place of a lower-level language such as C++ since lower-level languages often need more work to get the same results. This, coupled with having more to think about and do (such as free heap memory and dispose of unused pointers, et cetera) would make this task of creating a game engine much more tedious and long-winded than necessary (and possibly more bug-prone, meaning more time spent fixing bugs).

C# is object-oriented, and using an object-oriented language for this project was important too. Object-oriented programming is powerful in that it hinges on the use of inheritance and interfaces. These factors are very significant for game creation, as games often use inheritance to create new classes that are children of an base 'Entity'/'GameObject' type. This allows all entities to be grouped together in the same list, and have common features among them all, such as a draw or move function, and shared member variables such as a position or health.

The main reason that C# was chosen as the language for the project, however, is because it has a free, open-source framework called MonoGame.

2.2 Framework

MonoGame is a framework built on C#'s .NET framework. It is open-source and free, making it not need much commitment to begin using if it was paid. Additionally, it has natural integration for the popular and native C# IDE, Visual Studio 2022, it being easy to install as an extension and get a project up and running and ready to play quickly.

Another popular (albeit old) framework for making games/a game engine exists for C#, which is Microsoft XNA. It's very similar to MonoGame, but MonoGame is both more modern, and has more tools available to help the developer. In fact, MonoGame actually uses XNA as a base, so for a developer experienced in XNA, MonoGame is easy to adopt.

MonoGame has a lot of functionality packaged with it that helps the creation of games, however, it is a fairly low-level game engine. The fact that it allows the creation of a higher-level game engine from it is why it was chosen, with the addition of the high control the developer has over what exactly the engine does, due to the lower-level nature of the framework. With a high-level game engine like Unity or Unreal, a lot of the functionality, like saving maps/scenes, and rendering, is taken care of, even if it may be less optimised for what the developer wishes to use it for. MonoGame instead allows developers to program their own methods of achieving this functionality. Where this was especially the case, was with having level data be loaded in from a small, simple text file, which was intentionally easy to

edit and read. This wouldn’t be the case with a .unity save file, as it contains much more data than necessary for a tile-based game level.

Another big reason as to why MonoGame was used, is once again because the author is familiar with it. During college, it was briefly used for coursework, however the author wasn’t completely familiar with object-oriented programming yet, nor the tools that MonoGame offers. For the project, however, considering the scope of the project too, going with a framework which was familiar was an easy decision to make. Especially knowing that starting a game engine from complete scratch would’ve taken too much time and required significantly more knowledge of low-level systems.

3 Technical Documentation

3.1 Controls

- **W** or **Up Arrow** – move northerly.
- **A** or **Left Arrow** – move westerly.
- **S** or **Down Arrow** – move southerly.
- **D** or **Right Arrow** – move easterly.
- **L. Shift** or **R. Mouse Button** – dash.
- **Mouse Cursor** – look around.
- **L. Mouse Button** – attack.
- **Middle Mouse Button** – lock-on.
- **L** – level up stamina.
- **Escape** – quit game.

3.2 Controls Expanded & Gameplay Mechanics

Moving around is done via the “WASD” keys or the arrow keys. Keys can be combined, with “W + A” (“Up Arrow + Left Arrow”) moving the player northwesterly, with similar combinations for other tangential directions. “W + S” and “A + D” both cancel each other out, as movement works on both a horizontal and vertical -1 to 1 input – -1 + 1 being 0. The player can also make the PC dash by pressing “L. Shift” or “R. Mouse Button”. This will lock the direction and increase the movement speed of the PC for the duration of the dash. If no movement keys are being pressed, the PC dashes in the direction it was looking in.

Looking around is done via the “Mouse Cursor”. The PC rotates around the centre of its position, looking at wherever the player places the cursor on the screen. The player can additionally ‘lock on’ to an enemy by pressing the “Middle Mouse Button” close to one, highlighting the enemy by placing a white ring around it, larger than its size. This will force the PC to always look in the direction of the enemy, making landing attacks to be easier.

The player can attack using “L. Mouse Button”, which will draw a red polygonal arc outward from the centre of the PC, in the direction it’s looking. Attacks have three phases, with the first being split into two. The first phase is the windup phase. This is where the polygon is drawn transparent, slowly increasing in opacity until halfway through the windup timer (where it will also be at 50% opacity). For the second half of the windup, the PC is locked in place and rotation, and a black outline with be drawn onto the arc. The second phase is the linger phase. This is where attack polygon checks for collision and remains this way for the length of the ‘linger’, now being drawn fully opaque (still with the outline). The final stage is the cooldown. The polygon is no longer drawn to the screen, and collision against it isn’t tested. Instead, the player cannot attack for the duration of the cooldown. There is a small grey bar underneath the PC, that begins with a length of 0, and grows at a speed proportional to the cooldown timer until it reaches the full length of the PC’s sprite. Once it reaches that full length, it will disappear, indicating to the player that they can once again attack.

