# OCR Computer Science NEA

A group of people with text

Description automatically generated

Bullet hell A-level project using pygame.

[OCR Computer Science NEA 1](#_Toc188559744)

[Analysis 4](#_Toc188559745)

[Describing the problem. 4](#_Toc188559746)

[The goal 4](#_Toc188559747)

[Explanation of shoot ‘em up 4](#_Toc188559748)

[Explanation of bullet hell 5](#_Toc188559749)

[Identifying the stakeholders. 5](#_Toc188559750)

[Age and motivation 5](#_Toc188559751)

[Explanation 5](#_Toc188559752)

[Form questions 7](#_Toc188559753)

[Form responses 7](#_Toc188559754)

[Researching and identifying features. 12](#_Toc188559755)

[Inspiration 12](#_Toc188559756)

[Touhou 12](#_Toc188559757)

[Osu! 15](#_Toc188559758)

[Geometry dash 17](#_Toc188559759)

[Limitations 18](#_Toc188559760)

[Outlining computational methods. 19](#_Toc188559761)

[Abstraction 19](#_Toc188559762)

[Visualisation 20](#_Toc188559763)

[Logic and Pipelining 21](#_Toc188559764)

[Specifying game requirements. 23](#_Toc188559765)

[Development hardware used 23](#_Toc188559766)

[Hardware requirements 23](#_Toc188559767)

[Development software used 23](#_Toc188559768)

[Software requirements 24](#_Toc188559769)

[Final success criteria 24](#_Toc188559770)

[The scope of the solution 24](#_Toc188559771)

[Design 28](#_Toc188559772)

[Structure of the solution 28](#_Toc188559773)

[Flow chart 28](#_Toc188559774)

[Decomposition of the problem 28](#_Toc188559775)

[Music handler 28](#_Toc188559776)

[Input manager 29](#_Toc188559777)

[Event handler 29](#_Toc188559778)

[State manager 29](#_Toc188559779)

[Quitting 30](#_Toc188559780)

[Options 30](#_Toc188559781)

[UI collision checks 30](#_Toc188559782)

[Playing the game 30](#_Toc188559783)

[Pausing the game 31](#_Toc188559784)

[Collision management 31](#_Toc188559785)

[Updating entities 31](#_Toc188559786)

[Editor 31](#_Toc188559787)

[Saving a stage 32](#_Toc188559788)

[Creating stages and waves 32](#_Toc188559789)

[Enemy wave 32](#_Toc188559790)

[Boss wave 33](#_Toc188559791)

[Identifying and justifying object oriented techniques. 33](#_Toc188559792)

[Load class 33](#_Toc188559793)

[Audio class 34](#_Toc188559794)

[Widget base class 35](#_Toc188559795)

[Text class 36](#_Toc188559796)

[Button classes 36](#_Toc188559797)

[Popup class 37](#_Toc188559798)

[State class 38](#_Toc188559799)

[Game state classes 39](#_Toc188559800)

[Entity class 40](#_Toc188559801)

[Player and Enemy classes 40](#_Toc188559802)

[Projectile class 42](#_Toc188559803)

[Identifying and justifying other data structures. 42](#_Toc188559804)

[Global variables 42](#_Toc188559805)

[Lists, tuples, dictionaries and sets 43](#_Toc188559806)

[Strings, integers, Booleans, etc. 43](#_Toc188559807)

[Storage techniques 44](#_Toc188559808)

[Designing the algorithms to be used. 46](#_Toc188559809)

[General game algorithms 46](#_Toc188559810)

[State algorithms 47](#_Toc188559811)

[Audio algorithms 48](#_Toc188559812)

[User interface algorithms 49](#_Toc188559813)

[Entity algorithms 51](#_Toc188559814)

[Evaluation 53](#_Toc188559815)

# Analysis

## Describing the problem.

### The goal

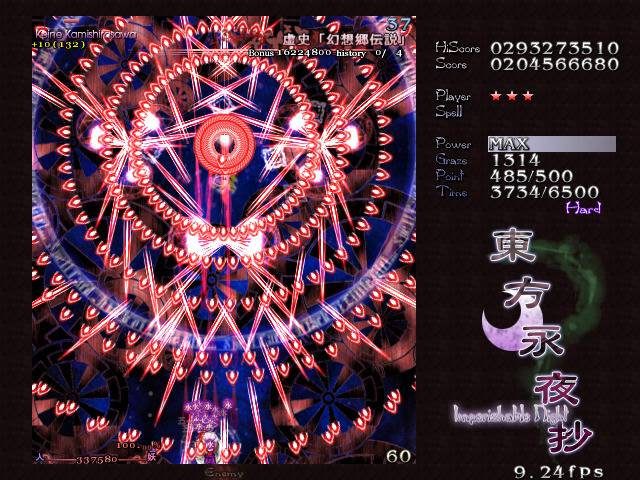
Using PyCharm Community Edition 2024, I will develop a ‘bullet hell’ game using the module pygame from python, as well as JSON (JavaScript Object Notation) to store metadata. A bullet hell is a subgenre of the popular arcade game genre of ‘shmup’ (or shoot ‘em up).

### Explanation of shoot ‘em up

This style of games is commonly described as a 2D shooter where a spaceship or any other substitution for the player moves around the screen and dodges enemies shooting projectiles. The objective is to defeat the enemies on the screen using the players gun or any other form of projectile firing tool. The most famous arcade example of this genre being ‘Space Invaders’ featuring the iconic aliens: 

### Explanation of bullet hell

The bullet hell subgenre takes this to the next level by keeping the main objective of defeating enemies but shifting the objective by ramping up the number of bullets on the screen until the focus of the player is to dodge the bullets and survive and defeating enemies becomes a side goal. An example of a Japanese arcade bullet hell called ‘The Touhou Project’ (or just ‘Touhou’) can be seen below:



## Identifying the stakeholders.

### Age and motivation

The target audience for my game are people anywhere up from 14 years old but will probably be most prevalent with college students who have enjoy playing video games for the personal improvement and sense of achievement it gives, although older video game enthusiasts who enjoy the modernized approach of the arcade origins of my game are also a feasible demographic. The target platform will be most operating systems for computers. The game will be developed in windows 10/11 and Linux mint (a Linux distribution based on Debian). While the game will not be tested on macOS, the game will probably run on apple computers as well.

### Explanation

The game is suitable for this demographic as it has a high ceiling (there is room for a large amount of skill improvement) however has a low skill floor as the mechanics of a bullet hell are simple to grasp by any person. This means the game can cater to a large audience while pleasing the players that want to push the boundaries of what’s possible in terms of difficulty and spend a lot of time in the game.

### Form questions

To further gather opinions from my target audience I set up a form which I gave out to my peers at my college and other friends online.

The questions featured in this form are:

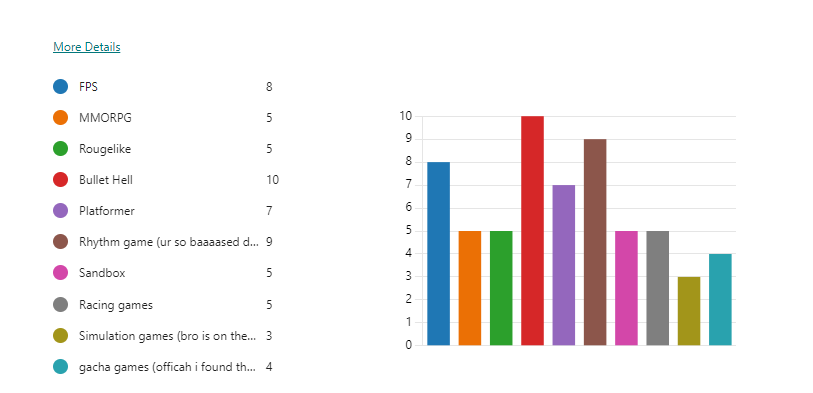
1. How would you describe your time spent gaming per week (in hours)?
2. What game genres have you played and enjoyed?
3. Rate the importance of these features when trying out new games:
   1. Difficulty
   2. Depth of mechanics
   3. Music
   4. Story
   5. Immersion
4. Why do you choose to spend your free time gaming?

### Form responses

A white background with black text

Description automatically generatedA screenshot of a computer

Description automatically generatedThe data showed that most people (aged 16-24) spend a large amount of time gaming, meaning they were relatively skilled and would most likely enjoy a game which provides a challenging experience.



Furthermore, when asked about people’s preferences in the genre of games being made, the most popular response was bullet hell, which means most stakeholders would have some experience with shoot ‘em up games and thus would expect the challenge accompanying similarly styled games. The second and third most popular game genres were both games that are entirely based on the user’s skill that being rhythm games (games focusing on inputs that are synchronised to beat with a song) and FPS games (First-Person Shooter games focusing on aim as well as awareness in combat), proving that games that reward skill are popular among my identified stakeholders.

The next set of questions asked people to rate how much certain game features impact their opinion of said game on a scale: 1 means the game feature has no impact on how much they enjoy the game, while 5 means the feature plays a big part in if they continue to play and enjoy the game. A graph with blue bars

Description automatically generated with medium confidence

The answers to this question made it apparent that graphics shouldn’t be the main drawing point of my game, which makes sense since that would exclude people with weaker hardware from playing my game. This would in turn make the game less popular to the public. However there should still be some attempt at good game assets even if the resolution of the game doesn’t matter. I decided to use pixel art as it takes less time to make than regular graphics while not looking low quality or rushed.

A graph with blue squares

Description automatically generated with medium confidenceA screenshot of a graph

Description automatically generatedWhile some may not always enjoy playing an intense, difficulty focused game all the time, it is reasonable that there are other slower paced games people play when relaxing. This is also true for games that require a deep understanding of its mechanics. However, when making my game I am assuming people are playing for the thrill and difficulty and are expecting some depth of mechanics that will take time to master.

To clarify, by depth of mechanics I am referring to the number of actions able to be performed at each moment, and the difficulty to perform those. E.g., while a bullet hell might be difficult, its mechanics are usually quite simple as all that is required are movement keys and a button to shoot which is usually just held down since the concept of ammo doesn’t exist in most. A mechanically deep game would be one where the user is required to know a lot of keybinds and actions that require significant skill to execute in short succession such as management simulation games since they usually involve large amounts of keybinds, or any game with movement bugs such as the famous b-hops (‘bunny hops’) present in a lot of 3D first person games where a certain combination of inputs allows for jumping continuously in a precise fashion to conserve momentum or increase movement speed.

A screenshot of a graph

Description automatically generated

While I won’t be marked for the quality of the music in my game, I made sure to feature music from other game OSTs (Original Soundtracks) that I have enjoyed and tried to fit said music to the theme of my game. Since I also produce music I decided to remix the song I picked as my main theme song for my game to keep the game original.