Attacking and dashing have two traits in common:

1. They both deplete stamina.
2. They are both *input buffer* actions.

Stamina is shown the by the fraction in the top left corner, and the green bar that's underneath it. Attacking depletes 7 stamina, and dashing depletes 5. Not having enough stamina to perform an action means that the player simply can't perform that action. This is bad if the PC is about to collide with (be hit by) an enemy's attack arc, with no way to quickly move out of the range using a dash. Stamina has a cooldown timer, which waits 1.5s after the last stamina depleting action before it starts to regenerate points again at a rate of 2 stamina/s. Stamina can also be levelled up, which raises the both the maximum points (denominator) and the current points (numerator) of stamina by 5. This, however, comes at a cost of 10 vitality (explained later), and can be repeated until death.

The 'input buffer' is a way to buffer certain actions performed (or rather, to be performed) by the player. The size of the buffer is 3, meaning that dashes and attacks (the only actions that can be buffered) can be queued up to three times (not mutually exclusively), the next action in the buffer being performed immediately after the current one ends. Note that an attack 'ends' after the linger phase, meaning that if a dash is buffered, it can be performed right after the linger ends, but if an attack is buffered, it must wait for the cooldown first. The difference between the input buffer and a normal queue, however, is that any fourth input overwrites the last in the buffer. For example, if, in the order they will be dequeued from the buffer (leftmost being first to be performed), the buffer contains "attack, attack, dash", and the player inputs an attack, the buffer overwrites the last (rightmost) action and the buffer now contains "attack, attack, attack". This is done to discourage the player from rapidly panic-inputting a dash, or recklessly pressing the attack button many times excitedly during an open window against an enemy, as the enemy may begin to attack again whilst the PC is stuck due to the position lock that occurs during an attack.

Vitality is the number in white, in front of the red circle at the bottom left of the screen. Enemies also possess vitality, of which they have a small red bar beneath them displaying what proportion of their maximum they have left, based off how far along the full length of the enemy sprite the red bars width is. Once the enemy/PC loses all its vitality (through attacks from the player, or friendly fire from other enemies) it dies and disappears, 'leaking' its vitality to the agent that dealt the killing blow, who absorbs it. Leaking and absorbing vitality is done by taking the max vitality value of the felled enemy, multiplying it by the leak percentage (70% by default), and essentially 'giving' it to the killer. This increases (by the amount leaked) both the max vitality and the current vitality of the killer, if it was an enemy, or just increases the current vitality of the PC (since it has no concept of 'maximum vitality').

Finally, the player can press "Escape" to end the game. This closes the window and exits the program.

3.3 Example Game

To run the example game, the release build must be selected in Visual Studio 2022, and "Ctrl + F5" pressed (alternatively, click on the hollow green arrow, labelled "Start Without Debugging", with the cursor). This will start the example game from "testmap1.txt".

What is loaded is the tile-map defined in "testmap1.txt", with the enemy placements, player spawn position, walls, and exits. The PC is a green circle with a smiley face, and the enemies are (by default) red circles with an angry face. The direction the player is/enemies are looking in is whichever direction looking at the face head-on would be considered 'down'

(basically, drawing a line beginning between the middle of the eyes and going through the centre of the mouth onwards).

In this level, the water tiles, and the stone tiles are the walls, and attempting to move through them will not do anything. Moving outside of a deviation range from the centre of the screen also make the camera move with the player. This sort of behaviour is discussed in *Scroll Back: The Theory and Practice of Cameras in Side-Scrollers***[6]**, but is alternatively implemented in this game engine. Two integers are stored in camera, one for horizontally permitted deviation, the other for vertically. Permitted deviation is just how far the PC position is allowed to move from the centre of the camera screen until it begins to follow. This allows more of the level to be seen at once, but also does away with the boring standard method of always having the PC in the dead centre. It could also possibly make the playing experience better because moving around to dodge enemy attacks doesn't constantly move the camera around, potentially disorienting the player.

An enemy is present, patrolling around the centre of the mess of path tiles (used to demonstrate how the tile-set works), who, once the PC is within it's 'aggro range', will begin to pursue and follow the PC. It has 20 vitality, and once the enemy reaches a distance from the PC that is equal to half its attack range, it will begin to *hover*. In this state, it attempts to remain at such a distance, has its speed halved, and its attack timer will begin. Enemies will only attempt to attack when in this hover state, supported by the fact that the attack timer will only be incremented during this state. The timer does keep its time value, however, so if the enemy enters the hover state again, instead of starting over from zero, the attack timer will simply resume.

There is also an enemy guarding the only exit from the level, on the southwest side of the map. This displays the other enemy AI state, completing the trio – guard, patrol, pursuit. This enemy has more vitality (actually, the default amount) at 30. Once again, if the PC enters the aggro range of the enemy, it will begin its pursuit. However, if an enemy loses sight of the PC (there are walls in the way), then it will make its way to the last position it saw it at, and if then it still cannot find the PC, it will path-find back to its guard position/patrol point.

Once past the guard enemy, the light emanating from the east side of the path tile shows where the exit of the level is. Moving into it reads "testmap2.txt" and loads all the details contained. This level is meant to replicate the scenery of a mountain valley/range and uses textures taken from a community member-made 8x8 Minecraft texture pack**[7]**. In this level, there are 6 exits, though only half are actual exits, while the other half function as in-level teleporters. There are many more enemies in this level, and two stones near the centre, with an enemy patrolling around the larger of the two. All the actual exits (the exit the player entered from, the eastmost exit, and the southmost exit) take the player back the first level, all in different locations (eastmost put the player back at their initial spawn point). The exits that act as teleporters (the two on each end of the waterfall in the southeast, and the one through the hidden hallway after moving through some ever-so-slightly differently-textured mountain rocks) lead to different places in the same level (the hallway one taking the player to the default spawn point of this level, and the waterfall ones teleporting the player between the two), and don't cause the parser to reread the same map and load it all again.

There are two alternate enemies in this level, and they are down a secret hallway to the west-southwest of the level. One is burgundy instead of red, while the other is dark blue. They have alternate stats/attributes which makes them more difficult to defeat, as one has a longer

attack range but slower windup (still quicker than the default red enemy), the other has a fast windup but shorter attack range (still longer than the default red enemy), and both have significantly faster rotational and movement speeds.

Dying (reaching 0 vitality) causes the player to disappear, leaving only the "Escape" button available to press to end the game.

3.4 Level File Syntax & Section Table

This project is about making a tile-based RPG *engine*, so it wouldn't be much use if a user couldn't make their own game using it. The engine is designed to make creation easy and simple to do, with only a text file needed to create whole new levels. It has a basic syntax, with very few mandatory fields, as there are default values for most things. The syntax follows:

- Sections are defined with a '`[Header]`' (in square braces) to begin them.
- Attributes are altered using '`field = value`'(s) listed underneath the header, and separated using line breaks.
- `value` can be either a
    - `string` (no quotation marks),
    - `integer`,
    - `float` (number with a decimal point),
    - `integer array` (integers comma separated, whole array surrounded in curly braces),
    - `Point` (an integer array of exactly two integers),
    - `Point array` (comma-separated integer arrays (each having exactly two integers) in an array (surrounded by curly braces)).
    - Note: `[Tilemap]` breaks the convention slightly.
- A section ends when there is an empty line (after a line break).
- The only mandatory sections are `[Tileset]`, `[MapMetaData]`, and `[Tilemap]`, **in that exact order**.
- Optional sections of `[Exit]` and `[Enemy]` can be repeated as many times as exits/enemies desired.
- Whitespace (tabs and spaces) are trimmed.

A table for Sections, their Fields, Value types, and an explanation of the value:

* means required, superscripted numbers correspond to the Note X of that number below.

Section Name	Field Name	Value type	Explanation
*** Tileset**	* Name	string	Name of tile-set to use.
	* Columns	integer	Number of columns in tile-set[1].
	* Rows	integer	Number of rows in tile-set[1].
	* TileSize	integer	Size of tiles in tile-set[1] (px).
*** MapMetaData**	* Width	integer	Width, in tiles, of tile-map.
	* Height	integer	Height, in tiles, of tile-map.

	* TileSize	integer	Size of tiles post-rendering (px).
	* WallTiles	integer array	Tiles on the tile-set[2] that are walls (impassible).
	* Spawn	Point	Default spawn point tile for the level.
*** Tilemap**	N/A	list[3] of integer arrays	Each integer corresponds to tiles on the tile-set[2].
Exit	* Tile	Point	Exit placement on tile-map.
	* Level	string	Level to load and enter[4].
	Destination	Point	Spawn point in new level[5].
Enemy	* Position	Point	Place of enemy spawn[6].
	* PatrolPoints	Point array	Nodes in patrol path[6].
	Sprite	string	Name of sprite to use.
	Speed	float	Movement speed.
	Vitality	integer	Starting vitality.
	Damage	integer	Damage that attacks do.
	AttackWindup	float	Length of windup (secs).
	AttackLinger	float	Length of linger (secs).
	AttackCooldown	float	Length of cooldown (secs).
	AttackRange	float	Range of attack (px).
	PatrolStartIndex	integer	If Position not specified, which PatrolPoint Point to spawn at.
	AIType	string	Guard/Patrol.
	PatrolType	string	Boomerang/Circular[7].
	AggroRange	integer	Distance for enemy to "see" and begin pursuing the PC (px).
	RotationSpeed	float	Speed enemy turns at (rad/s).

Note 1: Columns & Rows can be defined (must be both), or TileSize can be defined. If both all are defined, Columns & Rows will be used.

Note 2: Tiles on the tile-set are referenced via their 1D-array index.

Note 3: Tilemap requires `[MapMetaData].Height` number of integer arrays, each of length `[MapMetaData].Width`. The arrays shouldn't be surrounded by braces.

Note 4: The level specified can be the same as the current level the player is in.

Note 5: If not defined, the new level will be entered with the player at its default spawn point.

Note 6: Both Position and PatrolPoints can be defined, but only one of them is required.

Note 7: Boomerang – reverses travel direction through PatrolPoints once an end is reached, Circular – once the final PatrolPoint is reached, the first is then the next Point.