A graph with blue bars and white text

Description automatically generatedHowever, even though most people expect a good story in the game their playing, this is misleading in the context of my game, since bullet hell games have never had a focus on the story since the games were meant to be played repeatedly at an arcade, rather than a singular playthrough e.g. something like a horror game which normally people don’t play more than once fully. I decided to include some fun references to other games. This will leave the gameplay experience of those who don’t understand the reference the same, while providing a cool ‘easter egg’ (a secret for players to find) for those who understand it.

A screenshot of a graph

Description automatically generatedWhat I mentioned above about the inclusion of story can be applied to immersion as well: while people may enjoy immersive/realistic games, people playing bullet hells barely expect any immersion or realism as this is simply not the nature of the genre. However, I do plan to include vague theming relating to space as I plan to make my game nearly only black with white accents and space themed music.

A screenshot of a video game

Description automatically generatedHere I saw that playing with friends is the most popular motivation for gaming, with simply having free time being close behind. This means while making the game somehow multiplayer would definitely attract more customers, I concluded that it was not worth it due to how bullet hells were not designed to be played together meaning not only would I have to come up of a way of making bullet hells compatible with multiplayer, but a large amount of development time and resources (including the cost of running an entire online server) would have to be allocated to making this feature as well.

To summarise, based on the form, these are the requirements I ended up setting:

* Minimal focus on graphics – pixel art.
* Large focus on difficulty and gameplay experience.
* Make the game replayable.
* Add memorable background music.
* Add various interesting mechanics.
* Stick to a general theme but don’t be constrained by it.

As for the reasoning for the target platform, I chose PC as the game was made with a keyboard and mouse in mind. A computer makes most sense for this hardware. While you can connect a keyboard and mouse to some consoles and mobile phones, since I will be developing the game on a PC, developing the game for mobile or console would take too much time. While mobile is the most popular platform for gaming, most mobile gamers are not as competitive as PC players, so the majority of the audience I am trying to make my game for plays games on PC.

## Researching and identifying features.

### Inspiration

My game will feature heavy inspiration from Touhou and other games I’ve played, bullet hell or not, adapted from a 4:3 display ratio to a more modern 16:9 ratio leaving a wide amount of room around the player horizontally allowing more creative bullet patterns and game mechanics.

As mentioned above I have played the bullet hell Touhou extensively and used other games I have played, as well as game design concepts from books I have read to approach the problem. Here I will describe my findings in those games and present my method of making a bullet hell game.

### Touhou

The Touhou Project is a series of 2D vertically scrolling bullet hell shooting games made by Team Shanghai Alice. They were originally available to play on arcade, however, became available to play on PCs following the rise in availability of computers.

Since there are over 20 games featured in The Touhou Project, I will be referring to ‘Touhou Koumakyou: the Embodiment of Scarlet Devil’, the sixth game in the franchise, as I have the most time spent playing that game. From now on I will refer to this game as simply ‘Touhou 6’.

* Touhou 6 feature a set of available characters to pick from which change up the gameplay. Each character has multiple available shot types to pick from e.g. homing bullets in a wide spread, or fast-moving narrow bullets in a straight line. Below is an example of a ‘shot type’ from another Touhou game. It has a wide spread of blue bullets and 2 red lasers.

A screenshot of a video game

Description automatically generated

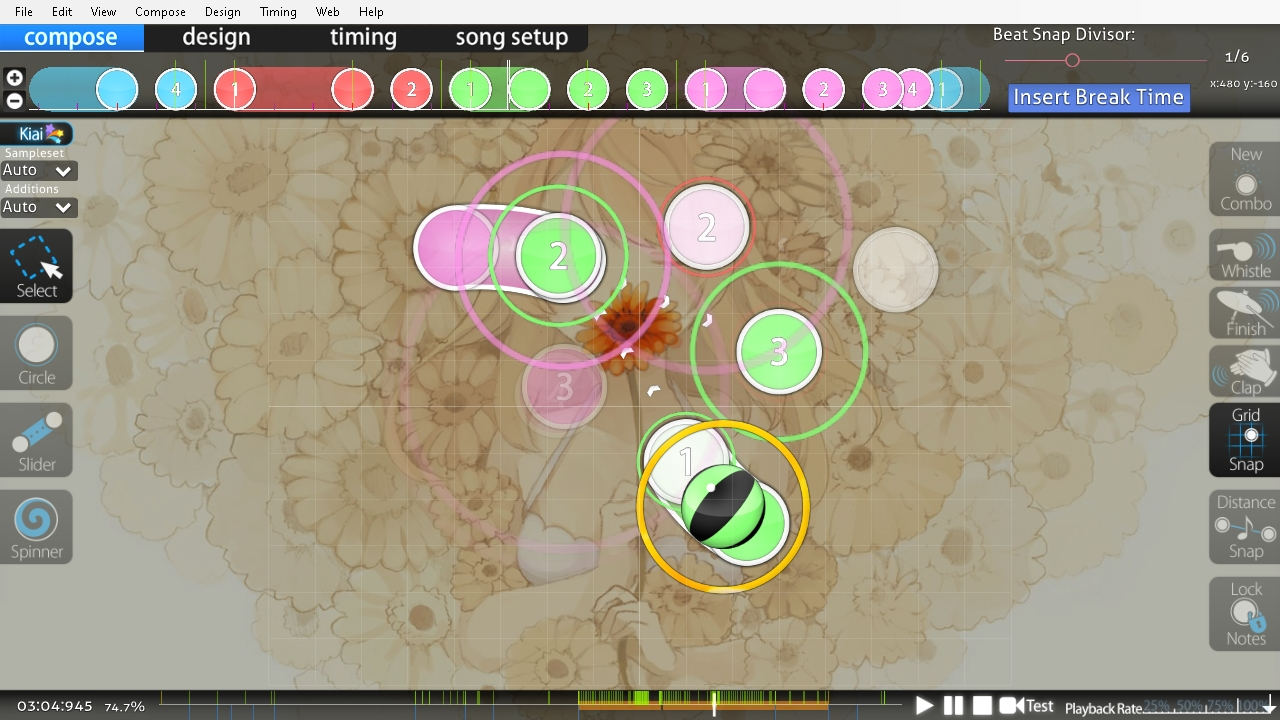
* When starting the game, it plays from start to finish all at once. The time it took me to beat Touhou 6 (not counting the vast majority of the time which is spent improving at the game) is around 30 minutes to an hour. The game is split up into stages, which are further split up into sections. Each stage has a theme and background to go along with it, e.g. a haunted mansion or a frozen lake. Each stage is split up into 4 sections.
* Section 1 and 3 consist of hordes of weaker enemies like what you would expect from a classic bullet hell, while stages 2 and 4 feature a singular stronger enemy. Usually, the stage 2 enemy is a weaker version of the boss that prepares the player for stage 4.
* The boss enemies (stage 2 and stage 4 enemies) in Touhou 6 have specific attacks, or bullet ‘patterns’ (arrangements of bullets fired by the boss) that can be learned to improve at fighting the boss. These are identified by ‘spell cards’ that announces the name of the attack. The attack can be either ended by dealing enough damage to the boss, or by running out of time (each spell card starts a timer that ticks down to signify the duration of the attack). The final spell card sometimes cannot be ended by dealing enough damage and must instead by ended purely by surviving the attack.
* The game also features a ‘bomb’ system. This is an expendable special attack that deletes all bullets on the screen and makes the character invulnerable for a short period, while dealing massive damage to the enemies on screen. The type of bomb is also dependent on the shot type as well as character picked.
* It also features a level system, whereby killing enemies drops a pickup which contribute to levelling up. Levels increase the number of bullets fired by the player, and upon reaching the level cap, all pickups turn into bonus points that reward an extra life after enough has been picked up (life system explained below). Referring back to the image of the shot type, you can see that the power bar is at MAX. If this was level one, you might expect way less blue bullets and the lack of the red lasers.
* The game uses a health system, with a life being lost each time the player collides with the enemy. Upon colliding, all enemies and their health persist through death and the game continues from the exact point the player died at. However, all bullets are cleared from the screen, the player loses some levels and becomes immune to dying again for a short period to help prepare for the next attack.
* A handy game mechanic is present in the form of ‘death bombing’ that is triggered by pressing the key to activate a bomb and is available for a very tiny period after dying. At this point the player has not gone through the death and respawn animation yet, so the bomb triggers as if the player is still alive. This will not take any lives away from the player and the game will play out as if the player had bombed to avoid damage instead of dying and having 1 health removed. However this mechanic is very hard to pull off consistently and mostly occurs when you were already going to use a bomb to dodge a bullet but pressed it a bit later than expected resulting in you dying for a split second before the special death bomb activated.

From my analysis of Touhou 6, I have decided to implement the following features:

* The player has a set number of lives, with the game ending when the lives reach zero. Having the player’s game end immediately at any mistake will make the game not be fun to learn, so introducing a set amount of lives keeps some difficulty but leaves room for error to learn the game.
* Touching a bullet instantly kills the player and removes a life and resets the position of the user to the bottom of the screen. In order to have the game’s focus be on dodging bullets, having hitting bullets provide immediate feedback provides an intuitive goal.
* There is some form of ability to make the player invulnerable or assist in defeating enemies. A special ability provides diversity in the gameplay in that different characters in the game can have different abilities with different benefits. In addition to this an ability to become invulnerable or removing all bullets on the screen provides a way to skip some bullet patterns that are problematic to learn improves the enjoyment of the game.
* Using the ability mentioned within a short time-frame of dying reverts the effects of the death. Rewarding the player for their reflexes when getting hit by a bullet raises the games skill ceiling and helps remove deaths that would feel unfair to the player.
* There are stages featuring different waves and a set theme. This keeps the game diverse and makes it obvious that the player is progressing through the game. It also acts as a way to visualise how far into the game you are by paying attention to which stage you are in.
* Stages feature enemies appearing in waves with a final challenge at the end. The fights being formatted into waves means that the player has time to rest while new enemies are appearing after the previous ones. It also means the player cannot be overwhelmed by an increasing amount of enemies if the player has not defeated them fast enough. This helps keep the focus of the game as dodging enemy projectiles rather than focusing on levelling up to clear enemies faster. The challenge marks the end of the stage and provides a different style of gameplay and serves as the focus of the entire stage and main difficulty of the game.
* Attacks from tough enemies will be telegraphed in some way. This provides the player with a way for anticipating and dodging attacks and means they get less frustrated when dying.
* There are multiple characters to select from. This provides game replayability and provides a different playstyle for each character which appeals to a larger audience as there are more ways of playing to choose from e.g. fast character, character that deals a lot of damage but can easily die.
* Characters have multiple types of bullets to choose from. This means each character has an added layer of complexity and more playstyles and their variations can be derived from less characters (which are harder to design and make than bullet types)
* Enemies drop collectibles which upgrade the player’s bullet. This provides an incentive to take risks in killing as many enemies as possible and getting closer to the enemy to pick up the upgrade. However the difference between the lowest level and max level should not be big enough where a person not focusing on collecting the upgrades would struggle to kill any enemies.

### Osu!

Osu! is a free-to-play rhythm game originally created and self-published by Australian developer Dean Herbert. Inspired by gameplay of the Osu! Tatakae! Ouendan series (released on the 28th of July 2005), it was released for Microsoft Windows on 16 September 2007. While Osu! is not a bullet hell game, I took great inspiration from Osu! when making the editor for my game. Osu! is a rhythm game where circles appear on the screen in set places and you have to aim and click them to the beat of a song playing. There are also sliders that you have to drag instead of click to the beat, and spinners which you have to spin as fast as possible. It features an editor for making ‘maps’ for different songs:



* Osu!’s main editor screen is split up into 5 UI (User Interface) elements: a side bar on the left, a sidebar on the right, a timeline at the top, a timeline at the bottom and a grid for placing objects that covers the rest of the screen.
* The timeline at the top has a header that can be dragged forward and back. The timeline counts in beats, set by the bpm (beats per minute) of the song, however the header can be moved in between beats by a set amount. The timeline acts as a slider with a set resolution that being some fraction of the bpm of the song used. The timeline also displays when an object appears in the song (more on this below).
* The left pane features different ‘modes’ the editor can be in, toggled by clicking between them (only one mode can be active at a time). The 4 modes are: select mode, circle mode, slider mode and spinner mode.
  + The select mode allows the user to select objects already placed and change their position on the screen by dragging them in the main grid or change their position in time by dragging the relevant object in the timeline.
  + The circle mode allows the user to place circles anywhere on the screen at the point of the header in the top timeline. The user can move the top header to place more circles at different points of the song.
  + The slider mode allows the user to do the same thing as circle mode. The only difference is that since sliders take time to hold through, they extend across in the timeline above.
  + The spinner mode does the same as mentioned above but for spinners.
* The right pane features utilities for editing existing objects placed on the screen.
* The bottom timeline is a condensed version of the entire song allowing for quick navigation to any point in the song. It also features a play button to see how the map appears when the song is playing.

The other screens of the editor include the design section which is for controlling the appearance of the map background, circles and general art aspects. The timing screen has utilities for changing the resolution of the top timeline mid song, for example changing the bpm fraction from 4 to 3 (in music this is equivalent to changing the time signature from 4/4 to 3/4, the song still remains on beat as this is independent from the songs bpm). Finally the song setup screen includes general information and data about the map such as name, song etc. It allows for saving and exporting the map as a file that can then be loaded up in order to edit or to play.

From my analysis of Osu!, I have determined I will implement the following features:

* My game will feature a timeline in some abstracted way. Enemies will be able to be placed on this ‘timeline’ in order to spawn on the screen at a specific point into the level. While some enemies do go away regardless of if the player killed them or not, some waves depend on being cleared entirely to progress. This would have to be displayed in some way on the timeline.
* In the case of bossfights (a wave featuring the boss enemy), the boss’s different attacks would also be able to be placed on the timeline.
* My game will feature a main grid or similar where enemies will be placed and their position will be able to be adjusted. However unlike Osu! multiple enemies will have the ability of appearing at once.
* The editor will feature modes for interacting with the main area like a select mode for highlighting enemies and a placing mode for placing new enemies on the screen. Once selected, an enemy will be able to have its data modified in a menu.
* The main area will feature a grid while will make aligning object easier. There will also be a manual way of typing in coordinates of objects as well.
* There will be a top header featuring tabs for the main editor I have been talking about above but also a different view for the general information about the stage being made.
* There will be a way of exporting and importing stages to and from files. These files will not only contain all the data for the editor but will also be used in order to play the stage.

### Geometry dash

Geometry Dash is a side-scrolling music platforming game series developed by Robert Topala. The game was released on 13 August 2013 on iOS and Android. In Geometry Dash, players control the movement of an icon by tapping and navigate music-based levels (called maps in Osu!) while avoiding obstacles such as spikes. This game is also not a bullet hell game, however features an editor I have used extensively similar to Osu!.

A screenshot of a video game

Description automatically generatedThe editor features a main bottom bar divided in to 3 sections:

* The leftmost tab, that features the 3 modes the editor can be in. Clicking on them also switches the contents of the middle tab but not the rightmost tab. Build mode allows you to place objects, edit mode allows for highlighting objects, and delete deletes objects.
* The middle tab contains the main contents based on the editor mode selected as mentioned above.
  + When in build mode, a list of selectable objects appear which can be clicked to select that object. That can then be placed on the main grid. When in the build tab, sub-tabs also appear above the bottom bar allowing for switching between different object types e.g. blocks that the player can jump on, spikes that kill the player, decorations that don’t affect the player etc.
  + When in edit mode, buttons for moving the object, rotating it, scaling it etc. appear.
  + When in delete mode, buttons for deleting objects are displayed. This includes filters for deleting such as delete block, delete spike, delete custom.
* The rightmost tab contains general buttons that modify the entire grid. The swipe button allows for placing blocks by swiping the finger across the screen (previously tapping would place blocks while swiping would move the grid). The free move button allows for moving objects off the grid, while snap means all objects are centred on the grid.
* The editor also features a layer system. This means that when objects are placed on one layer, they cannot be interacted with in any way on other layers, and have low opacity. This helps keep complex levels easy to build in, as when selecting a point on the screen where multiple objects are placed on the same tile, you have to click multiple times to select an object at the bottom. This wouldn’t be the case if objects were instead placed on multiple different layers.
* The editor features undo/redo buttons that can be pressed in order to undo the last action or redo the last action that was undone.
* I will also implement an abstracted version of the practice mode. This will allow people to playtest a stage by starting from a specific wave, or a way of playing the stage from the beginning to see how it plays out.

The game also features a level select menu where levels are stored in a UI list and can be selected to either play or edit. Levels can also be played in ‘practice mode’ where checkpoints can be placed during playing that respawn the player at that location when dying. This allows hard or long levels to be play-tested easier. From my analysis of Geometry Dash, I have decided to include the following features:

* A object select menu: in this case an enemy select window for selecting which enemy to place on the screen. Enemies will be sorted into groups depending on their features e.g. boss enemies. This menu will also be used for picking which boss attacks to place in boss fights.
* My game will also feature a screen for saving and loading stages, as well as a way to compile stages into full games (in this case a ‘game’ is just an arrangement of stages which will execute in succession.
* My game will also feature undo/redo buttons as well as a history log for viewing what was changed.

### Limitations

* While the best place to develop a game would be either in another more popular language like C++ (the most popular language for game development) or Rust for highly performant code, the time-frame and my background knowledge in python means that I don’t have the time to learn a new language, and even if I did already know it development would take longer since the languages are more complex. Python highlights and excels at rapid application development (RAD) which is a heuristic approach to making my game
* I also will not be able to use a game engine such as unreal engine or unity, even if it would greatly simplify the development of the game and provide the ability to greatly increase performance and quality of the graphics and visual effects. The time required to learn how to use such game engine would greatly offset the speed at which I can develop my game in the short-term (though would be an advantage for making future games). It is also not desirable for my particular project as a Computer Science NEA as such a game engine would abstract a lot of the code meaning less of the time spend making my game would be coding and thus less of the game would be marked.
* Since python uses an internal implementation detail called the GIL (global interpreter lock) in the C implementation of python (cPython) this means that I will not be able to utilise multiple cores in the CPU. All the game will run on one core in the CPU.
* I will also not be using multiple threads for running graphical processing. This is because for pygame to work all rendering needs to occur on one main thread. If my game has performance heavy logic outside of rendering, I might use multi-threading to run game logic on a separate thread.
* I will also not be utilising the GPU for rendering (apart from shaders). While this would help improve performance greatly since bullet hell games usually have many objects present on the screen at once and GPUs are optimised for parallel processing, pygame does not have a simple way of interacting with the GPU and implementing this would take too much development time.
* I will not be spending much time on the graphics or music of my game since this would extend development time. Instead I will use songs I like from other games and credit the original artist. As for graphics I will use pixel art to minimise time spent drawing game assets.
* The game will not feature downscaling as my game’s assets being pixel art means that any form of image compression will lead to pixels being lost or distorted at lower resolutions. The game will however feature upscaling to higher resolutions such as QHD (Quad High Definition) or 4K.
* My game will not feature an online leaderboard or any online elements, since that would require self-hosting a server that would take money to manage as well as time to set up.

## Outlining computational methods.

### Abstraction

Designing a game means a lot of the features have to be abstracted in order to create a scalable project that can be improved upon with more defined methods.

* The inputs of my game will come from the mouse and keyboard, with the users monitor/laptop being the output screen.
* The game will require multiple game states (A ‘state’ in this context refers to distinct screens in the game, such as the main menu or gameplay):
  + A title game state which appears upon opening the game. This includes the main game modes such as playing the game, accessing the editor and options states as well as exiting the game. It will feature the most developed graphical user interface as the main menu is what is going to be seen the most out of any screen in the game (not counting amount of time spent looking, just the amount of times it is displayed).
  + An options state with further sub-states for different categories for options. These include things like changing screen size, keybinds, graphics quality etc.
  + A game state where the game will occur. While this is counted as one state, it will encompass ever stage of the game which will be loaded in from files.
  + An editor state where the user will be able to make stages as described in my analysis of Osu! And Geometry Dash.

Other states will also be included that bridge between these states such as popups, but the states mentioned above are the main parts of the game

* The game will also require multiple UI objects:
  + A button object that can be clicked in order to perform an action. The button can have a text or image label.
  + A popup object (also counted as a state) with a text title and other UI elements below (such as buttons or an input text box)
  + A dropdown object that functions similarly to a button but when clicked reveals a list of other buttons that can be clicked in order to change the selection of the main dropdown element.
  + A combo-box object. This functioned identically to a dropdown, but the main element is a input text box that can be used to search through the elements of the dropdown.
  + A text input box object that functions like a button that can be typed in and features most of the features of a simple text editor (cursor, deleting highlighting)
  + A slider object which features a main bar and a slider head that can be dragged in one direction within the bounds of the bar.
* When in the actual gameplay portion of my game, entity objects will be required:
  + A player object for all the selectable characters that allows for moving around and stores data such as health bullet type etc.
  + An enemy object for all the different enemy types (except bosses) that stores the enemies health, attacks, as well as handles logic like when to attack and where to aim.
  + A boss object for controlling how boss enemies function (they are different enough from regular enemies to require their own object), their attacks and movement patterns.
  + A bullet object for creating bullets with different behaviours. This will define the movement of the bullet as well as its shape etc.
  + An attack type object that groups bullets into specific attacks for use by enemies. It stores the bullets start locations as well as paths and any special abilities such as homing or loose player tracking.
  + A bomb (special ability – this hasn’t been named yet) object for the players activate-able ability and its properties.
* My game will feature a state machine which will be an abstraction of different screens in my game. Each state will be a screen: e.g. the options state for the options screen, the game state for the screen where all gameplay occurs, and less obvious states like a popup state for displaying popups over other states.

### Visualisation

My game will require a large amount of assets in order to run.