4 Data Structures & Algorithms

4.1 Drawing Shapes

MonoGame has native sprite batching and can draw loaded textures onto the screen. This is limited, however, as drawing anything custom that isn't loaded in as a texture can be difficult to do. The only custom drawing that can be done is of axis-aligned rectangles. Rotation works, but not on the rectangles themselves. Instead, the rotations and transformations are done to the sprite at draw-time, and doesn't reflect the state of the original rectangle, even if that was desired. For this reason, was custom drawing developed for the engine. DCircle, DRectangle, and DPolygon (D standing for drawable) were added, along with a GraphicalMethods static class to hold static methods for certain shape-drawing processes.

These are the steps used in creating a drawable shape:

4.1.1 Rasterisation

Rasterisation is the creation of a series of pixels from a vector image. In this project, rasterisation is used to create an outline of a shape based on the desired type of shape, the centre of the shape, and any extra information the desired shape type requires. A DRectangle only requires the vectors of its vertices to generate the outline, but the vectors passed into GraphicalMethod.GenerateOutline() for a rectangle calls the same method as the DPolygon. DPolygons need an array of their vertices to be passed in, which the vertices of the rectangle are also considered as.

The method loops through each vertex, pairing it up with the next vertex in the array (wrapping around to the first vertex to pair up with the last), and then using Bresenham's Line Algorithm**[8]** to rasterise the line that's formed between the pair. The points that rest upon the line are added to a list of points named outline, and returned back to the DRectangle/DPolygon class as its outline.

For DCircle, an outline could be generated just by going through each pixel around its bounding box, checking which ones are closest to its radius. However, this is quite inefficient, and although Bresenham's Midpoint Circle Algorithm**[8]** can do a good job, it still uses floats which can be considered inefficient for the purposes of redrawing an outline every frame. Hence, an integer-only version using the least number of operations was chosen instead: Jesko's Method**[9]**. It sets up three variables before the main loop, then for every pixel in the circles octant, it only uses 5 operations, all only using integers (and a bool for an if-statement). During each loop, a method is called to place the pixel in each octant of the circle, as it's easy to place in the eight places at once only via mirroring (negating) around the centre coordinates.

Once again, the list of points is returned to DCircle, and is used as the outline for the next step in the process of drawing in a filled shape.

```
Point c = new Point(Utility.Round(centre.X), Utility.Round(centre.Y));
int r = (int)radius;

int x = r, y = 0;
int t1 = r >> 4;
int t2;

while (x ≥ y)
{
    PlaceInOctants(ref outline, (c.X, c.Y), (x, y));

    t1 += ++y;
    t2 = t1 - x;
    if (t2 ≥ 0)
    {
        t1 = t2;
        x--;
    }
}
```

Figure 1: Jesko's Method in C#

4.1.2 Scan Conversion

Scan conversion is the use of a pixel outline, or rather, upper and lower bounds of a shape, to fill it and make a solid shape**[10]**. In the case of the drawable shapes of this project, they pass the outline generated via rasterisation into the generation methods for scan conversion to get a filled-in shape at the end. All shapes have two settings to choose from: *scan lines* scan conversion, or *pixel* scan conversion. These end up using an almost identical method to each other, with the only difference being that scan lines returns a list of height-equals-one rectangles that's native to MonoGame to draw instead of a list of all the pixels to draw. Though there was planned use for the pixel scan conversion, it didn't actually end up being used at all, as scan lines are significantly more efficient – drawing lines in a single draw call instead of drawing a line in multiple calls as pixels.

This time, DRectangle is by its lonesome as DCircle and DPolygon share similar methods (though still separate). DRectangle just passes in its bounding box rectangle to the GenerateLineFill() function of GraphicalMethods, and all that's done is that a loop goes down every pixel of the height of the rectangle, creating a rectangle of height 1, and width the same as the rectangle passed into the function. The was reason behind creation of multiple lines within the full rectangle, instead of just returning the full rectangle (as that can be easily drawn in a single call by MonoGame), similar to the planned functionality of pixel scan conversion. This planned functionality was being able to pass in a predicate of some kind into the drawable shape's draw function, so that different pixels or different lines could be different colours. For example, if there's a fully circular attack, then it could be telegraphed by opacity starting in the middle and moving outwards from the centre, by using a predicate which tests if a pixel is a certain distance away from the centre to be set to opaque. This functionality never ended up being implemented, however.

The DCircle GenerateLineFill() method uses the innate symmetry of the circle down the vertical axis to be able to quickly perform the scan conversion efficiently. It finds the height of the circle by multiplying the radius by two, and adding one (as the radius doesn't count the centre point). Then it finds the smallest value of Y of the circle, which is the centre Y value subtract the radius. The symmetry then comes into play by having a method named FindSymmetricalMinXBounds() called, with the outline, height, and minimum Y value of the circle. In this method, an array of integers is created with every value being the maximum value for the integer primitive type. This makes it easy to find the smallest value. Then the outline is completely looped though, finding each Y value for each pixel on the outline and seeing whether the X value is smaller than the current X value at that Y position in the array.