* My game will use spritesheets in order to combine some related sprites into one file. Spritesheets work by having one image file have multiple sprites that will be unpacked using a JSON file of the same name. The file has the name of each sprite within the image as well as its relative position in the image. All sprites/spritesheets are stored using the PNG (Portable Network Graphic) file ending for the lossless quality. Then I use a function to create subsurfaces of the original spritesheet image in order to extract the sprites out of the spritesheet. Spritesheets/sprites include:
  + The player spritesheet which has different sprites of the player looking in 4 directions. There is also a variation of the player spritesheet with the player’s hitbox shown. Every player also has a player mask sprite attached that consists of an all-white shape that indicates the hitbox of the sprite.
  + The enemy sprites.
  + Bullet sprites.
  + Button spritesheets for every state of the button (default, hovered and clicked). My game uses custom hand drawn buttons for more important buttons in my game. These use the above mentioned spritesheet.
  + Game ‘splash art’: splash art in games is a visual representation of a character or key moment from a game that players see when they first start the game or during loading screens.
  + The game’s backgrounds for each state of the game such as options and each stage of the gameplay.
* The game will also feature 2 custom .ttf (True Type Font) fonts. One that will be used for all of the game and is pixelated to match the rest of the art, and one that will be used for text input and anywhere, where pixelated text would be unreadable.
* The game features many audio files all in .wav (waveform) form for the lossless quality It provides. Audio files include:
  + Main menu music
  + Stage soundtracks
  + SFX (special effects) for UI elements e.g. button click
  + SFX for bullets firing
  + SFX for enemy being hit and dying
  + SFX for player being hit as well as casting abilities
  + SFX for boss casting specific attacks

### Logic and Pipelining

Pygame’s main game loop works in a very procedural way. The way I have implemented my game loop is:

1. Check to see if the game is still considered running (no quit events have been passed)
2. Grab the events from pygame’s event queue using pygame.event.get() and pass them to a centralised event manager singleton class. (The Singleton is a design pattern that restricts a class to only one instance, which is globally accessible.)
3. Handle the events using an centralised singleton event handler class that uses the observer algorithm. (The Observer design pattern allows objects (observers) to subscribe to updates from a specified subject (in this case an event).)
4. Update the currently displayed state by calling its update method. This method handles the states logic.
5. Render the currently being displayed state by calling its render method. This method handles the states rendering.
6. Tick the internal clock forward by the game’s refresh rate.
7. Run pygame.display.flip() which clears the entire display preparing it for the next render.
8. Repeat

Every loop the game’s critical conditions are checked. Some critical conditions happen outside the scope of any displayed state.

These are:

* From the event manager:
  + Has a quit event occurred?
  + If any, which mouse buttons are being pressed?
  + If any, which mouse buttons being held down?
  + If any, which mouse buttons being released?
  + If any, which keyboard keys being pressed?
  + If any, which keyboard keys being held?
  + If any, which keyboard keys being released?
  + If any, which custom pygame events are in the event queue?
  + Is the existence of any of the above mentioned events being queried? If yes:
    - Are these events currently happening?
* From the event handler:
  + If any, which inputs relevant to currently bound actions are being detected by the event manager?
* From the audio manager:
  + Is audio being requested be played? If yes:
    - Is the channel busy playing audio?
  + Is audio playback being requested to be stopped?
  + Is the channel’s volume being changed?
* From the state manager:
  + Is a new state being requested to be displayed?
  + Is the currently displayed state being requested to be removed?

The aforementioned checks happen concurrently to the ones relevant to states mentioned below. Conditions relevant to states differ based on which state the person is in currently, however there are some global checks that are checked for all states:

* Is the state being displayed? If yes:
  + Has the state just been entered?
  + Is the state about to be removed?
  + Has the main display changed?
  + Has the state been requested to update its logic?
  + Has the state been requested to render itself to the screen?

There would normally be more checks relevant to each state, however because I have introduced an event manager and handler, all the critical conditions are checked within the class, meaning there are less checks along the lines of: if button right is pressed, move character right. These are replaced with statements rather than checks that happen every loop: Bind the right mouse button to right character movement, or, unbind the right mouse button from moving the character right.

## Specifying game requirements.

### Development hardware used

The game is being made on multiple computers, each with different hardware components. For the sake of identifying the minimum game development hardware requirements, I will use the worst performing system to identify my requirements.

* The game is being made on a 1920x1080 (FHD) display.
* The game takes up TODO storage (including documentation files as well as the virtual environment) as well as around 700MB for the Pycharm IDE (Integrated Development Environment).
* The game takes up TODO RAM when running the game as well as the PyCharm IDE and 10 tabs in the Firefox browser. I consider this a good representation of what developing the game would look like in terms of load on the RAM.

### Hardware requirements

When the game is deployed, it will require:

* A monitor which has a resolution of at least FHD. The game will not feature downscaling as my game being pixel art means that downscaling will lead to pixels being lost at lower resolutions. The game will however feature upscaling to higher resolutions such as QHD or 4K The game will run at a tickspeed of 165 ticks per second internally, meaning no matter how many frames per second (fps) are being displayed on the screen, all the game data and states update every 165th of a second. The game will however be compatible with running at refresh rates higher and lower than this internal update speed. The game will compensate by calculating the delta time in between each frame and adjusting the players speed accordingly.
* 4GB of free RAM to run the game (16GB total RAM recommended). More RAM recommended so that users can open other windows such as a browser while running the game allowing for a smooth experience.
* x86 64-bit CPU (Intel / AMD architecture). ARM CPUs are not supported.
* At least 5GB of free disk space. This is for storing game assets such as sprites and music, as well as storing the users game save files to allow for saving the users progress in the game.
* A mouse and keyboard with a recommended 3 key rollover. Otherwise, there might not be enough keys on the keyboard (depending on the keyboard’s form factor) for every keybind in the game to be bound to.

### Development software used

As mentioned above the game was developed on multiple computers. This includes multiple operating systems: Windows 10, 11 and Linux Mint 22 Cinnamon Edition. Each had the software requirements fulfilled. Though I have not tested the game on macOS, all of the libraries should be compatible with a mac (in theory). The software requirements for development include:

* Windows 7 or higher, any Debian based Linux distribution – preferably Linux Mint 22 or higher, (since Linux Mint is based on Debian, most Debian based distributions should run my game – though other distributions may still run the game as well) or possibly macOS (not tested).
* Python 3.12 or higher since my game’s code uses new features introduced in python 3.12.
* Pip 24.2 or higher.
* Pygame 2.6.1 or higher.
* Pycharm Community 2024.2.4 or higher, or any other compatible python IDE.

### Software requirements

Software requirements when deploying are identical to the ones required for development, the only difference is that Pycharm is not required since when playing you do not need to open the source code of the game.

## Final success criteria

### The scope of the solution

Having done all of the research and planning for my project, I will now identify key criteria that must be fulfilled in order to finish the game. I will begin with identifying what needs to be included in the final game.

1. Fully functional code for my game.
   1. A menu system.
      1. A main menu.
      2. An options menu.
         1. Graphics options.
         2. Keybinds options.
         3. Gameplay options.
         4. Audio options.
      3. An editor menu.
         1. Stage load/saving menu.
         2. Stage editing menu.
   2. Interactable UI elements.
      1. Buttons with labels.
         1. Text labels.
         2. Image labels.
      2. Input box with simple text manipulation features.
         1. Blinking caret displaying the location of the cursor in the text.
            1. Arrow keys adjust caret location.
            2. Clicking on the text moves the caret to the clicked position. If the clicked position is inside the text box but not inside the text the position is rounded to the beginning/end of the text.
            3. Home moves the caret to the beginning of the line.
            4. End moves the caret to the end of the line.
            5. Control + left and right arrows moves the caret forward or back a word.
         2. Text filtering if it is too long or invalid characters.
         3. Deleting text with backspace and delete.
         4. Pasting text from the paste buffer.
         5. Highlighting text by dragging mouse or control + a to highlight everything or shift + left or right arrow to start highlighting from the caret location.
            1. Moving highlighted end are forward and back with shift + left and right arrow.
            2. Copying highlighted text with control + c.
            3. Pasting over highlighted text with control + v.
            4. Cutting the highlighted text with control + x.
            5. Typing in order to replace highlighted text.
            6. Deleting all the highlighted text with backspace or delete.
      3. Dropdowns.
         1. Text input for the dropdown value.
      4. Sliders
         1. Text input for the slider value.
   3. Keybind features.
      1. Ability to bind customisable keybinds including the keyboard and/or mouse.
         1. Ability to bind combinations of keys/buttons.
      2. Ability to remove/restore keybinds to default values.
   4. Gameplay features.
      1. Players.
         1. Multiple selectable players.
            1. Each player has their own special abilities.
         2. Multiple selectable bullet types.
      2. Enemies.
         1. Each enemy has their own attacks.
         2. Simple enemy movement patterns.
            1. Predefined path.
            2. Random with limitations.
      3. Bosses.
         1. Each boss has a set of complex attacks.
      4. Stages.
         1. Stages have multiple waves.
            1. Normal enemy wave.
            2. Boss wave.
         2. Game has multiple stages.
         3. Each stage is themed.
      5. Items.
         1. Enemies drop items upon death.
            1. Items that level up the player’s attack.
            2. Items that increase the player’s score.
            3. Items that give the player more health.
            4. Items that give the player more special abilities.
      6. Bullets.
         1. Follow a straight line path.
         2. Follow a curved path.
         3. Feature simple player tracking.
         4. Special bullets (usually part of boss attacks).
            1. Can bounce.
            2. Strange pathing.
            3. Spawn more bullets.
            4. Other unorthodox effects.
            5. Lasers.
            6. Large obstacles
            7. Other unorthodox projectiles.
   5. Editor features.
      1. Loading in/saving stages from files.
      2. Playtesting stages.
         1. From specific waves
         2. From the beginning
      3. Adding waves of enemies
         1. Adding enemies to a location on the screen in a wave.
         2. Controlling how a wave ends.
            1. By killing all enemies.
            2. After a set amount of time.
            3. Combination of both of the above.
      4. Adding a boss wave.
         1. Adding boss attacks.
         2. Controlling how boss attacks end.
            1. By dealing enough damage to the boss.
            2. After a set amount of time.
            3. Combination of both of the above.
2. Assets for all parts of my game.
   1. Art.
      1. Backgrounds.
         1. For menus.
         2. For stages.
      2. Sprites/spritesheets.
         1. For players.
         2. For enemies.
         3. For bullets.
      3. UI graphics.
         1. For buttons dropdowns etc.
         2. For the editor.
      4. Splash art
   2. Music.
      1. Background music.
         1. For the main menu.
         2. For when paused in game.
         3. For different stages.
         4. For bossfights.
      2. SFX.
         1. For clicking UI elements.
         2. For bullet firing.
         3. For bullet hitting.
         4. For players/enemies being hit.
         5. For enemies dying.
         6. For special enemy attacks.
   3. Fonts.
      1. For UI elements.
      2. For text input.

# Design

## Structure of the solution

### Flow chartA group of white rectangular signs Description automatically generated

## Decomposition of the problem

In this section, each module/function will relate to a box/set of boxes in the flow chart of the structure of my game as seen above.

### Music handler

The music handler will be a class that when initialised, will create a new unique pygame.mixer.Channel instance. Audio will be able to be added to the channel with a tag to identify the audio file without the need of specifying the directory to the audio file each time. After all the relevant audio files have been added, they will be able to be played back and paused using their relevant tag.

I decided on using a music handler as music should run independent of any actions happening in the game, however, should also be accessible from any point in the game. By creating singleton classes, it ensures that the class is available from any scope. It also allows for lazy-loading audio files. Lazy loading is an optimization technique used to improve game performance by deferring the loading of non-critical resources until they are needed. This approach helps reduce initial page load times and enhances the user experience by loading only the essential content first.

### Input manager

The input manager will be a module containing functions that will be used to query if an input is active as well as the type of input. In order to run my game, more detail will be required for inputs than just ‘is the input active’. For example, for key management the input manager will be able to track and be queried on when a key is being pressed, when it is being held, and when it is being released (a key press and release only happen at the beginning and end of the key input being active). These states will be stores in a stack. This ensures all the components of my game can check which input being held down was pressed first/last.

I used this method of managing input in order to have every module have access to the game’s input instead of having to directly link to the pygame.event.get() stack. This also allows me to define my own event types such as a key pressed event (whereas pygame.event.get only notifies about key up and key down events, and doesn’t indicate if a key was already being held down)

### Event handler

The event handler also runs separate to the gameloop. It is a centralised singleton class that utilises the observer pattern to bind methods to be called when a specified event is detected by the input manager.

While it doesn’t make sense to use the event handler for objects in my game that require consistent updates rather than a one-off call, it is still useful in my game because of its efficiency. The observer design pattern is useful for keeping the input checking logic outside of classes that don’t require them e.g. global keybinds such as ALT + F4 for force quitting the game. Similar to the input manager it also allows for the event manager to be used in a global scope throughout my game (by importing when needed).

### State manager

In order for the game to be displayed, a State manager is needed. This will be a centralised singleton class that stores a dictionary as well as a stack in the form of a list. The dictionary will be used to store a reference to each state class to keep all the states centralised and avoid having to import all the states whenever they need to be switched, while the list will be used for quick appending and popping from a LIFO (last in first out) stack for easy tracking of how to go ‘back’ through states. The state manager will manage all its connected items: the options menu and all its screens, the editor, the game state and quitting the game. The state manager will be used for the 4 main parts of my game: the options menus, the main game, the editor, and handling quitting the game from any state.

The state manager is used to decouple the state managing logic from any one state. It also uses a singleton in order to keep its access global across the game ensuring any state can control the state stack, however still ensuring the state stack is protected under a layer of abstraction.

### Quitting

When quitting the game through any means, whether by pressing the quit button, or a key combination like ALT+F4, the state manager will call the active state’s ‘cleanup’ method that is usually called when a state is no longer at the top of the state stack. This ensures no data is lost and centralises exiting logic.

A cleanup method is required to ensure no data is accidentally lost when a state is forcefully exited. It also ensures no errors are raised and any processes can finish executing safely. This reduces the games volatility.

### Options

The options menu will be a state class that contains buttons that lead to more specific options such as keybind or graphics options. Each of those will also be a class of their own.

Allowing the user to change options in the game rather than through a configuration file allows for a clean user experience and minimises the need of restarting the game.

### UI collision checks

For each menu state class, collisions with UI elements such as buttons will need to be checked. This will be done from within the button class using an update method called from the class the UI element is part of.

Keeping button collisions separate from sprite collisions simplifies the collision manager used for sprites at the cost of minimal performance. I deemed this a worthy trade since performance isn’t a concern in menus as much as it is during gameplay. Having each button manage its collisions also makes sense since while in the game 2 sprites are colliding, in a UI it is always the mouse cursor that is colliding with a button, which is a point on the screen rather than a shape. Also this allows for both rectangle buttons as well as buttons with different shapes using pixel perfect collision (masks).

### Playing the game

The game will run as a state class that loads in a file that includes data about the waves and their enemies as well as stages. Why I chose this approach is explained in the editor section.

### Pausing the game

Pausing the game will halt all updates to the sprites in the game but keep them displayed.

It will allow to access the options screen and all of the settings from within the game, as well as provide a way of temporarily stopping and saving the game to a file if needed.

### Collision management

For collisions happening in the game, a centralised singleton collision manager class will be used. This class will keep track of all the active sprites and collide them as needed. It will check whether the player is collided with enemies or their projectiles, then check if enemies are collided with the player projectiles. It will then call the needed methods of both colliders.

This keeps collision checking in a centralised location while having collision logic present in the sprite. The sprite will be able to update differently based on which sprite colliding. This will be done by passing in the collided sprite when calling each sprite’s collision method to ensure it is aware of what hit the sprite and what it should do.

### Updating entities

After collisions have been detected, both players and enemies and their projectiles will be updated by calling their update methods. This will update the enemies and bullets positions on the screen, as well as update the player sprite based on pressed inputs.

I split the updating of each sprite and UI element into an update method and blit method. ‘Blit’ is the pygame word for drawing a source pygame surface object onto another surface object (usually the screen). The blit method handles updating the object visually while the update method handles logical updates such as sprite health or input handling. Updates must happen after collision checks to ensure processing time isn’t spent updating a sprite that is already dead. Rendering must happen after both collision checks and updates as what the user sees on the screen should reflect the latest state of the game.

### Editor

The editor will be the place where the gameplay of my game will be able to be made. Rather than hardcoding all the gameplay into the game I decided to make an editor for my game and make all the gameplay through that system.

This reduces the time spent making the game at the cost of increased upfront programming complexity. It abstracts seemingly endless enemies into concrete game events that can be saved and stored in files. It also provides scalability in the form of ability to add more stages/alternate game modes.

### Saving a stage

Once done developing, individual stages will have the ability of being saved to a JSON (JavaScript Object Notation) file that can be read to continue editing. A collection of stages will be able to be loaded in and played like a normal game using a metadata file that links all the relevant stages together. Linking stages together into a game will have its own menu and UI in the game.

I chose to save into a JSON file as they can be easily translated into readable dictionaries using the json package natively included with python. Using dictionaries to store information on stages, waves and enemies is forced by using .json as the file type, however by far not the worst way of representing this. A big dictionary of stages can store waves that in turn have dictionaries of the enemies in the waves with dictionaries for the enemy’s identifier (to identify which enemy to spawn) as well as metadata such as the enemy’s spawn location. The enemy’s base data such as health and attacks is stored in the game directly since it doesn’t change within the same enemy identifier.

### Creating stages and waves

Once in the editor, you will have to first create a blank stage. A stage can have its own background and song, as well as regular and boss enemies. Inside individual stages waves will be able to be created. There are 2 types of waves: an enemy wave and a boss wave.

This abstraction helps separate the endless feeling waves while playing the game into concrete collections of enemies that can be stored as dictionaries in the json files mentioned above.

### Enemy wave

An enemy wave consists of a group of enemies that appear on the screen at the same time. Waves can either be ended by killing all the enemies in the wave or running out of time for the wave. If a timer is tied to the end of the wave, the enemies will leave the screen upon the timer running out. When a specific wave is selected, the user will have the ability of picking enemies out of an enemy list and placing them on the screen. Other forms of editing such as removing a placed enemy as well as moving an already placed enemy will also exist.

I abstracted the making of waves into a collection of enemies and their spawn locations on the screen. This means instead of storing the spawn locations and time at which they spawn, no timing has to be involved as all enemies in a wave spawn at once, and the next wave starts after the previous one is finished (at which point it will somehow notify the next wave to start). While in a wave spawn times can be staggered, this is always relative to the wave start, which is easier than somehow timing every enemy spawn and somehow having it adjust based on the speed of enemies defeated. The end of waves being abstracted and decomposed into waves that end when enemies are defeated and waves that end after a specific time (at which point all enemies despawn) also helps each wave have a relative time instead of absolute time as none requires knowing the time since the stages started. All timing happens within a wave.

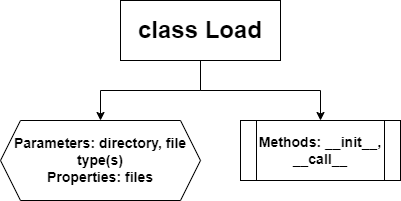
### Boss wave

A boss wave consists of a single boss enemy, which has specific attacks tied to it. These attacks will be grouped into their own ‘waves’ in that they will either end upon enough damage being dealt to the boss, or by running out of time. Boss attacks will either be specific attacks or general attacks. General attacks will select from a pool of general attacks the boss can perform and use it. Specific attacks are reserved for complex attacks used at specific points during the bossfight. Boss attacks will be picked from a list of attacks similar to picking an enemy in an enemy wave. They will then be able to be placed on the screen or removed.

Since boss enemies contain complicated attacks and are one enemy it wouldn’t make sense having them as standalone waves, however if I make a special type of wave where the enemies are replaced with boss attacks the structure makes sense again.

## Identifying and justifying object-oriented techniques.

### Load class



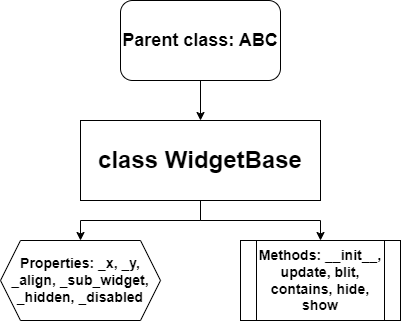
In order to load in all the files from my game such as images, music as well as custom text fonts that are scattered between multiple directories I will use a Load class. The parameters of directory and file type(s) will be used in order to specify where to look for files and which files to look for. The \_\_init\_\_ method of the class will use a recursive walk algorithm to traverse and locate all the files under the given directory even if they are in nested directories. Then, the \_\_call\_\_ method will take in a file name as a parameter and return the absolute path to that file. The property files will be a dictionary storing the name of the file as the key and the file path as the value. A class is useful for this case as the class can be instantiated multiple times with different file types in order to separate files logically (e.g. an instance for all images and an instance for all music).

### Audio class

### C:\Users\r.timantsev\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\959CA29E.tmp

The Audio class will be a class that when initialised, will create a new unique pygame.mixer.Channel by adding one to the \_channel\_counter class attribute that counts the number of times the audio class has been initialised. The unique channel will be stored in channel\_id defined in the \_\_init\_\_ method. It will also initialise the channel (the actual pygame.mixer.Channel instance), no\_audio (an identifier that specifies if an audio can be played from a suitable audio device), sounds (a dictionary storing the name of a sound as a string tag identifier as the key, and the path to that sound file as the value) and current\_audio (a property that stores the tag of the currently playing audio). Audio will be able to be added to the channel using add\_audio with a tag to identify the audio file without the need of specifying the directory to the audio file each time. After all the relevant audio files have been added, they will be able to be played back and paused using their relevant tag. The play audio method takes in the tag of the audio to play, the number of loops to play it for, and whether to override the previously playing audio if it was playing. The last argument allows sound effects like button clicks to always override the last click if it was playing, while background music to never override so the music doesn’t reset to the start of the song each time play\_audio is called. The other volume methods just manipulate the volume of the channel in specific ways such as setting it to a specific volume number, or incrementing or decrementing it by an arbitrary amount that is relative to what the volume was set to before. The stop method just stops the currently playing audio. A class is useful in this case as the existence of class attributes means each object can be assigned a globally unique channel that can be used across my game.