Once the outline is looped through, the array of the min X values is returned, and the length of the line to get to the max X values are found by finding the distance between each min X and the centre X, and adding that onto the centre X (this equates to `2 * (centre.X - minX[y])`). With this, the beginning of the line (min X) and the length of the line (distance multiplied by two) is found and can be used to construct each rectangle for the scan line conversion.

DPolygon has the most computationally intensive of the scan conversions. This is because it doesn't have any inherent symmetry that can be utilised like with DRectangle and DCircle. Instead, the outline, and the bottom and top Y values are needed. FindXBounds() is called, passing in the outline and the minimum and maximum Y values of the polygon. This does a similar thing to the symmetrical version, but two arrays are made instead. One for the minimum Xs, and one for the maximum. The minimum Xs array is filled with maximum values again, while the maximum is filled with the minimum integer values. From there, the outline is looped, checking the same as before, but with the addition of checking for the max X value at that Y value too.

From there, the arrays are both returned, and at each Y value in the full height of the polygon, the line has width `maxX[y] - minX[y]`. Following the same process now to return the new rectangles that form the scan conversion lines.

4.2 Ray-casting

Although ray casting does fall under the umbrella of 'collision', it will have its own section here. Ray casting is used in the game engine to let enemies know if there is nothing blocking the line of sight between itself and the player. Admittedly, standard collision detection between the ray line converted to a very, very thin rectangle that's been rotated, and the walls on the tile-map may have sufficed, but it's not very efficient considering ray casting is supposed to be an extremely lightweight process. Therefore, custom ray casting was coded.

A ray, which, in the code, is called Ray2D, is a struct holding the information of a start and end vector of the ray, the direction of the ray, the length of it, and the vector between the start and end vectors (`end - start`). This can be converted into something described in the code as a LinearEquation, as this makes it easier to check whether the ray is intersecting any shapes. This converts the ray into the well-known mathematical form of a straight line: $y = mx + c$, with the values of M and C as members of the struct. This makes it easy to check if a point is on the line, as a vector can just be used to substitute the values of Y and X into the equation, and checking if each side is equal. There is a complication, which is that floats are

estimations of decimal values, and often times has trouble being precise enough for the purpose of equality-checking, which led to the creation of a utility function named NearlyEquals() which compares two floats based on if they're close enough to be called equal (within a certain margin of error).

To find whether a ray intersects with a circle, the three things that need to be checked are: if the vector between the beginning of the ray and the centre of the circle, projected onto the ray and added to the start of the ray is within the radius of the circle, or if the end of the ray is within the radius of the circle, and if the previous one is true, then if the projected point from before is on the ray (using the linear equation from beforehand).

To find whether a ray intersects a rectangle, the rectangle must be converted to a list of vectors describing its vertices, then it must be checked as a polygon. Now, to find whether a ray intersects a polygon, the ray is checked against every single edge of the polygon. The edges of the polygon are found by pairing vectors in the vertices list. If only a single edge is intersected by the ray, then the ray intersects the polygon. To check if the ray intersects the polygon edge, both are passed into the LineIntersectsLine() function as lines defined by a start and end vector. Values u and t are found by assuming that line A intersects line B at line A start + the full vector between the start and end of line A multiplied by some value t, with line be intersecting line A similarly, but using u as the multiplicative value. Due to u and t being coefficients to mark a point along the full line, they both have to be between 0 and 1 to be intersecting. To find u and t, the denominator can be found that requires a numerator to get the value of u or t. The denominator is calculated by calculating the cross product between the full vector between the starts and ends of line A and line B. The point of calculating the denominator first before the numerators is that the denominator needs to be checked. Specifically, it needs to be checked if it is equal to 0. If it is, then it obviously can't be used as dividing by 0 is undefined. Actually, a denominator of 0 in this instance means that lines A and B are collinear.

The numerator for u is found by calculating the cross product between the vector between start of line B and the start of line A, and the full vector from the start to end of line A. It's similar for t, except, instead of using the full vector for line A, the full vector for line B is used. In the case that lines A and B are collinear, it's important that it isn't disregarded as the lines never meeting, as they can still intersect. If the numerator for u is 0, then that shows that both lines fall upon the same line. However, it must be tested whether any of the points on each of the lines overlap, or that either is a segment of the other. If that is the case, then the lines do in fact intersect as they are directly touching each other.

If the lines are not collinear, then the simple task of finding u and t by dividing the numerators for each by the non-zero denominator found earlier (non-zero because it's already been checked if they're collinear). Once u and t are found, then if they're both between 0 and 1, then the lines intersect.