### Widget base class



The widget base class is meant as a parent class for every UI element used in my game. It inherits from ABC (Abstract Base Class) in order to specify that the update and blit methods are abstract methods and must be overridden. The \_\_init\_\_ method initializes all the properties. \_x and \_y are the coordinates of the widget, while the \_align property specifies to which point on the widget the coordinates should reference (for example top left, centre, bottom right). The \_disabled property indicates if all interaction with the widget is disabled, while \_hidden indicates if the widget is being drawn on the screen or not. The \_sub\_widget property doesn’t serve any purpose in the base class, however needs to be a property of all widgets so widget handlers don’t get exceptions saying the property doesn’t exist. It is meant for widgets that are part of other widgets: for example when a popup widget has text (which is its own widget) or a dropdown widget is using buttons. The contains method doesn’t have any collision logic since this is a base class but includes a check for if the widget is \_disabled and if yes, just returns False. A class is useful in this case (and for all other cases where a class is mentioned as being a ‘base’ class) as this being a base class inherently implies inheritance will be used. Since multiple objects of widgets will be used throughout my game it makes sense to use classes to achieve disconnected instances with unique instantiation parameters.

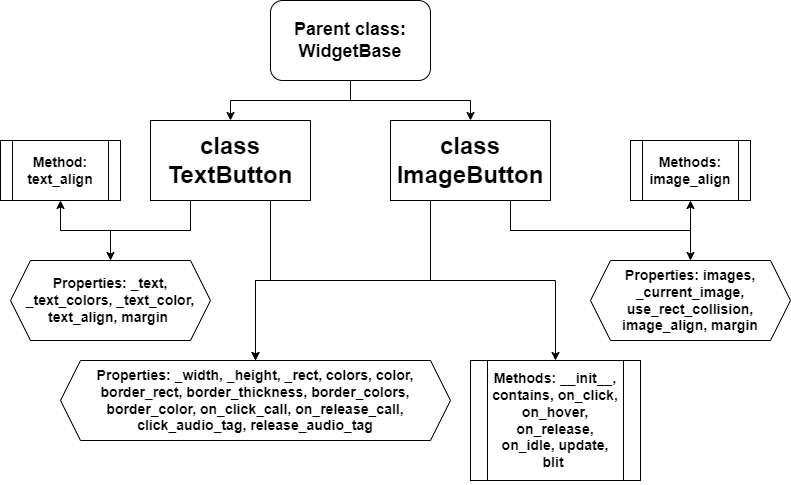
### Text class

A diagram of a class text

Description automatically generated

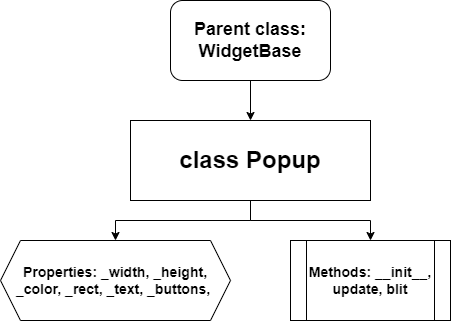
The text class is a widget for placing text on the screen. I have left out all the properties and methods talked about in the parent class, and only included the ones unique to the text class. The \_text property stores the actual text to be displayed, \_font stores the pygame.font.Font instance for the text, \_font\_size stores the size of the font, \_text\_surface stores the rendered text surface that can be blitted onto the screen or other widgets, and \_rect stores the location and dimensions of the bounding box of the text on the screen. The \_align\_rect method aligns the bounding box of the text allowing it to move, while the \_render\_text method renders (or re-renders) text onto the \_text\_surface which can be used if e.g. the font or font size changes.

### Button classes



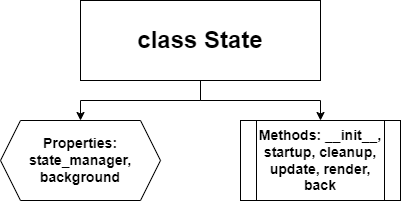
The button class is a widget for placing other widgets that can be interacted with. A regular button consists of a background rectangle (\_rect) which detects if it is clicked or hovered (detected in the update method), and a label (either text or an image) drawn on top. The background rectangle and the text needs a tuple property to describe the 3 colors the rect/text will be at each state. Alternatively an image can be used as the entire button. In this case pixel perfect collision will be used on the image (or if the use\_rect\_collision property is True the rect collision will still be used if the image was rectangular anyway and using pixel perfect collision would be a performance loss). This works by having the button image have a transparent background and colliding if the mouse is touching any part of the image that is not transparent. In addition to this, 3 images will need to be used. One for the button if clicked, one for if it is hovered, and one for when it is neither clicked nor hovered. The button will be able to have a sound played and/or function called when the button is clicked or released. If I have enough time I might implement a way of navigating through buttons and being able to select them with the keyboard only.

### Popup class



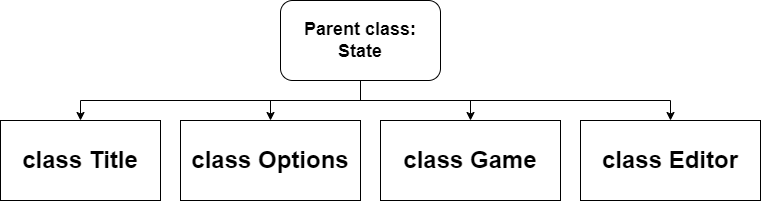
The popup class is a widget class inheriting from WidgetBase (as all widgets do) that creates a box on the screen with text and/or buttons in order to notify the user of something. This box can be exited by pressing escape, clicking outside the box or on a button for exiting out of the popup (if applicable). The popup blocks updates to the currently displayed state, however still allows it to be drawn onto the screen. Along with the properties defined in Widgetbase, it additionally has a property for the background box: \_rect, as well as its \_width and \_height. \_color describes the color that the background box should be, and the \_text and \_button properties function similarly to an OR gate in that text and buttons can be present, or only one of them, but never none of them (otherwise the popup would literally just be the background box). Exiting via pressing outside the popup box and pressing escape functions can be disabled if the popup is important and the user must choose one of the options (however you can't disable this functionality if only text is present, because the user would have no way of exiting the popup then). The \_\_init\_\_ method defines all the properties as well as aligns the buttons, text, and rect. All aligning of text and buttons inside the popup is handled automatically inside the class: e.g. if I pass in only text, it gets placed in the middle of the rectangle, and the same happens with just one button. When more than one button is placed they are placed in the middle vertically, however horizontally they are spaced equally across the width of the button. When both buttons and text are present, the text is positioned above the buttons equally spaced vertically with the height of the button. This ensures all math on the position of the sub-widgets of the popup occur inside the class. The update method updates all the sub-widgets part of the popup, while the blit method draws the background rectangle as well as the sub-widgets onto the screen.

State class



The state class is a base class for all states in my game. It has a state\_manager property which is a reference to the state\_manager instance for managing states, and a background property which is the object that gets drawn at the beginning of each game loop. The background property defaults to a black surface so new states don't instantly get errors when not defining a background, however the property is meant to be redefined when inheriting. The \_\_init\_\_ method defines the above mentioned properties, the startup and cleanup methods are methods that are called every time the state is entered and left. In the base class it registers and deregisters the escape key (in the input binder class) from the back method (the purpose of this is so that each state can define its own unique back method, or remove it entirely: e.g. while going back to the previous state is the default functionality of the back method, the first method entered is at the bottom of the state stack so going back to a previous state would lead to an error (there are no states to go back to), so it must define the back method to actually append a state to the state stack instead). The update method is meant for all logical updates to the game and the base functionality is to update the Widget handler, while the render method is meant for drawing every object onto the screen and it blits the background to the screen and then blits all the widgets from the Widget handler. I implemented this since most of the states that inherit from the state class use these features and it is easier to not have to worry about basics like rendering the background in all child states.

### Game state classes



While I will probably make more states for screens such as the credits screen or a paused screen for when in the game, the 4 main states of my game are the title, options, game and editor states. These will all inherit from the state class, however will all be different.

* The title screen will be the first screen the user sees when opening the game. It will feature title art as well as buttons to access the other 3 states. Pressing escape in this state will enter the options since it is the first state. It will also have its own background music that will keep playing in the options and editor states, however will stop in the game state as the game music starts.
* The options state will have a lot of widgets for changing the settings of the game. These include graphical settings like the screen size, refresh rate etc., audio options such as background music volume, sound effects volume and an option to mute the game when not in focus (minimised) and keybinds options for changing the functionality of some keys such as the key to shoot bullets or the keys to move around.
* The game state will house all the gameplay of my game. It will include the character and enemies that will be loaded in using a file defining the order and type of waves, made using the editor.
* The editor state will have all the features outlined in my decomposition of the problem: a main grid for placing enemies, a timeline at the top for managing the order waves appear, a sidebar for selecting which enemies to place, and a way of saving the current stage to a file.

### Entity class

A diagram of a class entity

Description automatically generated

The entity class will be a base class for all sprites in my game. It will inherit from the pygame Sprite class in order to add more utilities to the class. This will be done by adding attributes such as \_spawn to store the coordinates of where the sprite should appear on the screen (when first drawn). The spawn\_alignment will specify to which part of the sprite does the \_spawn coordinates refer to. The sprite property will hold the pygame.Sprite object and the \_rect property will store the sprites hitbox. The \_abs\_rect property holds the actual bounding box of the rectangle (for situations where the sprite’s hitbox is smaller or bigger than it appears. The \_\_init\_\_ method will initialise all of the above-mentioned properties. The move\_to\_spawn method will reset the coordinates of the sprite to the ones specified by \_spawn. The on\_collide method will be called by the collision manager and pass the object of the entity it is colliding with. The method will house logic that will react accordingly to the entity received. The update method will be called every game loop and handle logical updates to the sprite while the blit method will draw the sprite onto the screen.

### Player and Enemy classes

A diagram of a class

Description automatically generated

The entity class will be the parent class for both Player and Enemy classes. The player class will be a base class for every playable character in my game while the enemy class will be a base class for every enemy (including bosses, which will have their own base class that inherits from the enemy class). They both share the health, x, y and speed properties as each has its own health value and x and y coordinates of their location on the screen.

* The player additionally has a game property which is an instance of the current game state so that the player can interact with it. The spritesheet attribute holds any extra animation frames the sprite might have like animations when moving shooting or casting abilities. I might implement this into enemy classes in order to animate them as well later. The keys will be a list currently pressed keys (specifically movement keys like the arrow keys or the wasd keys). The list will function like a LIFO stack where keys pressed later are appended to the end of the list in the order they are pressed and removed in that order when the key is released. The purpose of this is to have the player’s direction be based on the most recent key pressed. While I haven’t designed the movement system yet, I plan to use the dx and dy properties to indicate how much a player should move and use the move\_ip inbuilt pygame method to move the player that amount. The move\_ip method updates the coordinates of the rect specified (in this case the player’s hitbox) by adding the values to them. This means every game loop I will calculate the dx and dy values by checking for movement keys in the x and y direction that are at the top of the keys stack and multiplying by the speed. This will have the effect of being able to hold left then right and having the player move left and then right even if left is still being held. This creates a more responsive feedback system. While both players and enemies have an attack method they are not shared because they have different functionality: a player’s attack is activated by holding a button down while an enemies is not. The player’s attack method includes logic for what to do when attacking. The ability method will include some kind of ability for the player with a limited use count and cooldown though I haven’t thought of what this is yet.
* The enemy class additionally has its own attack and move functions. The move method will include logic about how to move around the screen. This will usually just be a random point with some extra limitations, so the game doesn’t become annoying. It will be called at random or with a cooldown in the update method. The attack method will include logic for attacking. Every enemy will have a set number of attacks, with bosses having named boss attacks while enemies have hidden attacks that are chosen at random whenever the attack method is called. This will also be called at a cooldown or at random in the update method.

### Projectile class

A diagram of a class project

Description automatically generated

This will be a base class for all the projectiles in my game, player or enemy. Most of the properties and all of the methods relevant to the projectiles in my game are already defined in the entity class so they don’t need to be mentioned here. It simply includes a dx and dy property for a similar reason that the player class includes them. While players and projectiles need to update their positions from their last position, enemies simply need to know where to move to, as the path is preset and I don’t plan to make any enemies that follow the player, so dx and dy is not needed for them. The bullet will be updated using the update and blit methods and logic such as direction of the bullet can be calculated using some trigonometry. More complicated bullets however will need more calculations if the pattern is more unorthodox (e.g. curving bullet paths).

## Identifying and justifying other data structures.

### Global variables

While global variables are discouraged in most programming languages, they seem necessary when global settings are needed to be changed and kept throughout the entire game. I plan to solve this issue by using a dataclass in its own module file (while this is an object-oriented technique it is more of a substitute for global variables so I decided to include it here) in order to store any global variables such as refresh rate or game settings set in the options screen like fullscreen.

The other global variables I plan to include are module level globals used in general game logic such as \_running and \_playing (to be used in the mainloop to check if the game is being played e.g. not in any menu, or still running or meant to be exited) or \_clock to store a global instance of pygame.time.Clock() so that it remains in sync throughout every scope in the game. Globals for file management such as a sources\_root variable to store the root directory of the game are also useful for making sure all files are loaded in using a generalised directory depending on the OS and location of files on the system. A dt global variable will also be required for making sure the game is not FPS dependent (e.g. runs faster when at higher refresh rates and slower when at lagging).

### Lists, tuples, dictionaries and sets

While as I have already mentioned the numerous uses of lists, tuples and dictionaries, I will reevaluate the purpose of the most important uses of these data structures.

* My state handler will have a dictionary of the name of every state in my game as the key and the class that name refers to as the key. This allows me to refer to all of the states without importing them, avoiding circular import errors.
* The above mentioned key names will be used in a LIFO stack which will store a stack of the currently active state at the top of the stack. When a user exits via a back button the state will be popped off the top of the stack and the state below it will be loaded in as the new active state. If there is only one state in the stack, trying to go back will instead append the options menu to the state stack. This is also useful for states such as popup states, as they can use the state stack to render the state below them and have the appearance of transparency.
* A dictionary will be used to store all the data loaded in from files such as audio and images. As mentioned previously the Load class will be used to recursively search directories for files of a specific file type, and when found will store the name of the file as a key and the path to the location of the file as the value. This will be used to easily refer to the file path to an asset, since when loading in assets the file path is needed, but referring to the full path every time is tedious.
* Tuples will be used for representing RGB colour codes, since they are constants not meant to be changed, and tuples being faster than lists help optimise game performance. They will also be used to represent coordinate pairs such as the x and y of the centre of the screen or the screen size.
* Lists will be used in the input manager in order to track different inputs. Lists such as keydown\_events or keyup\_events will use a LIFO stack to keep track of inputs in the order they were received.
* While I haven’t thought of any uses for sets in planning, I might have to use some in order to eliminate duplicates or something similar.

### Strings, integers, Booleans, etc.

These are the main uses of each variable type in my game:

* Strings: used for any text to be displayed as well as for some keys in dictionaries.
* Integers: Used for any coordinates or values that makes sense being represented as numbers e.g. the refresh rate or player health.
* Booleans: Used for any method or function that acts as a query e.g. is\_input\_active would return a Boolean.
* Floats: Used for angles when using radians like when calculating a bullet’s trajectory.
* Doubles: I don’t think I will need decimals to the precision of a double as movement with pygame Rects only use whole numbers so the precision and space used would be redundant.

### Storage techniques

To store data for my game I have analysed many popular methods of storing data for each use case.

The ways of storing data include:

* Using a Text (.txt) file. The benefit of this method is that it requires no libraries, as well as is humanly readable. However, it is often impractical to make as there is no ‘language’ in text files so you must create your own rules for you to then follow when reading a file.
* Using JavaScript Object Notation (JSON). This method solves the above-mentioned problem, as JSON files use syntax identical to python dictionaries in order to store data while also remaining humanly readable. In this way, you can create a hierarchy in your data as well as assign values keys in order to identify them better. However you do need to use the json library to parse the file which takes some processing power.
* Using the pickle library in python. This is a way to serialise a python object into a bytestream that is then stored in a file. These files are smaller in size that JSON files however are not humanly readable. It is by nature compatible with python classes however the syntax for serialising and deserialising files does need to be learnt.
* Using an SQL database. An SQL database allows for creating relationship tables and the ability to query large amounts of data with good performance. However if not used with large amounts of data it becomes redundant and writing a select statement to get a piece of data is tedious. It comes with a lot of boilerplate that doesn’t pay itself off unless there is a sufficient enough amount of data, however this does mean that using an SQL database very future-proof in case I decide to make a server for the game and need to store player data. SQL databases can also be hooked up as a backend database for online servers better than the other techniques.

In order to decide which storage techniques to use where, I identified the place I would need to store/receive global data:

* Spritesheets require a form of metadata to interpret and decompile the composite images into specific sprites. To store the metadata of spritesheets (which includes data such as which sprite is where on a larger image, its size as well as the name of the sprite and its order) I decided to use a json file that would have the same name as the image file. There is another way of doing it by having a specific coloured outline around each sprite in order to separate them in order to reduce the amount of files necessary, but I found this method to be too tedious and impractical. While a text file could be used for this purpose as well, I found that the fact I had to create my own syntax rules made using that method way to tedious, however the main reason was that there exist tools to autogenerate a JSON file to accompany a spritesheet, which creates all the necessary dictionaries and the only thing necessary to make is a spritesheet is a loader that looks for the json file of the same name as the image file, and sorts out the data in the dictionaries in order to separate the image into multiple surfaces. Since these tools only exist for JSON, this automatically makes it the most handy option out of all of them. To generate these JSON files as well as positioning sprites in the spritesheet I used a software called Aseprite which is often use for pixel art creation and includes a feature to export a spritesheet with a JSON file. The JSON format is popular with many pixel art artists as well as game developers for making spritesheets and already has all of the documentation written, so this means I don’t have to create and document my own hierarchy of data for the metadata of spritesheets.
* As discussed earlier, global settings (both public and private) need to be stored in an external file so that they persist when the game is closed. I think the best way to store this would also be using pickling. Since the data only needs to be changed in game, it does not need to be readable outside of that and additionally allows for private globals to be stored without the risk of a leak. And since I already plan to have a dataclass to store globals, serialising that class is a natural way of storing the data compactly and efficiently without needing to parse any syntax. In order to read settings I will need to deserialise the file when opening the game and change all the settings accordingly.
* In order to store the game stages that will be made using the game’s editor I will also use the pickle module to store info about the stage in a serialised file. When loading stages this file can be deserialised and read in order to play the game. Using a JSON file seems too complicated here as a stage would most likely have a large amount of data that would each have to be labelled, and loading it in mid game would likely cause a considerable amount of lag. The stages also do not need to be readable outside the game as well so the negative of pickling is alleviated.
* When a game is paused and exited mid stage, some data is required to be stored so that the game can be resumed. I plan to store this data in a JSON or text file depending on the amount of data and the complexity of the data I will need to store. When resuming a game the pickled stage file mentioned above will be loaded in, however extra metadata is required in order to specify where in the stage the player left, their health and other data that is relevant in the middle of a playthrough. While I don’t know how much data will be required I don’t think it will be on the same scale of amount of data required to store the stage itself. However there might be concerns that a player will be able to read the data in the file and modify it in order to cheat themselves extra lives etc. In this case I might consider either hashing the file to make it unreadable to humans or just use the pickle module.

Other decisions were also made about the way of storing other data such as audio files. All audio files will be stored in a .wav (Waveform Audio File) format, since this extension stores uncompressed audio and is popular globally.

All image files will be stored in a .png (Portable Network Graphic) format since .jpg features compression and this does not look good in a pixel art game. The PNG format is also the most popular lossless image format, so compatibility is not an issue (unlike BMP).

I decided to use spritesheets for multiple small sprites such as players, enemies, and buttons since while images such as backgrounds are only loaded in sparsely, images that are loaded in frequently such as sprites benefit from having every images loaded in at once, since this increases performance and reduces the need to constantly load in new images.

## Designing the algorithms to be used.

### General game algorithms

These include algorithms generally used throughout the entirety of the game or only upon startup.

In order to run the game, I will use a procedure to initialise the game for running:

Procedure start:

Checks if the file is being executed directly. If yes:

Initialises the first state remotely.

When the whole game loop ends, attempts to exit pygame.

Stops the code.

In order for the remote initialisation of the state to work a procedure in the scope of the state manager will initialise the state passed to it:

Procedure initialise(starting state):

Add the starting state to the top of the state stack.

While the game is running the main procedure will serve as the game loop:

Procedure main:

Check if the state stack is empty. If yes:

Raise an error.

While the running variable is true:

Process the inputs via the input manager.

Check if the input manager is requesting the game to close. If yes:

Stop the game loop by setting the running variable to false.

Update the top state on the state stack.

Render the top state on the state stack.

Update the global delta time variable.

Clear the display.

In order to load in all of the game data from files a load class is needed as mentioned before. The \_\_init\_\_ magic method of this class is described below:

Method \_\_init\_\_(directory, file ending):

Recursively search the directory specified for files of the file ending specified:

If one is found:

Add the name of the file and the filepath as a key value pair to a dictionary.