4.3 Collision

In this project, collision was completely rewritten three times. Initially, only axis-aligned bounding-box collision was used due to its incredibly low computational requirements**[11]**. However, sprites became circular and there was a need to alter collision to allow collisions against circles. Further on, it was decided that attacks would use polygons rather than simple

circles or unrotated rectangles, so collision was rewritten again to permit polygonal collision. After many bugs from poor organisation of the collision system, as there was a lot of repeated code in different classes when they were supposed to collide the same way, a massive refactor occurred in which everything 'collision' was placed in the CollisionStuff namespace. Here, there are collision shapes – direct counterparts to the drawable shapes, ray casting, the main static Collision class which holds all the collision methods, and the CollisionDetails struct, which is used to order and resolve collisions for entities implementing the interface IRespondable, in addition to the basic collision interface ICollidable.

Collision works for a moving IRespondable entity by having two positions for it. The first is its actual position in the world. The second is the target position. The target position is where it would want to move, but before collision was handled. And after every resolution to each collision, the target position would change, and when the collision was fully resolved, the entity would then move to the target position. The way collision was ordered to be resolved was by using the CollisionDetails struct that comes with the CollidesWith() method and placing the collision with the largest IntersectionArea first.

Entities that implement ICollidable must have a Collider CShape (C stands for collidable). CShape is an abstract base class for all the concrete collidable shapes: CRectangle, CCircle, CPolygon. All of them require a vector for the centre position, but all also need their own details. CRectangle needs width and height bounds, CCircle needs a radius, and CPolygon needs an array of vectors for its list of vertices. At the very least, they all contain their own definition for a bounding box. The bounding box is important because it mirrors the AABB collision from earlier, which is extremely fast and simple. This is either used as-is for rectangle-on-rectangle collision, or used as a preliminary test to see if its worth investigating the collision of other shape combinations closer.

4.3.1 Rectangular Collision & Resolution

Rectangular collision, as stated previously, is easy to check. All that occurs is the sides of the rectangle are checked if they overlap. Is the left side of rectangle A smaller than the right side of rectangle B, and is the left side of rectangle B smaller than the right side of rectangle A? A similar test is done for the top and bottom sides, returning true on collision if all individual tests return true.

For resolution, however it's slightly more complicated. The normal that will resolve the collision of rectangle A is found by finding the displacement from rectangle B to A, by subtracting the centre of B from the centre of A. The absolute values for each displacement axis is used to check which axis has the largest displacement; if the largest value is on the x-axis, then the way to resolve the collision is by pushing A and B away from each other by the intersection width, with the normal for A being along the x-axis too. The same can be said about the y-axis, if the largest absolute value in terms of displacement was on that axis.

4.3.2 Circular Collision & Resolution

Circular collision is also fairly simple to check, even more so than rectangular. However, this is only in terms of logic – rectangular collision is still the least computationally expensive.

The radii of circle A and B need to be added together, and if that is larger than the distance between the centres, then they are colliding.

To resolve the circular collision, all that needs to be done is to push the circles away from each other for the depth of the collision. The depth of the collision is just the radii added together, subtract the distance between the centres.

Unfortunately, it gets much worse. Rectangle-on-circle collision is much complicated, though, once polygonal collision is reached, this will be thought of as quite simple. To detect rectangle-on-circle collision, the rectangle is localised so that its centre is (0,0), and the circle is localised around the rectangle, using absolute values for its centre now because of the symmetrical nature of both circles and rectangles. The cardinal points of the circle are then tested against the bounds of the rectangle, and if they overlap, then they are colliding. If not, then the corners of the rectangle are checked to see if they fall within the radius of the circle. If the positive corner of the rectangle (calculated by taking the size of the rectangle and dividing by two) subtracted from the radius of the circle (which is positive anyways due to the absolute values and the localisation around (0,0)) is larger than zero, then the corner is within the radius of the circle and there is collision.

To resolve this rectangle-on-circle collision, two different methods are available depending on how the circle and rectangle collided. If the cardinal points of the circle were within the bounds of the rectangle, then rectangular collision resolution can be performed. If a corner of the rectangle was within the radius of the circle, then circular collision resolution can be performed.

4.3.3 Polygonal Collision & Resolution

Polygons are much more complex than circles and rectangles. There is no guaranteed symmetry to take advantage of, so the Separating-Axis Theorem**[12]** must be used instead. This theorem is one that proposes that if there are two convex polygons on a plane, and they aren't colliding, then there must be a line (axis) that can be drawn between the two. It's like taking the shadow of the shapes at all angles and seeing if there's any gaps in the shadow. Instead of testing every single angle, it's been proven that the only necessary axes to test are the normals of ones that make up the polygons.

For polygonal collision (and polygon-on-rectangle, as the rectangle is taken as a polygon with four vertices), each polygon is gone through at a time. Polygon A's first edge has the normal for it found, and both polygon A and B are tested against this normal. To test against the normal axis, each vertex in each polygon is used in a dot product with the axis, which projects the point onto the axis. Along this axis, the maximum and minimum points for each polygon can be found. And once they are found for both shapes on the first axis, it is checked whether there is any overlap. Overlap is calculated by taking the minimum between `B max – A min`, and `A max – B min`. If the overlap is less than or equal to 0, then there is no collision (clearly, as there is no overlap). This is repeated for each edge on each polygon until no overlap is found, because if there is a single axis that has no overlap, then that means the polygons can have that line drawn between them, showing that they aren't colliding.

To resolve polygonal collisions, when checking for overlap, the smallest overlap must be found and kept throughout all the testing of projecting vertices. The normal for A to move

away from B is also dependent on whether the overlap ended up being `B max – A min`, or `A max – B min`. If it was `B max – A min`, then the normal is equal to the axis, and if not, then it's equal to the axis negated. The normal of the depth must be calculated however, and that's done by dividing the depth by the length of the normal of A. Once that has happened, though, the normal of A should also be normalised, so its length is equal to 1. This should be repeated with the normal of B too, but only when testing the polygon B against A, instead of A against B to find the normal of A. This leaves two depth values. The depth value of A and the depth value of B. The solution is to take the smallest depth value, and use the normal for that value to (just make the other normal the negative of the smallest depth normal).

For polygon-on-circle collision, although the axis is still going to be found from the polygon edge normals, and the projection for that polygon still works the same in finding the max and min projected values, the circle only has one max and min all the way round on the circumference. This max and min values are found by getting the circumference point by normalising the axis and multiplying by the radius of the circle, then taking the dot product of the axis with the centre of the circle ± the circumference point. Since it's ambiguous as to which direction from the centre to the circumference point is the max or the min, the min was just made equal to the smaller of the two, and the max, the larger. Everything else is the same, but with just one more axis to test. The final axis is the closest vertex the polygon has to the circle, subtract the vector that is the centre of the circle. Once the same projections and max and min values have been found, it's the exact same with polygonal collision. Even the resolution.

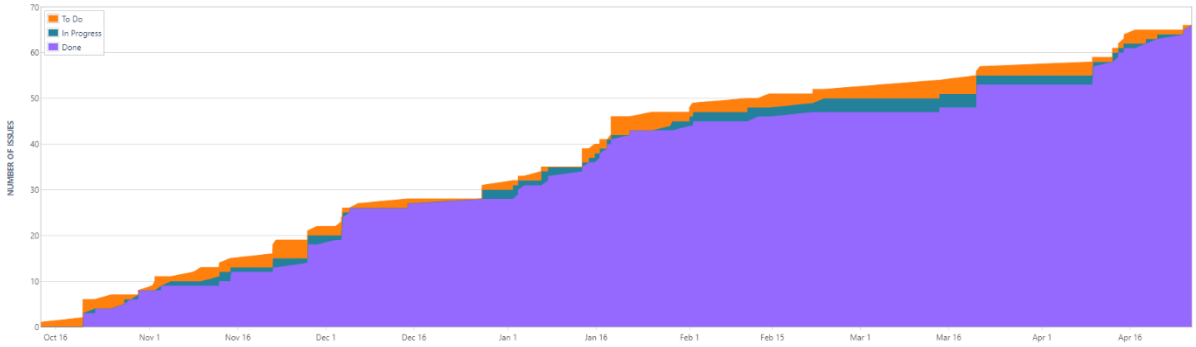
4.4 Pathfinding

The pathfinding used in the game and engine boils down to just the A* algorithm**[13]**. A* is a modification of Dijkstra's shortest path algorithm**[14]** for graphs, but with a heuristic. The way it works is by having an open list, ordered by lowest F-value first, and a closed list. In A*, or at least, in the implementation in this project, the F-value of a node is calculated by adding the G-value (the least amount of steps to get to this node) and the H-value (the heuristic, which is Euclidean distance here). The way it was ordered first was by using a priority queue for store them. The closed list also got a speed upgrade by the use of a hash-set, so it didn't need to be search entirely through every time when checking to see if a node position was already in there.

A* works by adding the starting position to the open list as a node with G-value equal to 0, and the shortest path parent to it being null. Then the main loop is entered, checking every time if the open list has any elements in it, and if it has, then it pops that value off the queue. From there, if the just popped node has the position of the target position, then a path is made backwards by looping through all the shortest path parents to trace the path back to the first node. If not, though, then all the adjacent nodes from this one are generated and looped through. If the closed list contains the node position, then skip this adjacent node. If the open list contains the node position, and the one in the open list has a smaller F-value than the adjacent node, then skip this adjacent node, else, the adjacent node should be added to the open list (in the correct place).

Once the path is returned, it has all the corners extracted, because it's not worth having all tiles on the path be targets to walk back over, since only the corners are needed. The tiles are then changed back into vectors by multiplying them by the current tile-maps tile size, and adding an extra half of the tile size.

5 Project Planning



This is the Jira Cumulative Flow Diagram. It shows how there's usually been steady work done on this project. There have been one or two plateaus, mainly between early- to late-December, and late-February to mid-March. These are notorious periods of time, as they are when the coursework for other modules are due. Before/during these periods, it was brought up to the Supervisor that not much, if any at all, work would be done on the Capstone Project.

In general, however, work was intended to be done on Weekends, with at least one major addition per two weeks. This was able to be kept up for quite a while, but not forever, as the easier-to-see developments became far and few between, and more complex features were being added that took longer and didn't have as much to show for.