Return the dictionary.

To parse the spritesheets that will be used to group sprites a function is needed:

Function parse spritesheets(spritesheet):

Load in the entire spritesheet as a surface.

Load in the JSON file associated with the image.

Check the JSON file for the coordinates of the individual sprites. For each one:

Create a surface of each one and add it to a list.

Return the list.

The input manager gets a list from pygame of every key that is down and adds the relevant keys to all the lists storing the status of all the keys.

Procedure process events(event list):

Clear all key down and key up events.

Clear all mouse down and mouse up events.

For every event in the event list:

Check for the type of event:

If the type is a key down:

Add the key to the key down and held keys lists.

If the type is a key up:

Add the key to the key up list and remove it from the held keys list.

If the type is a mouse down:

Add the mouse button to the mouse down and held buttons lists.

If the type is a mouse up:

Add the mouse button to the mouse up list and remove it from the held buttons list.

If the type is a quit event:

Request the game loop procedure to attempt to stop the game.

Update the mouse position variable with the current mouse position.

### State algorithms

These algorithms will include all of the algorithms used by states or the state manager.

Each state, while having different logic, has the same base methods:

Method \_\_init\_\_:

Load in the background image.

Load in any other UI elements to be used in the state.

Load in any entities to be used in the state.

The update method is ran every game loop if the state is at the top of the state stack.

Method update:

Update any music playing if necessary.

Update all of the UI elements.

Update all of the entities.

The render method is ran every game loop after the update method if the state is at the top of the state stack.

Method render:

Draw the background onto the screen.

Draw the UI elements onto the screen.

Draw the entities onto the screen.

The startup method is called whenever a state is entered, whether it be by being added to the state stack or a state above being removed leading to the one below being entered.

Method startup:

Bind every input to be used to the relevant procedure in the state using the input manager.

Load anything that needs to be refreshed when a state is entered.

The cleanup method is called just before a state is exited whether it be by being removed from the top of the state stack, or a new state being placed on top of the current state.

Method cleanup:

Unbind every input that has been bound in the startup method.

Unload anything that should not persist upon the state being popped.

The following algorithms are used to interact with the state stack with a level of abstraction. The append method is used to add a new state to the top of the state stack.

Method append(state):

If the state stack is not empty:

Notify the current state that it is going to be exited out of so it can end all tasks.

Add the state to the end of the state stack.

Notify the state it has just been entered in order to set up.

The pop method is used to remove the state at the top of the state stack.

Method pop:

If there are no states in the state stack:

Raise an error.

Notify the current state that it is going to be exited out of so it can end all tasks.

Pop the state off the state stack.

If the state stack is not empty:

Let the state below know it has just been entered in order to set up.

### Audio algorithms

All of these algorithms are part of the audio class and are used to interact with audio files separately in different audio channels that can have their individual audio started or stopped, as well as their own volume control.

Method \_\_init\_\_:

Get a unique channel id for this instance of the class using the global channel counter.

Add one to the global channel counter so that the next instance is unique.

Initialise the new channel with the unique channel id.

Create a dictionary for sounds in this channel to be stored in.

The add audio method allows for new audio to be added to a channel.

Method add audio(audio path):

Get the audio file at the audio path specified.

If the audio file does not exist:

Raise an error.

Add a key value pair to the dictionary of the name of the audio file as the key

and an instance of the audio file as a sound object as the value.

The play audio method plays the audio track if the channel is currently not playing anything. While I could make it always override the currently playing audio, this process of handling audio makes it so that it is way easier to implement things like background audio, which shouldn’t restart every time it is requested to be played.

Method play audio(audio):

Check if the audio dictionary has any sounds added. If not:

Warn the user that a sound was attempted to be played that has not been added.

Check if the channel is currently playing another audio. If not:

Play the audio by getting the audio from the dictionary.

The stop method is just an abstracted version of the stop method used in the pygame channel object.

Method stop:

Stop playing any audio in the channel.

The set volume method allows for channels to have their volume specified, meaning that settings can be added that adjust certain sounds e.g. sound effects and music separately.

Method set volume(volume):

Set the channel volume to the volume specified.

### User interface algorithms

These algorithms are used for any user interface elements as well as algorithms for the widget manager.

For any text widget in the game:

Method \_\_init\_\_(position, text, font, font size, color, alignment):

Initialise all of the arguments passed.

Create a Font object using the font and font size.

Align the text to the position using the alignment parameter.

Render the text to a text surface.

The update method is called from whichever state the widget is a part of. It makes sure that the widget remains interactive to inputs and changes.

Method update:

Update any of the properties if needed e.g. position or text contents.

The contains method checks if a point on the screen is within the widget’s bounding rectangle. It is used to make interactive widgets such as clickable text or buttons.

Method contains(x, y):

Check if the point at the coordinates are in the bounding text box. If yes:

Return True.

Else:

Return False.

The blit method is called after the update method from the state that the widget is a part of. It draws the widget onto the screen.

Method blit:

Draw the text surface to the screen.

For any buttons in my game:

Method \_\_init\_\_(position, size, alignment, colours, click audio, on click command):

Initialise all of the arguments passed.

Create a rectangle object of size and colour given.

Move the rectangle to the position and alignment specified.

In the button the update method is overridden to account for the different, more complex functionality of the button. It checks using branching to check if the button is being clicked, released or just hovered over and updates the button accordingly.

Method update:

Update any of the properties if needed e.g. rectangle colour or position.

Check if the button contains the mouse cursor. If yes:

Check if the mouse is currently being released. If yes:

Call the on click command.

Check if the mouse is currently being pressed. If yes:

Call the update click method.

Else:

Call the update hover method.

Else:

Call the update idle method.

The update click method is simply to decouple the click logic from the update method. The click hover and click idle methods also serve this purpose.

Method update click:

Play the click audio.

Update the button rectangle colours to the clicked colours.

Method update hover:

Update the button rectangle colours to the hovered colours.

Method update idle:

Update the button rectangle colours to the idle colours.

For any text button in the game (inherits from the button):

Method \_\_init\_\_(text, text colours, font, font size, text alignment):

Finish running the button’s \_\_init\_\_ method.

Create a Text widget passing all the relevant parameters to it.

Method blit:

Finish running the button’s blit method.

Draw the text widget onto the screen.

Method update:

Finish running the button’s update method.

Update any text properties if needed.

For any image button in the game (inherits from the button):

Method \_\_init\_\_(image, image alignment):

Finish running the button’s \_\_init\_\_ method.

Initialise the image property.

Align the image using the image alignment parameter.

Method blit:

Finish running the button’s blit method.

Draw the image surface onto the screen.

Method update:

Finish running the button’s update method.

Update any image properties if needed.

For the popup widgets the following algorithms are used:

Method \_\_init\_\_(position, size, color, text, buttons):

Initialise all the properties with the arguments passed.

Create a background rectangle of size given.

Place the text and buttons onto the rectangle.

Method update:

Call the text and button update methods.

Method blit:

Call the text and button blit methods.

### Entity algorithms

Entity algorithms consist of the Collision manager algorithms as well as player and enemy algorithms and their bullets.

The update collisions procedure takes in the current game state instance in order to access all of the entities present at the moment, and checks for collisions between each interactable sprite group.

Procedure update collisions(game):

Check for collisions between the player sprite and enemy sprite group.

Pass the result to the sprite collision procedure (below).

Check for collisions between the player’s bullet sprite group and enemy sprite group.

Pass the result to the group collision procedure (below).

Check for collisions between the enemies’ bullet sprite group and the player sprite.

Pass the result to the sprite collision procedure (below).

The sprite and group collisions having their own procedures decouples the logic from the main update collisions procedure allowing for them to be used in more places if more groups that can interact are added to the game. The sprite collisions method handles when one singular sprite needs to be checked for collisions with a group of sprites held in the collisions list.

Procedure sprite collisions(sprite1, collisions):

For every sprite in the collisions list:

Call the sprite’s collision method with sprite it collided with (sprite1).

Call sprite1’s collision method with the sprite it collided with in the list.

The group collisions method handles when 2 groups of sprites need to be checked against each other for collisions.

Procedure group collisions(group1, group2):

For sprite1 in group1:

For sprite2 in group2:

Call sprite1’s collision method and pass sprite2 as the collided sprite.

Call sprite2’s collision method and pass sprite1 as the collided sprite.

The entity base class is meant to be inherited by both players and enemies and has features used by both.

Method \_\_init\_\_(spawn, sprite, sprite hitbox, spawn alignment):

Set all the entities properties.

Call the move to spawn procedure (below).

I plan to make a move to spawn method to decouple the spawn logic from the \_\_init\_\_ method and allow the entity to be moved to the spawn later as well.

Method move to spawn:

Move the sprite to the spawn position using the spawn alignment given.

The blit method will be called after the update method (which doesn’t exist in the base class since each entity has its own unique updates) in order to draw the sprite onto the screen.

Method blit:

Draw the sprite onto the screen at the position of its hitbox.

While I have not thought out what features I want each of my players to have, I will have a player base class inheriting from the entity class that has features every player will have.

Method \_\_init\_\_(game, spritesheet, stats):

Call the entity’s \_\_init\_\_ method.

Unpack the stats dictionary into properties e.g. health, speed etc.

Initialise the rest of the player’s properties.

The update method which didn’t exist in the entity class will be implemented in the player class in order to handle moving around using keys on the keyboard. I will use a seperate

Method update:

Set dx and dy to 0.

Call the input manager’s method to check if a key is pressed.

For every key that is pressed that is a movement key:

If the key is the key to move up:

Set dy to the -speed stat of the player.

Set the current sprite to the sprite in the spritesheet for when the sprite for moving up.

If the key is the key to move down:

Set the dy to the speed stat of the player.

Set the current sprite to the sprite in the spritesheet for when the sprite for moving down.

If the key is the key to move left:

Set the dx to the -speed stat of the player.

Set the current sprite to the sprite in the spritesheet for when the sprite for moving left.

If the key is the key to move right:

Set the dx to the speed stat of the player.

Set the current sprite to the sprite in the spritesheet for when the sprite for moving right.

Else:

Set the current sprite to the sprite in the spritesheet for when

the sprite is not moving.

If the attack key is being pressed down in the input manager:

Call the attack method.

Move the player hitbox by the dx and dy specified.

Clamp the sprite back into the screen boundaries so the sprite cannot go

past the edges of the screen.

Method attack:

Shoot the bullet.

Reset the delay.

Method on collide:

Remove 1 health from the player’s health.

Move the sprite to its spawn position.

Like the player base class, enemies also have a base class with the following methods:

Method \_\_init\_\_(spritesheet, stats):

Call the entity’s \_\_init\_\_ method.

Unpack the stats dictionary into properties e.g. health, speed etc.

Initialise the rest of the enemy’ properties.

Method update:

If health is 0 or less:

Kill this sprite.

If random chance to move rolled and time is more than minimum delay between movement:

Pick a point on the screen and move to it with the enemy’s speed.

# Evaluation