Weekly meetings were very rarely not upheld with the Supervisor, giving good motivation to always have something new to talk about having done.

6 Conclusion

From the design to the algorithms, creative to the concrete, this project ended successfully in creating a tile-based RPG engine. Although basic, a lot of complex functionality is present, with a lot of control over what can be done with the engine. The example game, though lacking much substance, still portrays a lot of what the engine can do.

The author over-scoped this project massively in the beginning. But even their supervisor notifying them of this, and them coming to a realisation about it halfway through, a surprisingly substantial amount of work was done. Things had to be dialled back slightly from what was originally intended, but that may just become future work. The majority of what was intended to be delivered by the end, was, indeed, delivered though.

In the future, perhaps the testing purpose of the game could come to fruition, which was to test whether players would have an easier time defeating a boss or enemies if their attacks matched up to background music that was playing. Now that there is a solid base for the project in the game engine, continuing and doing this in the future seems much more likely, even when the initial goal had to be given up on.

One thing that would definitely serve to improve the engine, is including spritesheet animations. So perhaps when an enemy winds up an attack, the player actually sees a weapon get drawn backwards before it slices forward. But other than that, the author is content where the project is leaving off for now.

7 References

[1] M. J. P. Wolf, *Before the Crash: Early Video Game History*. Detroit: Wayne State University Press,.

[2] Enterbrain, *『RPG ツクール XP』英語版 海外サイトにてダウンロード販売を開始*, Kadokawa. Available: (via a Wayback Machine archive) https://web.archive.org/web/20061208105128/http://www.enterbrain.co.jp/jp/c_outline/pdf/tkoolxp_global.pdf

[3] FromSoftware Inc., *Elden Ring DLC "Shadow of the Erdtree" Press Release*. www.fromsoftware.jp. Available: https://www.fromsoftware.jp/ww/pressrelease_detail.html?tgt=20240222_eldenring_dlc_debut

[4] *Steam Search*, store.steampowered.com. Available: https://store.steampowered.com/search/?tags=29482&category1=998%2C10&ndl=1. [Accessed: April 22-24, 2024]

[5] SteamDB, *Steam Game Release Summary by Year*, steamdb.info. Available: https://steamdb.info/stats/releases/ [Accessed: April 22, 2024]

[6] Itay Keren, *Scroll Back: The Theory and Practice of Cameras in Side-Scrollers*.

[7] FishyMint, *[8x8] Perch! - A Tiny RPG Experience*, CurseForge, April 2, 2020. Available: https://www.curseforge.com/minecraft/texture-packs/x8-perch-a-tiny-rpg-experience. [used version for 1.19]

[8] J. E. Bresenham, "Algorithm for computer control of a digital plotter," in *IBM Systems Journal*, vol. 4, no. 1, pp. 25-30, 1965, doi: 10.1147/sj.41.0025.

[9] "Algorithms – Systemberatung Schwarzer." https://schwarzers.com/algorithms/ [Accessed: April 26, 2024]

[10] G. Enderle, M. Grave, and F. Lillehagen, *Advances in Computer Graphics I*. Springer Science & Business Media, 2013.

[11] Yiyu Cai and Sui Lin Goei, *Simulations, Serious Games and Their Applications*. Singapore Springer Singapore, Imprint: Springer, 2014.

[12] J. Huynh, "Separating Axis Theorem for Oriented Bounding Boxes." Available: https://www.jkh.me/files/tutorials/Separating%20Axis%20Theorem%20for%20Oriented%20Bounding%20Boxes.pdf

[13] František Duchoň, Andrej Babinec, Martin Kajan, Peter Beňo, Martin Florek, Tomáš Fico, Ladislav Jurišica, *Path Planning with Modified a Star Algorithm for a Mobile Robot*, Procedia Engineering, Volume 96, 2014, Pages 59-69, ISSN 1877-7058, https://doi.org/10.1016/j.proeng.2014.12.098. Available: https://www.sciencedirect.com/science/article/pii/S187770581403149X

[14] M. Noto and H. Sato, "A method for the shortest path search by extended Dijkstra algorithm," Smc 2000 conference proceedings. 2000 ieee international conference on systems, man and cybernetics. 'cybernetics evolving to systems, humans, organizations, and their complex interactions' (cat. no.0, Nashville, TN, USA, 2000, pp. 2316-2320 vol.3, doi: 10.1109/ICSMC.2000.886462.

8 Appendix

1. Using “Games” and “Demos” as search tags on Steam, 116,412 results are returned. Adding “Souls-like” to the search tags returns 2138 results.

$$100 \times \frac{2138}{116{,}412} = 1.857\% \ (3dp)$$

Note: result counts obtained on April 22, 2024**[4]**. Exact value may vary depending on date obtained.

2. Using “Games”, “Demos”, and “Souls-like” search tags on Steam returns 2138 results. Adding “Metroidvania” to the search tags returns 436 results.

$$100 \times \frac{436}{2138} = 20.393\% \ (3dp)$$

Note: result counts obtained on April 23, 2024**[4]**. Exact value may vary depending on date obtained.